conference

proceedings

2025 USENIX Annual Technical Conference

Boston, MA, USA

July 7–9, 2025

Sponsored by

USENIX

THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

ISBN 978-1-939133-48-9

USENIX

Proceedings of the 2025 USENIX Annual Technical Conference (USENIX ATC '25)

Boston, MA, USA July 7–9, 2025

USENIX ATC '25 Sponsors

Gold Sponsors

FUTUREWEI Technologies

NSF

Bronze Sponsors

Meta

NetApp®

Open Access Sponsor

جامعة الملك عبدالله للعلوم والتقنية
King Abdullah University of Science and Technology

USENIX Supporters

USENIX Patrons

Amazon • Bloomberg • Futurewei • Google • Meta

USENIX Partner

Thinkst Canary

Open Access Supporter

Google

Open Access Publishing Partner

PeerJ

USENIX Association

Proceedings of the
2025 USENIX Annual Technical Conference

July 7–9, 2025
Boston, MA, USA

© 2025 by The USENIX Association

All Rights Reserved

This volume is published as a collective work. Rights to individual papers remain with the author or the author's employer. Permission is granted for the noncommercial reproduction of the complete work for educational or research purposes. Permission is granted to print, primarily for one person's exclusive use, a single copy of these Proceedings. USENIX acknowledges all trademarks herein.

ISBN 978-1-939133-48-9

Conference Organizers

Program Co-Chairs

Deniz Altınbüken, *Google*

Ryan Stutsman, *University of Utah and Stellar Development Foundation*

Program Committee

Reto Achermann, *University of British Columbia*

Bilge Acun, *FAIR and Meta*

Atul Adya, *Databricks*

Irfan Ahmad, *Magnition*

Ibrahim Umit Akgun, *Google*

Ram Alagappan, *University of Illinois at Urbana–Champaign and VMware Research*

Omid Alipourfard, *Jane Street*

Ali Anwar, *University of Minnesota*

Behnaz Arzani, *Microsoft Research*

Godmar Back, *Virginia Tech*

Saurabh Bagchi, *Purdue University*

Yungang Bao, *Institute of Computing Technology, Chinese Academy of Sciences*

Ivan Beschastnikh, *University of British Columbia*

Ankit Bhardwaj, *MIT CSAIL*

Ketan Bhardwaj, *Georgia Institute of Technology*

Stella Bitchebe, *McGill University*

Philippe Bonnet, *University of Copenhagen*

Nathan Bronson, *OpenAI*

Rodrigo Bruno, *INESC-ID, Instituto Superior Técnico, University of Lisbon*

Anton Burtsev, *University of Utah*

Burcu Canakci, *Microsoft*

Zhichao Cao, *Arizona State University*

Maria Carpen-Amarie, *Huawei Research*

Somali Chaterji, *Purdue University and KeyByte*

Xusheng Chen, *Huawei Cloud*

Youmin Chen, *Shanghai Jiao Tong University*

Kang Chen, *Tsinghua University*

Yue Cheng, *University of Virginia*

Vijay Chidambaram, *The University of Texas at Austin*

Steven Chien, *University of Edinburgh*

Young-ri Choi, *UNIST (Ulsan National Institute of Science and Technology)*

Marcello Cinque, *Università di Napoli Federico II*

Landon Cox, *Microsoft*

Jon Crowcroft, *University of Cambridge*

Dilma Da Silva, *Texas A&M University*

Alexandre da Silva Veith, *Nokia Bell Labs*

Eyal de Lara, *University of Toronto*

Murat Demirbas, *MongoDB Research*

Xianzhong Ding, *Lawrence Berkeley National Laboratory*

Han Dong, *Boston University and Red Hat*

Wan Du, *University of California, Merced*

Abhinav Duggal, *Amazon*

Eric Eide, *University of Utah*

Sameh Elnikety, *Microsoft Research*

Pratik Fegade, *Google*

Dan Feng, *Huazhong University of Science and Technology*

Xinwei (Mason) Fu, *Amazon*

Massimo Gallo, *Huawei*

Aishwarya Ganesan, *University of Illinois at Urbana–Champaign and VMware Research*

Wei Gao, *University of Pittsburgh*

Manya Ghobadi, *Massachusetts Institute of Technology*

Yossi Gilad, *Hebrew University of Jerusalem*

Aniruddha Gokhale, *Vanderbilt University*

Kartik Gopalan, *Binghamton University*

Justin Gottschlich, *Merly and Stanford University*

Helen Gu, *North Carolina State University*

Arobinda Gupta, *Indian Institute of Technology Kharagpur*

Suyash Gupta, *University of Oregon*

Kyle Hale, *Oregon State University*

Tim Harris, *Microsoft*

Niranjan Hasabnis, *Code Metal*

Y. Charlie Hu, *Purdue University*

Liting Hu, *University of California, Santa Cruz*

Yu Hua, *Huazhong University of Science and Technology*

Haochen Huang, *Whova*

Călin Iorgulescu, *Oracle Labs*

Dae R. Jeong, *Georgia Tech*

Jinkyu Jeong, *Yonsei University*

Abhilash Jindal, *Indian Institute of Technology Delhi*

Sang-Woo Jun, *University of California, Irvine*

Myoungsoo Jung, *Korea Advanced Institute of Science and Technology (KAIST)*

Rohan Kadekodi, *University of Washington*

Kostis Kaffes, *Columbia University*

Vasiliki Kalavri, *Boston University*

Ankita Kejriwal, *Snowflake*

Tanvir Ahmed Khan, *Columbia University*

Daehyeok Kim, *The University of Texas at Austin*

Jongyul Kim, *University of Illinois at Urbana–Champaign*

Eddie Kohler, *Harvard University*

Kenji Kono, *Keio University*

Purushottam (Puru) Kulkarni, *Indian Institute of Technology Bombay*

Sándor Laki, *Eötvös Loránd University*

Michael Le, *IBM*

Baptiste Lepers, *Inria*

Alberto Lerner, *University of Fribourg*

Shih-Wei Li, *National Taiwan University*

Dong Li, *University of California, Merced*

Yin Li, *University of Wisconsin–Madison*

Yu Liang, *ETH Zurich*

Peng Liu, *The Pennsylvania State University*

Jean-Pierre Lozi, *Inria*

Haonan Lu, *University at Buffalo*

Lanyue Lu, *Snowflake*

Youyou Lu, *Tsinghua University*

Rajan M A, *Tata Consultancy Services Research and Innovation*

Aravind Machiry, *Purdue University*

Divya Mahajan, *Georgia Institute of Technology*

Paul E. McKenney, *Meta Platforms, Inc.*
Amirsaman Memaripour, *MongoDB*
Alex Merenstein, *IBM*
Michael Mesnier, *Independent*
Subrata Mitra, *Adobe Research*
Apoorve Mohan, *IBM Research*
Sibin Mohan, *The George Washington University*
Sue Moon, *Korea Advanced Institute of Science and Technology (KAIST)*
Shuai Mu, *Stony Brook University*
Dalit Naor, *The Academic College of Tel Aviv-Yaffo*
Khanh Nguyen, *Texas A&M University*
Ruslan Nikolaev, *The Pennsylvania State University*
Sam H. Noh, *Virginia Tech*
Pierre Olivier, *The University of Manchester*
Arthi Padmanabhan, *Harvey Mudd College*
Roberto Palmieri, *Lehigh University*
Abhisek Pan, *Microsoft*
Aurojit Panda, *New York University*
Ashish Panwar, *Microsoft Research India*
Jonggyu Park, *University of Washington*
Fernando Pedone, *Università della Svizzera italiana*
Kevin Pedretti, *Sandia National Laboratories*
Soujanya Ponnapalli, *University of California, Berkeley*
Vijayan Prabhakaran, *Databricks*
Feng Qian, *University of Southern California*
Haoran Qiu, *Microsoft Azure Research*
Jan S. Rellermeyer, *Leibniz University Hannover*
Xiang Ren, *University of Toronto*
João Rodrigues de Campos, *University of Coimbra*
Stephen Rumble, *Google (YouTube)*
Suranga Seneviratne, *The University of Sydney*
Mohammad Shahrad, *University of British Columbia*
Yizhou Shan, *Huawei Cloud*
Jiacheng Shen, *Duke Kunshan University*
Ji-Yong Shin, *Northeastern University*
Liuba Shrira, *Brandeis University*
Manish Shukla, *TCS Research*
Linhai Song, *The Pennsylvania State University*
Sriram Subramanian, *Amazon Web Services*
Nik Sultana, *Illinois Institute of Technology*
Hailong Sun, *Beihang University*
Gongjin Sun, *Samsung Semiconductor, Inc.*
Xuan Sun, *University of Edinburgh*
Steven Swanson, *University of California, San Diego*
Adriana Szekeres, *VMware Research*
Cheng Tan, *Northeastern University*
Alain Tchana, *Grenoble INP*
Gaël Thomas, *Inria Saclay*
Chia-Che Tsai, *Texas A&M University*
Dmitrii Ustiugov, *Nanyang Technological University*
Nilay Vaish, *Google*
Marco Vieira, *University of North Carolina at Charlotte*
Haris Volos, *University of Cyprus*
Zhuang Wang, *Amazon Web Services*
Chenxi Wang, *Institute of Computing Technology, Chinese Academy of Sciences*
Yang Wang, *The Ohio State University and Meta*
Michael Wei, *VMware Research*
Dan Williams, *Virginia Tech*
Youjip Won, *Korea Advanced Institute of Science and Technology (KAIST)*
Timothy Wood, *George Washington University*
Kan Wu, *Google*
Suzhen Wu, *Xiamen University*
Erci Xu, *Shanghai Jiaotong University and Alibaba*
Gang Yan, *University of California, Merced*
Ming-Chang Yang, *The Chinese University of Hong Kong*
Anil Kumar Yelam, *Google*
Eiko Yoneki, *University of Cambridge*
Jiangshan Yu, *The University of Sydney*
Tieying Zhang, *Bytedance*
Yuan Zhang, *Fudan University*
Jie Zhang, *Peking University*
Yongle Zhang, *Purdue University*
Zheng (Eddy) Zhang, *Rutgers University*
Zihao Zhang, *Stony Brook University*
Hao Zhang, *University of California, San Diego*
Minjia Zhang, *University of Illinois at Urbana–Champaign*
Diyu Zhou, *Peking University*

Mentoring Co-Chairs
Vasiliki Kalavri, *Boston University*
Cheng Tan, *Northeastern University*

Artifact Evaluation Committee Co-Chairs
Vijay Chidambaram, *The University of Texas at Austin*
Alain Tchana, *Grenoble INP*

Artifact Evaluation Committee
Ghazal Abdollahi, *University of Utah*
Ghadeer Almusaddar, *Binghamton University*
Ayush Bansal, *University of Illinois Urbana–Champaign*
Binong Kiri Bey, *Indian Insitute of Technology*
Archit Bhatnagar, *University of Michigan*
Rahul Bothra, *University of Illinois Urbana–Champaign*
Cláudia Brito, *INESC TEC and University of Minho*
Quinn Burke, *University of Wisconsin–Madison*
Cathy Cai, *University of Illinois Urbana–Champaign*
Tingjia Cao, *University of Wisconsin–Madison*
Sarthak Chakraborty, *University of Illinois Urbana–Champaign*
Qixuan Chen, *Boston University*
Xiang Chen, *Hong Kong University of Science and Technology*
Yi Chen, *University of Michigan*
Hongzheng Chen, *Cornell University*
Feng Cheng, *Duke University*
Georgia Christofidi, *IMDEA Software Institute*
Jackson Clark, *University of Illinois Urbana–Champaign*
Chuanqi Cong, *Beihang University*
Guangyang Deng, *Xiamen University*
Jiaqi Deng, *Georgia Institute of Technology*
Sriram Devata, *University of Illinois Urbana–Champaign*
Huibing Dong, *University of Minnesota*
Juechu Dong, *University of Michigan*
Chunfeng Du, *Zhengzhou University*

Enguang Fan, *University of Illinois Urbana–Champaign*
Yuan Feng, *University of California, Merced*
Shen Fu, *University of Science and Technology of China*
Kaihui Gao, *Zhongguancun Laboratory*
Xiangyu Gao, *University of Washington*
Raja Gond, *Microsoft Research*
Jinnan Guo, *Imperial College London*
Ke Han, *Purdue University*
Wanning He, *University of Michigan*
Kiran Hombal, *University of Illinois Urbana–Champaign*
Lana Honcharuk, *Crusoe Cloud*
Tony Hong, *University of Illinois Urbana–Champaign*
Guanzhou Hu, *University of Wisconsin–Madison*
Hao Hu, *Harare Institute of Technology*
Tiancheng Hu, *Peking University*
Yibo Huang, *University of Michigan*
Hao Huang, *Harbin Institute of Technology*
Lei Huang, *Boston University*
Ryan Huang, *University of Michigan*
Songlin Huang, *The University of Hong Kong*
Malvika Jadhav, *University of Florida*
Shashwat Jaiswal, *University of Illinois Urbana–Champaign*
Yuchen Ji, *ShanghaiTech University*
Haitian Jiang, *New York University*
Teng "Sebastian" Jiang, *Columbia University*
Zhihan Jiang, *Chinese University of Hong Kong*
Zewen Jin, *University of Science and Technology of China*
Sagar Bharadwaj Kalasibail Seetharam, *Carnegie Mellon University*
Rishika Varma Kalidindi, *University of Michigan*
Saisha Kamat, *University of North Carolina at Charlotte*
Ashish Kashinath, *University of Illinois Urbana–Champaign*
Zhaokang Ke, *University of Minnesota*
Jungwoo Kim, *Stanford University*
Fadhil Kurnia, *University of Massachusetts Amherst*
Hanchen Li, *Stealth Startup and University of California, Berkeley*
Jiansong Li, *Li Auto Inc*
Junzhe Li, *The University of Hong Kong*
Shengkai Li, *University of Chinese Academy of Sciences*
Sijia Li, *Virginia Tech*
Tuo Li, *Tsinghua University*
Yueying Li, *Cornell University*
Yuke Li, *University of California, Merced*
Zeyu Li, *Hong Kong University of Science and Technology*
Zhifei Li, *University of California, Berkeley*
Borui Li, *Southeast University*
Junxuan Liao, *University of Wisconsin–Madison*
Georgios Liargkovas, *Columbia University*
Changyuan Lin, *University of British Columbia*
Jing Liu, *Max Planck Institute for Security and Privacy*
Peizhe Liu, *University of Illinois Urbana–Champaign*
Yiqi Liu, *University of Illinois Urbana–Champaign*
Jiaheng Lu, *University of Michigan*
Shouxi Luo, *Southwest Jiaotong University*
Yilong Luo, *Peking University*
Jeff J. Ma, *University of Michigan*
Shivani Mehendarge, *Amazon*
Deepanjali Mishra, *Carnegie Mellon University*
Farzad Mohammadi, *Imperial College London*
Grigoris Ntousakis, *Brown University*
João Oliveira, *INESC-ID and Instituto Superior Técnico, Universidade de Lisboa*
Nikolaos Pagonas, *Columbia University*
Jiaqi Pan, *University of Illinois Urbana–Champaign and Tsinghua University*
Lichen Pan, *Peking University*
Melissa Z. Pan, *University of California, Berkeley*
Yanqi Pan, *Harbin Institute of Technology*
Zaifeng Pan, *University of California, San Diego*
Santosh Pandey, *Rutgers University*
Konstantinos Papaioannou, *IMDEA Software Institute*
Michael Paper, *Stanford University*
Magdalena Pasternak, *University of Florida*
Sibendu Paul, *Amazon Prime Video*
Pedro Henrique Penna, *Microsoft Research*
Stratos Psomadakis, *National Technical University of Athens*
Rohan Puri, *Samsung Semiconductor*
Shangshu Qian, *Purdue University*
Feiran (Alex) Qin, *North Carolina State University*
Suyan Qu, *University of Wisconsin–Madison*
Vishal Rao, *Georgia Institute of Technology*
Siddhant Ray, *University of Chicago*
Jonah Rosenblum, *University of Michigan*
Giannis Roumpos, *IMDEA Software Institute*
Amit Samanta, *University of Utah*
Srinjoy Sarkar, *Indian Insitute of Technology*
Aditya Shah, *Google*
Shail Shah, *Microsoft*
Shazibul Islam Shamim, *Kennesaw State University*
Tanvi Sharma, *Purdue University*
Xiaoxiang Shi, *Shanghai Jiao Tong University*
Sachin Kumar Singh, *University of Utah*
Jiansen Song, *Chinese Academy of Sciences*
Yuhang Song, *Boston University*
Zihe Song, *The University of Texas at Dallas*
Cesar A. Stuardo, *ByteDance*
Yiming Su, *University of Illinois Urbana–Champaign*
Qingxiao Sun, *China University of Petroleum*
Tong Sun, *Zhejiang University*
Xin Tan, *Chinese University of Hong Kong*
Syed Salauddin Mohammad Tariq, *University of Michigan*
Sahil Tyagi, *Oak Ridge National Laboratory*
Diogo Vaz, *INESC-ID and Instituto Superior Técnico, Universidade de Lisboa*
Alan Wang, *Northeastern University*
Ruihong Wang, *Purdue University*
Wenlong Wang, *University of Minnesota*
Xinkai Wang, *Shanghai Jiao Tong University*
Yun Wang, *Shanghai Jiao Tong University*
Chiyue Wei, *Duke University*
Hengfeng Wei, *Nanjing University*
Caeden Whitaker, *University of Wisconsin–Madison*
Chao Wu, *Nanjing University of Science and Technology*

Yanran Wu, *Purdue University*
Zeqing Wu, *Zhengzhou University*
Ziqiang Wu, *Huazhong University of Science and Technology*
Xingyu Xiang, *Harvard University*
Shaokang Xie, *University of California, Davis*
Yifan Xie, *National University of Defense Technology*
Guanglei Xu, *Huazhong University of Science and Technology*
Hanchen Xu, *Virginia Tech*
Haocheng Xu, *University of California, Irvine*
Tianyin Xu, *University of Illinois Urbana–Champaign*
Kaiwen Xue, *University of Michigan*
Zhijun Yang, *Huazhong University of Science and Technology*
Mingxuan Yang, *Imperial College London*
Sheng Yao, *Hong Kong University of Science and Technology*
Chenhao Ye, *University of Wisconsin–Madison*
Jia Yin, *Renmin University*
Jinsun Yoo, *Georgia Institute of Technology*
Boxi Yu, *Chinese University of Hong Kong*
Diman Zad Tootaghaj, *Hewlett Packard Labs*
Wahid Uz Zaman, *The Pennsylvania State University*
Ioannis Zarkadas, *Columbia University*
Junyao Zhang, *Duke University*
Siyuan Zhang, *Ohio State University*
Songhao Zhang, *University of Illinois Urbana–Champaign*
Wenhui Zhang, *ByteDance*
Xiangqun Zhang, *Syracuse University*
Xiao Zhang, *The University of Texas at Austin*
Yuanliang Zhang, *National University of Defense Technology*
Naiqian Zheng, *Peking University*
Xinying Zheng, *University of Illinois Urbana–Champaign*
Yusheng Zheng, *University of California, Santa Cruz*
Shawn Zhong, *University of Wisconsin–Madison*
Tianle Zhong, *University of Virginia*
Yangjie Zhou, *National University of Singapore*
Siqi Zhu, *Tsinghua University*
Zeying Zhu, *University of Maryland*
Jianian Zhu, *Huazhong University of Science and Technology*
Xiangyu Zou, *Harbin Institute of Technology*

Steering Committee
Irina Calciu, *VMware Research*
Ada Gavrilovska, *Georgia Institute of Technology*
Casey Henderson-Ross, *USENIX Association*
Geoff Kuenning, *Harvey Mudd College*
Brian Noble, *University of Michigan*
Jiri Schindler, *Tranquil Data*
Hakim Weatherspoon, *Cornell University*
Erez Zadok, *Stony Brook University*
Noa Zilberman, *University of Oxford*

External Reviewers

Saksham Agarwal
Muhammad Bilal
Jayson Boubin
Trevor E. Carlson
Haoxian Chen
Mingyu Gao
Jinyu Gu
Dorgival Guedes
Sagar Jha
Gyuyeong Kim
Jaehong Kim
Guyue (Grace) Liu
Yuxin Ren
Robbert van Renesse
Joel Wolfrath
Lei Zhang
Mai Zheng
Jie Zhou

Experts

Sebastian Angel
Ying Chen
Thaleia Dimitra Doudali
Zhihao Jia
Yimin Jiang
Ellie Kitanidis
Tianyu Li
Luo Mai
Ethan Miller
Hyungon Moon
Pavel Panchekha
P. Sadayappan
Amedeo Sapio
Jiwon Seo
Aashaka Shah
Jaewoong Sim
Chunqiang Tang
Guanhong Tao
Vasily Tarasov
Deepak Vasisht
Guanhua Wang
Yunming Xiao
Zhen Xie
Jingling Xue
Kang Yang
Weijie Zhao
Danyang Zhuo

Message from the USENIX ATC '25 Program Co-Chairs

Introduction

We are thrilled and honored to be holding the all-time largest USENIX ATC program, which, unfortunately, will also be ATC's last. The joint USENIX ATC and OSDI conference brings together top academic researchers, industry practitioners, and newcomers in the field; we are looking forward to meeting with you all and to many great discussions during the conference. For many of us in the community, USENIX ATC has been the conference that best captured a balance between academic depth and production pragmatism, and between deep technical advances and creative new ideas. We hope the work in this year's program reflects that legacy.

The rest of this document provides some insights into the submission and selection process that culminated in the 100 accepted works that will be presented at the conference.

Submission Process

As in the recent past, USENIX ATC '25 solicited three types of papers: full-length 12-page research papers, short 6-page research papers that describe complete and well-evaluated ideas using fewer pages, and full-length "Deployed Systems" papers. The Deployed Systems submissions align with the USENIX mission of bringing together researchers in academia and systems practitioners. These papers had different criteria for acceptance from research papers; they were judged more carefully on their practical insights rather than on their new ideas.

In addition to the submission, authors were asked to provide optional supplementary materials, a description of changes that had been made to the submission since prior submissions, and, for Deployed Systems submissions, a justification for why the submission was appropriate for that track.

Program Committee Selection Process

Our program committee (PC) was assembled to meet several competing goals like past USENIX ATC PCs: good coverage across diverse computer-systems topics, balance between academia and industry, a mix of veterans of prior USENIX ATC PCs with individuals in early stages of their professional careers, geographic diversity, and adherence to the USENIX diversity and inclusion principles.

The assembled PC had:

- 172 members from 20 countries.
- Geographic distribution of 67% from North America, 11% from EMEA and 22% from APAC.
- 73% of members from academia and 27% from industry.
- 61% veteran members who served USENIX ATC at least once in the past 3 years.
- 15% female representation, similar to the 14% at USENIX ATC '24.
- And primary areas of expertise from Distributed Systems (19%), Operating Systems (19%), Storage (15%), AI/Machine Learning (14%), Cloud (10%), Security (8%), and Networking (7%).

For the PC selection process, we initially drew from a pool of experienced PC members who had served at least once in the prior 3 iterations of USENIX ATC, removing those who were concurrently serving for OSDI. We prioritized several factors in sending invitations, including: reviewers flagged as producing good reviews for previous conferences; recommendations from invitees who were unable to serve; topic matches to anticipate the need for Machine Learning expertise; and female representation. For the cases in which we needed expertise for a paper where all reviewers identified low levels of expertise, we solicited recommendations from the PC and invited external reviewers.

Review Process

USENIX ATC '25 received 634 submissions across all tracks, which was a 30% increase over USENIX ATC '24 and an 80% increase over USENIX ATC '23. Of these, 32 (5%) were deployed systems papers and 56 (9%) were short papers. The most popular topics for submissions, as specified by authors were: systems and machine learning (37% of submissions), parallel and distributed systems (26%), storage (22%), cloud computing (20%), networking (16%), and operating systems (14%).

We retained USENIX ATC's double-blind review process to minimize bias with strict anonymity rules. Eight papers were

ultimately rejected due to violations of these rules. We rejected six other submissions without review due to violations of the formatting guidelines.

To increase the quality and relevance of reviews, we ran a bidding process where PC members had 3 days to bid on papers within their expertise. USENIX ATC '25 had two double-blind rounds of reviews.

- **Round 1:** The goal was to identify early rejections and papers where reviewers lacked expertise. We assigned 3 reviewers per paper, resulting in 1857 round 1 review assignments. We notified authors of the 345 papers rejected in round 1 (56%) early to give them more time for preparing future submissions of their work.
- **Round 2:** We assigned at least two additional reviewers to the 274 remaining submissions, bringing the total to 5 reviews for each of these papers.
- **Author Response Period:** After the second review period and an online discussion period the reviews were made available for authors to respond to. Authors had 3 days to write a response. The reviewers then continued asynchronous online discussion for another 2 weeks.
- **PC Meeting:** 92 papers were pre-accepted, leaving 27 papers for discussion at the synchronous virtual PC meeting, of which 8 (30%) were accepted. The PC meeting was one day, and it provided a high-bandwidth channel to resolve discussions and select the final program.

The PC selected 100 papers for a 16% acceptance rate. Of these, 6 were deployed systems papers, 7 were short, and 87 were full-length research papers. After selecting the program, the program chairs selected the best paper awards based on nominations from the PC.

Artifact Evaluation Process

USENIX ATC '25 continued to run a joint artifact evaluation process with OSDI '25. The authors of all accepted papers were invited to submit an artifact for an evaluation, and 47 out of the 100 USENIX ATC papers did so.

- 98% of artifacts received an "Available" badge.
- 94% received a "Functional" badge.
- 77% received a "Reproduced" badge.
- 77% of papers received all three badges.

Acknowledgements

Putting on a conference of USENIX ATC's scale is a massive task that requires the help of hundreds of individuals, most of whom are volunteers. First, we thank our Program Committee who bore the spike in submissions, wrote detailed reviews, and argued for papers both online and at the Program Committee Meeting. We also deeply appreciate the Artifact Evaluation chairs and the numerous participants on the Artifact Evaluation Committee that helped to evaluate submissions' software artifacts; this work is crucial for ensuring accountability and reproducibility in our field. Next, as always, the USENIX staff along with the USENIX ATC Steering Committee were instrumental in every step of the process. They were quick to help us when we needed it, and they kept everything organized and moving forward. Finally, we want to express our deep gratitude to the authors, whose incredible work is the bedrock of a great conference; we hope that everyone that submitted to USENIX ATC benefitted from the process.

Thank you to all of the authors and to everyone who has helped make USENIX ATC what it has been over the years—a premier venue where systems research is both principled and grounded in reality. If this is indeed the last USENIX ATC, we hope it's a fitting one.

Deniz Altınbüken, *Google DeepMind*
Ryan Stutsman, *University of Utah*
USENIX ATC '25 Program Co-Chairs

USENIX ATC '25: 2025 USENIX Annual Technical Conference

July 7–9, 2025
Boston, MA, USA

Monday, July 7

Cloud Computing: Speed, Scale, and Serverless

Fast ACS: Low-Latency File-Based Ordered Message Delivery at Scale 1
Sushant Kumar Gupta, Anil Raghunath Iyer, Chang Yu, Neel Bagora, Olivier Pomerleau, Vivek Kumar, and Prunthaban Kanthakumar, *Google LLC*

Poby: SmartNIC-accelerated Image Provisioning for Coldstart in Clouds 19
Zihao Chang and Jiaqi Zhu, *SKLP, Institute of Computing Technology, CAS; University of Chinese Academy of Sciences;* Haifeng Sun, *Peking University;* Yunlong Xie, Kan Shi, Ninghui Sun, Yungang Bao, and Sa Wang, *SKLP, Institute of Computing Technology, CAS; University of Chinese Academy of Sciences*

Burst Computing: Quick, Sudden, Massively Parallel Processing on Serverless Resources 39
Daniel Barcelona-Pons, *Universitat Rovira i Virgili and Barcelona Supercomputing Center*; Aitor Arjona, Pedro García-López, Enrique Molina-Giménez, and Stepan Klymonchuk, *Universitat Rovira i Virgili*

DeepServe: Serverless Large Language Model Serving at Scale 57
Junhao Hu, *Peking University and Key Lab of HCST (PKU), MOE;* Jiang Xu, Zhixia Liu, Yulong He, Yuetao Chen, Hao Xu, Jiang Liu, Jie Meng, Baoquan Zhang, Shining Wan, Gengyuan Dan, Zhiyu Dong, Zhihao Ren, and Changhong Liu, *Huawei Cloud;* Tao Xie, *Key Lab of HCST (PKU), MOE and Peking University;* Dayun Lin, Qin Zhang, Yue Yu, Hao Feng, Xusheng Chen, and Yizhou Shan, *Huawei Cloud*

***Cosmic:* Cost-Effective Support for Cloud-Assisted 3D Printing 73**
Yuan Yao, *University of Southern California;* Chuan He and Chinedum Okwudire, *University of Michigan;* Harsha V. Madhyastha, *University of Southern California*

Accelerating ML Training: Parallelism, Tuning, and Modalities

GMI-DRL: Empowering Multi-GPU DRL with Adaptive-Grained Parallelism 89
Yuke Wang, *Rice University;* Boyuan Feng, *University of California Santa Barbara;* Zheng Wang, *University of California San Diego;* Guyue Huang, *University of California Santa Barbara;* Tong (Tony) Geng, *University of Rochester;* Ang Li, *Pacific Northwest National Laboratory;* Yufei Ding, *University of California San Diego*

***m*Tuner: Accelerating Parameter-Efficient Fine-Tuning on Multi-GPU Servers with Elastic Tensor 105**
Kezhao Huang, Siqi Zhu, Mingshu Zhai, Liyan Zheng, Kinman Lei, Jiaao He, Yuyang Jin, and Jidong Zhai, *Tsinghua University*

Jenga: Enhancing LLM Long-Context Fine-tuning with Contextual Token Sparsity 123
Tuowei Wang and Xingyu Chen, *Tsinghua University;* Kun Li and Ting Cao, *Microsoft Research;* Ju Ren and Yaoxue Zhang, *Tsinghua University*

FlexPipe: Maximizing Training Efficiency for Transformer-based Models with Variable-Length Inputs 143
Hairui Zhao, *Jilin University and University of California, Riverside;* Qi Tian, and Hongliang Li, *Jilin University and Key Laboratory of Symbolic Computation and Knowledge Engineering of the Ministry of Education, China;* Zizhong Chen, *University of California, Riverside*

Optimus: Accelerating Large-Scale Multi-Modal LLM Training by Bubble Exploitation 161
Weiqi Feng, *Harvard University;* Yangrui Chen, *ByteDance;* Shaoyu Wang, *University of Southern California;* Yanghua Peng and Haibin Lin, *ByteDance;* Minlan Yu, *Harvard University*

Networking: From Cloud to In-Network Intelligence

Towards Optimal Rack-scale µs-level CPU Scheduling through In-Network Workload Shaping 179
Xudong Liao, *Hong Kong University of Science and Technology;* Han Tian, *University of Science and Technology of China;* Xinchen Wan, *Hong Kong University of Science and Technology;* Chaoliang Zeng, *BitIntelligence;* Hao Wang, *Hong Kong University of Science and Technology;* Junxue Zhang, *University of Science and Technology of China;* Mengyu Ma, *Inspur;* Guyue (Grace) Liu, *Peking University;* Kai Chen, *Hong Kong University of Science and Technology*

TGW: Operating an Efficient and Resilient Cloud Gateway at Scale 199
Yifan Yang, Lin He, Jiasheng Zhou, Xiaoyi Shi, Yichi Xu, and Shicheng Wang, *Tsinghua University;* Jinlong E, *Renmin University of China;* Ying Liu, *Tsinghua University;* Junwei Zhang, Zhuang Yuan, and Hengyang Xu, *Tencent*

MARC: Motion-Aware Rate Control for Mobile E-commerce Cloud Rendering 217
Yuankang Zhao, *Alibaba Group; University of Chinese Academy of Sciences;* Furong Yang, *Alibaba Group;* Gerui Lv, *University of Chinese Academy of Sciences;* Qinghua Wu, *University of Chinese Academy of Sciences and Purple Mountain Laboratories;* Yanmei Liu, Jiuhai Zhang, Yutang Peng, Feng Peng, Hongyu Guo, and Ying Chen, *Alibaba Group;* Zhenyu Li, *University of Chinese Academy of Sciences and Purple Mountain Laboratories;* Gaogang Xie, *University of Chinese Academy of Sciences and Computer Network Information Center, Chinese Academy of Sciences*

Accelerating Distributed Graph Learning by Using Collaborative In-Network Multicast and Aggregation 233
Zhaoyi Li, *Central South University and Nanyang Technological University;* Jiawei Huang, Yijun Li, and Jingling Liu, *Central South University;* Junxue Zhang, *Hong Kong University of Science and Technology;* Hui Li, Xiaojun Zhu, Shengwen Zhou, Jing Shao, Xiaojuan Lu, Qichen Su, and Jianxin Wang, *Central South University;* Chee Wei Tan, *Nanyang Technological University;* Yong Cui, *Tsinghua University;* Kai Chen, *Hong Kong University of Science and Technology*

Opening Up Kernel-Bypass TCP Stacks 249
Shinichi Awamoto and Michio Honda, *University of Edinburgh*

Operating Systems: Scheduling, Security, and Extensibility

GPreempt: GPU Preemptive Scheduling Made General and Efficient. 263
Ruwen Fan and Tingxu Ren, *Tsinghua University;* Minhui Xie, *Renmin University of China;* Shiwei Gao, Jiwu Shu, and Youyou Lu, *Tsinghua University*

μEFI: A Microkernel-Style UEFI with Isolation and Transparency 273
Le Chen, Yiyang Wu, Jinyu Gu, Yubin Xia, and Haibo Chen, *Shanghai Jiao Tong University*

PageFlex: Flexible and Efficient User-space Delegation of Linux Paging Policies with eBPF 291
Anil Yelam and Kan Wu, *Google;* Zhiyuan Guo, *UC San Diego;* Suli Yang, *Google;* Rajath Shashidhara, *University of Washington;* Wei Xu and Stanko Novaković, *Google;* Alex C. Snoeren, *Google and UC San Diego;* Kimberly Keeton, *Google*

Asterinas: A Linux ABI-Compatible, Rust-Based Framekernel OS with a Small and Sound TCB 307
Yuke Peng, *SUSTech;* Hongliang Tian, *Ant Group;* Junyang Zhang and Ruihan Li, *Peking University and Zhongguancun Laboratory;* Chengjun Chen and Jianfeng Jiang, *Ant Group;* Jinyi Xian, *SUSTech;* Xiaolin Wang, Chenren Xu, Diyu Zhou, and Yingwei Luo, *Peking University and Zhongguancun Laboratory;* Shoumeng Yan, *Ant Group;* Yinqian Zhang, *SUSTech*

Rex: Closing the language-verifier gap with safe and usable kernel extensions. 325
Jinghao Jia and Ruowen Qin, *University of Illinois Urbana-Champaign;* Milo Craun and Egor Lukiyanov, *Virginia Tech;* Ayush Bansal and Minh Phan, *University of Illinois Urbana-Champaign;* Michael V. Le, Hubertus Franke, and Hani Jamjoom, *IBM;* Tianyin Xu, *University of Illinois Urbana-Champaign;* Dan Williams, *Virginia Tech*

The Programmable Data Plane: SmartNICs and Beyond

Barre: Empowering Simplified and Versatile Programmable Congestion Control in High-Speed AI Clusters. 343
Yajuan Peng, *Shanghai Key Laboratory for Intelligence Information Processing, Fudan University, China;* Haoran Wei, Xiaolong Zhong, Junkai Huang, Haohan Xu, Zicheng Wang, Yang Bai, Zhuo Jiang, and Jianxi Ye, *ByteDance;* Xiaoliang Wang; Xiaoming Fu, *Shanghai Key Laboratory for Intelligence Information Processing, Fudan University, China;* Huichen Dai, *ByteDance*

FLB: Fine-grained Load Balancing for Lossless Datacenter Networks 365
Jinbin Hu, *Central South University, Hong Kong University of Science and Technology, Changsha University of Science and Technology;* Wenxue Li, Xiangzhou Liu, Junfeng Wang, and Bowen Liu, *Hong Kong University of Science and Technology;* Ping Yin, *Inspur;* Jianxin Wang and Jiawei Huang, *Central South University;* Kai Chen, *Hong Kong University of Science and Technology*

Snary: A High-Performance and Generic SmartNIC-accelerated Retrieval System 381
Qiaoyin Gan, *Institute of Computing Technology, Chinese Academy of Sciences;* Heng Pan, *Computer Network Information Center, Chinese Academy of Sciences;* Luyang Li, Kai Lv, and Hongtao Guan, *Institute of Computing Technology, Chinese Academy of Sciences;* Zhaohua Wang, *Computer Network Information Center, Chinese Academy of Sciences;* Zhenyu Li, *Institute of Computing Technology, Chinese Academy of Sciences;* Gaogang Xie, *Computer Network Information Center, Chinese Academy of Sciences*

Minos : A Lightweight and Dynamic Defense against Traffic Analysis in Programmable Data Planes. 399
Zihao Wang, *Pengcheng Laboratory and Tsinghua Shenzhen International Graduate School;* Qing Li, Guorui Xie, Dan Zhao, Kejun Li, and Zhuochen Fan, *Pengcheng Laboratory;* Lianbo Ma, *Northeastern University;* Yong Jiang, *Pengcheng Laboratory and Tsinghua Shenzhen International Graduate School*

Performance: Benchmarking, Caching, and Workload Characterization

GeneralSparse: Bridging the Gap in SpMM for Pruned Large Language Model Inference on GPUs 417
Yaoyu Wang, Xiao Guo, Junmin Xiao, De Chen, and Guangming Tan, *SKLP, Institute of Computing Technology, CAS; and University of Chinese Academy of Sciences*

HyCache: Hybrid Caching for Accelerating DNN Input Preprocessing Pipelines. 433
Keshav Vinayak Jha, *Independent Researcher;* Shweta Pandey, *Indian Institute of Science;* Murali Annavaram, *University of Southern California;* Arkaprava Basu, *Indian Institute of Science*

The Koala Benchmarks for the Shell: Characterization and Implications. 449
Evangelos Lamprou and Ethan Williams, *Brown University;* Georgios Kaoukis, *National Technical University of Athens;* Zhuoxuan Zhang, *Brown University;* Michael Greenberg, *Stevens Institute of Technology;* Konstantinos Kallas, *University of California, Los Angeles;* Lukas Lazarek and Nikos Vasilakis, *Brown University*

KVCache Cache in the Wild: Characterizing and Optimizing KVCache Cache at a Large Cloud Provider 465
Jiahao Wang, Jinbo Han, and Xingda Wei, *Shanghai Jiao Tong University;* Sijie Shen, *Alibaba Group;* Dingyan Zhang, *Shanghai Jiao Tong University;* Chenguang Fang, *Alibaba Group;* Rong Chen, *Shanghai Jiao Tong University;* Wenyuan Yu, *Alibaba Group;* Haibo Chen, *Shanghai Jiao Tong University*

Tuesday, July 8

Storage Innovations: Logs, Tiers, and Modern Flash

LogCrisp: Fast Aggregated Analysis on Large-scale Compressed Logs by Enabling Two-Phase Pattern Extraction and Vectorized Queries 483
Junyu Wei, Guangyan Zhang, and Junchao Chen, *Tsinghua University;* Qi Zhou, *Alibaba Cloud*

HotRAP: Hot Record Retention and Promotion for LSM-trees with Tiered Storage 497
Jiansheng Qiu, Fangzhou Yuan, *Tsinghua University*; Mingyu Gao, and Huanchen Zhang, *Tsinghua University and Shanghai Qi Zhi Institute*

Mitigating Resource Usage Dependency in Sorting-based KV Stores on Hybrid Storage Devices via Operation Decoupling. 513
Qingyang Zhang and Yongkun Li, *University of Science and Technology of China;* Yubiao Pan, *Huaqiao University;* Haoting Tang, *University of Science and Technology of China*; Yinlong Xu, *University of Science and Technology of China, and Anhui Provincial Key Laboratory of High Performance Computing*

SolFS: An Operation-Log Versioning File System for Hash-free Efficient Mobile Cloud Backup 531
Riwei Pan, *Department of Computer Science, City University of Hong Kong;* Yu Liang, *ETH Zurich;* Lei Li and Hongchao Du, *Department of Computer Science, City University of Hong Kong;* Tei-Wei Kuo, *Delta Electronics and National Taiwan University;* Chun Jason Xue, *Mohamed bin Zayed University of Artificial Intelligence*

Z-LFS: A Zoned Namespace-tailored Log-structured File System for Commodity Small-zone ZNS SSDs. 547
Inhwi Hwang, *Seoul National University;* Sangjin Lee, *Chung-Ang University;* Sunggon Kim, *Seoul National University of Science and Technology;* Hyeonsang Eom, *Seoul National University;* Yongseok Son, *Chung-Ang University*

Serving Intelligence: Efficient LLM Inference at the Edge and Cloud

CLONE: Customizing LLMs for Efficient Latency-Aware Inference at the Edge 563
Chunlin Tian, Xinpeng Qin, Kahou Tam, Li Li, Zijian Wang, Yuanzhe Zhao, Minglei Zhang, and Chengzhong Xu, *University of Macau*

WEAVER: Efficient Multi-LLM Serving with Attention Offloading 587
Shiwei Gao, Qing Wang, Shaoxun Zeng, Youyou Lu, and Jiwu Shu, *Tsinghua University*

Torpor: GPU-Enabled Serverless Computing for Low-Latency, Resource-Efficient Inference 597
Minchen Yu, *The Chinese University of Hong Kong, Shenzhen; and Hong Kong University of Science and Technology;* Ao Wang, *Alibaba Group;* Dong Chen, Haoxuan Yu, Xiaonan Luo, Zhuohao Li, and Wei Wang, *Hong Kong University of Science and Technology;* Ruichuan Chen, *Nokia Bell Labs;* Dapeng Nie, Haoran Yang, and Yu Ding, *Alibaba Group*

Toppings: CPU-Assisted, Rank-Aware Adapter Serving for LLM Inference 613
Suyi Li, Hanfeng Lu, and Tianyuan Wu, *Hong Kong University of Science and Technology;* Minchen Yu, *The Chinese University of Hong Kong, Shenzhen;* Qizhen Weng, *TeleAI, China Telecom;* Xusheng Chen and Yizhou Shan, *Huawei Cloud;* Binhang Yuan and Wei Wang, *Hong Kong University of Science and Technology*

QFactory: Accelerating Quantized Large Language Model Serving with Qtile Graphs 631
Qihao Zhang, Mingshu Zhai, Rui Sun, and Jidong Zhai, *Tsinghua University*

Optimizing ML Execution: Compilers, Pipelines, and Runtimes

PluS: Highly Efficient and Expandable ML Compiler with Pluggable Graph Schedules. 647
Ruofan Wu, *Renmin University of China;* Zhen Zheng, *Microsoft;* Feng Zhang, *Renmin University of China;* Chuanjie Liu, *Microsoft;* Zaifeng Pan, *Renmin University of China;* Jidong Zhai, *Tsinghua University;* Xiaoyong Du, *Renmin University of China*

Obscura: Concealing Recomputation Overhead in Training of Large Language Models with Bubble-filling Pipeline Transformation 665
Yuzhou Huang and Yapeng Jiang, *Sun Yat-sen University;* Zicong Hong, *Hong Kong University of Science and Technology;* Wuhui Chen, *Sun Yat-sen University;* Bin Wang and Weixi Zhu, *Huawei Technologies;* Yue Yu, *Peng Cheng Laboratory;* Zibin Zheng, *Sun Yat-sen University*

PPipe: Efficient Video Analytics Serving on Heterogeneous GPU Clusters via Pool-Based Pipeline Parallelism 679
Z. Jonny Kong, Qiang Xu, and Y. Charlie Hu, *Purdue University*

Voltrix: Sparse Matrix-Matrix Multiplication on Tensor Cores with Asynchronous and Balanced Kernel Optimization 699
Yaqi Xia and Weihu Wang, *Wuhan University;* Donglin Yang, *Nvidia Corporation;* Xiaobo Zhou, *University of Macau;* Dazhao Cheng, *Wuhan University*

Resilient Systems: Failure Detection, Consistency, and Scalability

***NetKeeper*: Enhancing Network Resilience with Autonomous Network Configuration Update on Traffic Patterns and Anomalies 715**
Zhaoyang Wan, Rongxin Han, Haifeng Sun, Qi Qi, Zirui Zhuang, and Bo He, *State Key Laboratory of Networking and Switching Technology, Beijing University of Posts and Telecommunications;* Liang Zhang, *Huawei Technologies Co., Ltd;* Jianxin Liao and Jingyu Wang, *State Key Laboratory of Networking and Switching Technology, Beijing University of Posts and Telecommunications*

Greyhound: Hunting Fail-Slows in Hybrid-Parallel Training at Scale. 731
Tianyuan Wu and Wei Wang, *Hong Kong University of Science and Technology;* Yinghao Yu, Siran Yang, and Wenchao Wu, *Alibaba Group;* Qinkai Duan, *Hong Kong University of Science and Technology;* Guodong Yang, Jiamang Wang, Lin Qu, and Liping Zhang, *Alibaba Group*

Crash Consistency in Block-Level Caching Systems: An Open CAS Case Study 749
Shaohua Duan, *Washington State University;* Youmin Chen, *Shanghai Jiao Tong University*

FiDe: Reliable and Fast Crash Failure Detection to Boost Datacenter Coordination 765
Davide Rovelli, *Università della Svizzera Italiana and SAP SE;* Pavel Chuprikov, *Télécom Paris and Institut Polytechnique de Paris;* Philipp Berdesinski, *turba;* Ali Pahlevan, *SAP SE;* Patrick Jahnke, *turba;* Patrick Eugster, *Università della Svizzera Italiana*

Network Performance & Protocols: From Space to VR

LEOCraft: Towards Designing Performant LEO Networks 789
Suvam Basak and Amitangshu Pal, *Indian Institute of Technology Kanpur;* Debopam Bhattacherjee, *Microsoft Research India*

Emulating Space Computing Networks with RHONE **815**
Liying Wang, *Peking University;* Qing Li, *Beijing University of Posts and Telecommunications;* Yuhan Zhou and Zhaofeng Luo, *Peking University;* Donghao Zhang and Shangguang Wang, *Beijing University of Posts and Telecommunications;* Xuanzhe Liu and Chenren Xu, *Peking University; and Key Laboratory of High Confidence Software Technologies, Ministry of Education (PKU)*

Roaming Free in the VR World with MP² **833**
Yifei Xu, *University of California, Los Angeles;* Xumiao Zhang, *University of Michigan and Alibaba Cloud;* Yuning Chen, *University of California, Merced;* Pan Hu, *Uber Technologies, Inc.;* Xuan Zeng, Zhilong Zheng, Xianshang Lin, and Yanmei Liu, *Alibaba Cloud;* Songwu Lu, *University of California, Los Angeles;* Z. Morley Mao, *University of Michigan;* Wan Du, *University of California, Merced;* Dennis Cai and Ennan Zhai, *Alibaba Cloud;* Yunfei Ma, *Uber Technologies, Inc.*

STORM: a Multipath QUIC Scheduler for Quick Streaming Media Transport under Unstable Mobile Networks **851**
Liekun Hu and Changlong Li, *East China Normal University, Jianghuai Advance Technology Center, and MoE Engineering Research Center of Hardware/Software Co-Design Technology and Application*

Internet Connection Splitting: What's Old is New Again **867**
Gina Yuan, Thea Rossman, and Keith Winstein, *Stanford University*

Distributed Systems: Communication, Consensus, and Data Structures

WIC: Hiding Producer-Consumer Synchronization Delays with Warp-Level Interrupt-based GPU Communications **889**
Jiajian Zhang, *Xi'an Jiaotong-Liverpool University and University of Liverpool;* Fangyu Wu, *Xi'an Jiaotong-Liverpool University;* Hai Jiang, *Beijing University of Posts and Telecommunications;* Qiufeng Wang, *Xi'an Jiaotong-Liverpool University;* Genlang Chen and Chaoyi Pang, *NingboTech University*

Primus: Unified Training System for Large-Scale Deep Learning Recommendation Models **905**
Jixi Shan, *ByteDance Inc.;* Xiuqi Huang, *Zhejiang University;* Yang Guo, Hongyue Mao, Ho-Pang Hsu, Hang Cheng, Can Wang, and Jun Song, *ByteDance Inc.;* Rui Shi, *Bytedance Inc.;* Xiaofeng Gao, *Shanghai Jiao Tong University;* Jingwei Xu, Shiru Ren, Jiaxiao Zheng, Hua Huang, Lele Yu, and Peng Xu, *ByteDance Inc.;* Guihai Chen, *Shanghai Jiao Tong University*

Chitu: Avoiding Unnecessary Fallback in Byzantine Consensus **923**
Rongji Huang, Xiangzhe Wang, Xiaofeng Yan, and Lei Fan, *Shanghai Jiao Tong University;* Guangtao Xue and Shengyun Liu, *Shanghai Jiao Tong University and Shanghai Key Laboratory of Trusted Data Circulation, Governance and Web3*

Fast Distributed Transactions for RDMA-based Disaggregated Memory **943**
Haodi Lu, Haikun Liu, Yujian Zhang, Zhuohui Duan, Xiaofei Liao, Hai Jin, and Yu Zhang, *Huazhong University of Science and Technology*

Cuckoo for Clients: Disaggregated Cuckoo Hashing **959**
Stewart Grant and Alex C. Snoeren, *UC San Diego*

Virtualization and Isolation: Security, Sharing, and Performance

LITESHIELD: Secure Containers via Lightweight, Composable Userspace µKernel Services **973**
Kaesi Manakkal, *The University of Texas at Arlington;* Nathan Daughety and Marcus Pendleton, *Air Force Research Laboratory (AFRL);* Hui Lu, *The University of Texas at Arlington*

Accelerating Nested Virtualization with HyperTurtle **987**
Ori Ben Zur and Jakob Krebs, *Technion - Israel Institute of Technology;* Shai Aviram Bergman, *Huawei Zurich Research Center;* Mark Silberstein, *Technion - Israel Institute of Technology*

Efficient Performance-Aware GPU Sharing with Compatibility and Isolation through Kernel Space Interception **1003**
Shulai Zhang, Ao Xu, Quan Chen, Han Zhao, and Weihao Cui, *Shanghai Jiao Tong University;* Zhen Wang, Yan Li, and Limin Xiao, *Lenovo;* Minyi Guo, *Shanghai Jiao Tong University*

AnchorNet: Bridging Live and Collaborative Streaming with a Unified Architecture 1021
Tong Meng, Wei Zhang, Dong Chen, Zhen Wang, Quanqing Li, Changqing Yan, Wei Yang, Chao Yuan, Le Zhang, Jianxin Kuang, and Jianlin Xu, *ByteDance*

Scaling Complex Models: Distribution, Heterogeneity, and Efficiency

KATZ: Efficient Workflow Serving for Diffusion Models with Many Adapters 1037
Suyi Li, Lingyun Yang, Xiaoxiao Jiang, Hanfeng Lu, and Dakai An, *Hong Kong University of Science and Technology;* Zhipeng Di, Weiyi Lu, Jiawei Chen, Kan Liu, Yinghao Yu, Tao Lan, Guodong Yang, Lin Qu, and Liping Zhang, *Alibaba Group;* Wei Wang, *Hong Kong University of Science and Technology*

PopFetcher: Towards Accelerated Mixture-of-Experts Training Via Popularity Based Expert-Wise Prefetch 1053
Junyi Zhang, Chuanhu Ma, Xiong Wang, and Yuntao Nie, *Huazhong University of Science and Technology;* Yuqing Li, *Wuhan University;* Yuedong Xu, *Fudan University;* Xiaofei Liao, *Huazhong University of Science and Technology;* Bo Li, *Hong Kong University of Science and Technology;* Hai Jin, *Huazhong University of Science and Technology*

HYPERECA: Distributed Heterogeneous In-Memory Embedding Database for Training Recommender Models 1071
Jiaao He, Shengqi Chen, Kezhao Huang, and Jidong Zhai, *Tsinghua University*

CrossPipe: Towards Optimal Pipeline Schedules for Cross-Datacenter Training 1089
Tiancheng Chen, Ales Kubicek, Langwen Huang, and Torsten Hoefler, *ETH Zurich*

Wednesday, July 9

Hunting Elusive Bugs: Verification and Analysis from Compilers to Hardware

Unveiling Compiler Faults via Attribute-Guided Compilation Space Exploration 1109
Jiangchang Wu, Yibiao Yang, Maolin Sun, and Yuming Zhou, *State Key Laboratory for Novel Software Technology, Nanjing University*

Understanding and Detecting Fail-Slow Hardware Failure Bugs in Cloud Systems 1127
Gen Dong and Yu Hua, *Huazhong University of Science and Technology;* Yongle Zhang, *Purdue University;* Zhangyu Chen and Menglei Chen, *Huazhong University of Science and Technology*

CONVEROS: Practical Model Checking for Verifying Rust OS Kernel Concurrency 1143
Ruize Tang, *State Key Laboratory for Novel Software Technology, Nanjing University;* Minghua Wang, *Ant Group;* Xudong Sun, *University of Illinois Urbana-Champaign;* Lin Huang, *Ant Group;* Yu Huang and Xiaoxing Ma, *State Key Laboratory for Novel Software Technology, Nanjing University*

BIN2WRONG: a Unified Fuzzing Framework for Uncovering Semantic Errors in Binary-to-C Decompilers 1161
Zao Yang and Stefan Nagy, *University of Utah*

HEC: Equivalence Verification Checking for Code Transformation via Equality Saturation 1181
Jiaqi Yin and Zhan Song, *University of Maryland, College Park;* Nicolas Bohm Agostini and Antonino Tumeo, *Pacific Northwest National Laboratory;* Cunxi Yu, *University of Maryland, College Park*

Software-Hardware Synergy: Accelerators, Memory, and Interconnects

Para-ksm: Parallelized Memory Deduplication with Data Streaming Accelerator 1197
Houxiang Ji, *University of Illinois Urbana-Champaign;* Minho Kim and Seonmu Oh, *Daegu Gyeongbuk Institute of Science and Technology;* Daehoon Kim, *Yonsei University;* Nam Sung Kim, *University of Illinois Urbana-Champaign*

DSA-2LM: A CPU-Free Tiered Memory Architecture with Intel DSA 1213
Ruili Liu, *Tsinghua University and University of Electronic Science and Technology of China;* Teng Ma, *Alibaba Group;* Mingxing Zhang, Jialiang Huang, and Yingdi Shan, *Tsinghua University;* Zheng Liu, *Alibaba Group;* Lingfeng Xiang, Zhen Lin, Hui Lu, and Jia Rao, *The University of Texas at Arlington;* Kang Chen and Yongwei Wu, *Tsinghua University*

Turbocharge ANNS on Real Processing-in-Memory by Enabling Fine-Grained Per-PIM-Core Scheduling 1223
Puqing Wu, Minhui Xie, Enrui Zhao, Dafang Zhang, and Jing Wang, *Renmin University of China;* Xiao Liang and Kai Ren, *Kuaishou;* Yunpeng Chai, *Renmin University of China*

SwCC: Software-Programmable and Per-Packet Congestion Control in RDMA Engine 1243
Hongjing Huang, Jie Zhang, Xuzheng Chen, Ziyu Song, Jiajun Qin, and Zeke Wang, *Zhejiang University*

DRack: A CXL-Disaggregated Rack Architecture to Boost Inter-Rack Communication 1261
Xu Zhang and Ke Liu, *SKLP, Institute of Computing Technology, CAS; and University of Chinese Academy of Sciences;* Yuan Hui and Xiaolong Zheng, *Huawei;* Yisong Chang, *SKLP, Institute of Computing Technology, CAS; and University of Chinese Academy of Sciences;* Yizhou Shan, *Huawei Cloud;* Guanghui Zhang, *Shandong University;* Ke Zhang, Yungang Bao, Mingyu Chen, and Chenxi Wang, *SKLP, Institute of Computing Technology, CAS; and University of Chinese Academy of Sciences*

Securing the Stack: Attestation, Memory Protection, and Privacy

ShieldReduce: Fine-Grained Shielded Data Reduction 1281
Jingyuan Yang, Jun Wu, Ruilin Wu, and Jingwei Li, *University of Electronic Science and Technology of China;* Patrick P. C. Lee, *The Chinese University of Hong Kong;* Xiong Li and Xiaosong Zhang, *University of Electronic Science and Technology of China*

MemoryTrap: Booby Trapping Memory to Counter Memory Disclosure Attacks with Hardware Support 1297
Chenke Luo, *Wuhan University and Tulane University;* Jiang Ming, *Tulane University;* Dongpeng Xu, *University of New Hampshire;* Guojun Peng and Jianming Fu, *Wuhan University*

Separate but Together: Integrating Remote Attestation into TLS 1319
Carsten Weinhold, *Barkhausen Institut;* Muhammad Usama Sardar, *TU Dresden;* Ionut Mihalcea and Yogesh Deshpande, *Arm;* Hannes Tschofenig, *University of Applied Sciences Bonn-Rhein-Sieg;* Yaron Sheffer, *Intuit;* Thomas Fossati, *Linaro;* Michael Roitzsch, *Barkhausen Institut*

DDLumos: Understanding and Detecting Atomic DDL Bugs in DBMSs 1327
Zhiyong Wu, *Tsinghua University;* Jie Liang, *Beihang University;* Jingzhou Fu, Wenqian Deng, and Yu Jiang, *Tsinghua University*

Hardware-Specific Optimizations: Space, Accelerators, and AI Chips

***SpaceExit*: Enabling Efficient Adaptive Computing in Space with Early Exits 1343**
Jiacheng Liu, *Shanghai Jiao Tong University and The Chinese University of Hong Kong;* Xiaozhi Zhu, Tongqiao Xu, Xiaofeng Hou, and Chao Li, *Shanghai Jiao Tong University*

XRT: An Accelerator-Aware Runtime for Accelerated Chip Multiprocessors 1359
Neel Patel and Mohammad Alian, *Cornell University*

DShuffle: DPU-Optimized Shuffle Framework for Large-scale Data Processing 1371
Chen Ding, Sicen Li, and Kai Lu, *Wuhan National Laboratory for Optoelectronics, Huazhong University of Science and Technology;* Ting Yao, Daohui Wang, and Huatao Wu, *Huawei Cloud;* Jiguang Wan, Zhihu Tan, Changsheng Xie, *Wuhan National Laboratory for Optoelectronics, Huazhong University of Science and Technology*

Accelerating Model Training on Ascend Chips: An Industrial System for Profiling, Analysis and Optimization 1387
Yuhang Zhou, Zibo Wang, Zhibin Wang, Ruyi Zhang, Chen Tian, Xiaoliang Wang, Wanchun Dou, and Guihai Chen, *Nanjing University;* Bingqiang Wang, Yonghong Tian, Yan Zhang, and Hui Wang, *Peng Cheng Laboratory;* Fuchun Wei, Boquan Sun, Jingyi Zhang, Bin She, Teng Su, Yifan Yao, Chunsheng Li, Ziyang Zhang, and Yaoyuan Wang, *Huawei;* Bin Zhou, *Shandong University;* Guyue Liu, *Peking University*

Finding Faults: Concurrency, Numerics, and Kernel Fuzzing

CAFault: Enhance Fault Injection Technique in Practical Distributed Systems via Abundant Fault-Dependent Configurations. 1409
Yuanliang Chen, Fuchen Ma, Yuanhang Zhou, Zhen Yan, and Yu Jiang, *Tsinghua University*

Revealing Floating-Point Accumulation Orders in Software/Hardware Implementations 1425
Peichen Xie, Yanjie Gao, Yang Wang, and Jilong Xue, *Microsoft Research*

Inferring Likely Counting-related Atomicity Program Properties for Persistent Memory 1441
Yunmo Zhang and Junqiao Qiu, *City University of Hong Kong;* Hong Xu, *The Chinese University of Hong Kong;* Chun Jason Xue, *MBZUAI*

Optimizing Input Minimization in Kernel Fuzzing 1451
Hui Guo, *East China Normal University;* Hao Sun, *ETH Zurich;* Shan Huang, Ting Su, and Geguang Pu, *East China Normal University;* Shaohua Li, *The Chinese University of Hong Kong*

IRHash: Efficient Multi-Language Compiler Caching by IR-Level Hashing 1467
Tobias Landsberg and Johannes Grunenberg, *Leibniz Universität Hannover;* Christian Dietrich, *Technische Universität Braunschweig;* Daniel Lohmann, *Leibniz Universität Hannover*

Advanced Distributed Systems: ML, Data, and Performance Tuning

On-Demand Container Partitioning for Distributed ML 1481
Giovanni Bartolomeo, Navidreza Asadi, Wolfgang Kellerer, and Jorg Ott, *Technical University of Munich;* Nitinder Mohan, *TU Delft*

PathWeaver: A High-Throughput Multi-GPU System for Graph-Based Approximate Nearest Neighbor Search 1501
Sukjin Kim, Seongyeon Park, Si Ung Noh, Junguk Hong, Taehee Kwon, Hunseong Lim, and Jinho Lee, *Seoul National University*

Universal Checkpointing: A Flexible and Efficient Distributed Checkpointing System for Large-Scale DNN Training with Reconfigurable Parallelism 1519
Xinyu Lian, *University of Illinois Urbana–Champaign;* Sam Ade Jacobs, Lev Kurilenko, and Masahiro Tanaka, *Microsoft;* Stas Bekman, *Snowflake;* Olatunji Ruwase, *Microsoft;* Minjia Zhang, *University of Illinois Urbana–Champaign*

Towards High-Performance Transactional Stateful Serverless Workflows with Affinity-Aware Leasing 1535
Jianjun Zhao, Haikun Liu, Shuhao Zhang, and Haodi Lu, *Huazhong University of Science and Technology;* Yancan Mao, *School of Computing, National University of Singapore;* Zhuohui Duan, Xiaofei Liao, and Hai Jin, *Huazhong University of Science and Technology*

Swift: Fast Performance Tuning with GAN-Generated Configurations. 1553
Chao Chen and Shixin Huang, *Shenzhen Institutes of Advanced Technology, Chinese Academy of Sciences;* Xuehai Qian, *Tsinghua University;* Zhibin Yu, *Shenzhen Institutes of Advanced Technology, Chinese Academy of Sciences; and Shuhai Lab, Huawei Cloud*

OS, Mobile, and Reliability Challenges

PMR: Fast Application Response via Parallel Memory Reclaim on Mobile Devices 1569
Wentong Li, *MoE Engineering Research Center of Software/Hardware Co-Design Technology and Application, East China Normal University, Shanghai, China; and School of Computer Science and Technology, East China Normal University;* Li-Pin Chang, *National Yang Ming Chiao Tung University, Taiwan;* Yu Mao, *City University of Hong Kong, Hong Kong; and MBZUAI, Abu Dhabi;* Liang Shi, *MoE Engineering Research Center of Software/Hardware Co-Design Technology and Application, East China Normal University, Shanghai, China; and School of Computer Science and Technology, East China Normal University*

SAVE: Software-Implemented Fault Tolerance for Model Inference against GPU Memory Bit Flips 1585
Wenxin Zheng, Bin Xu, Jinyu Gu, and Haibo Chen, *Shanghai Jiao Tong University*

Identifying and Analyzing Pitfalls in GNN Systems 1605
Yidong Gong, Arnab Kanti Tarafder, Saima Afrin, and Pradeep Kumar, *William & Mary*

Bluetooth Low Energy Security Testing with Combinatorial Methods 1625
Dominik-Philip Schreiber, Manuel Leithner, and Jovan Zivanovic, *SBA Research;* Dimitris E. Simos, *SBA Research, Salzburg University of Applied Sciences, and Paris Lodron University of Salzburg*

Intelligent Resource Management: Federation, Colocation, and ML for Systems

Resource Multiplexing in Tuning and Serving Large Language Models 1639
Yongjun He and Haofeng Yang, *ETH Zurich;* Yao Lu, *National University of Singapore;* Ana Klimovic and Gustavo Alonso, *ETH Zurich*

Colocating ML Inference and Training with Fast GPU Memory Handover 1657
Jiali Wang, Yankui Wang, Mingcong Han, and Rong Chen, *Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University*

AssyLLM: Efficient Federated Fine-tuning of LLMs via Assembling Pre-trained Blocks 1677
Shichen Zhan, Li Li, and Chengzhong Xu, *University of Macau*

Learning-Enhanced High-Throughput Pattern Matching Based on Programmable Data Plane 1693
Guanglin Duan and Yucheng Huang, *Peng Cheng Laboratory, Tsinghua Shenzhen International Graduate School;* Zhengxin Zhang, *Peng Cheng Laboratory, Tsinghua Shenzhen International Graduate School, and Cornell University;* Qing Li and Dan Zhao, *Peng Cheng Laboratory;* Zili Meng, *Hong Kong University of Science and Technology;* Dirk Kutscher, *Hong Kong University of Science and Technology* (*Guangzhou*)*;* Ruoyu Li, *Shenzhen University and Peng Cheng Laboratory;* Yong Jiang, *Tsinghua Shenzhen International Graduate School;* Mingwei Xu, *Tsinghua University*

Fast ACS: Low-Latency File-Based Ordered Message Delivery at Scale

Sushant Kumar Gupta *Anil Raghunath Iyer* *Chang Yu* *Neel Bagora*
Olivier Pomerleau *Vivek Kumar* *Prunthaban Kanthakumar*

Google LLC

fast-acs-wp@google.com

Abstract

Low-latency message delivery is crucial for real-time systems. Data originating from a producer must be delivered to consumers, potentially distributed in clusters across metropolitan and continental boundaries. With the growing scale of computing, there can be several thousand consumers of the data. Such systems require a robust messaging system capable of transmitting messages containing data across clusters and efficiently delivering them to consumers. The system must offer guarantees like ordering and at-least-once delivery while avoiding overload on consumers, allowing them to consume messages at their own pace.

This paper presents the design of Fast ACS (an abbreviation for Ads Copy Service), a file-based ordered message delivery system that leverages a combination of two-sided (inter-cluster) and one-sided (intra-cluster) communication primitives—namely, Remote Procedure Call and Remote Memory Access, respectively—to deliver messages. The system has been successfully deployed to dozens of production clusters and scales to accommodate several thousand consumers within each cluster, which amounts to Tbps-scale intra-cluster consumer traffic at peak. Notably, Fast ACS delivers messages to consumers across the globe within a few seconds or even sub-seconds (p99) based on the message volume and consumer scale, at a low resource cost.

1 Introduction

Low-latency message delivery is crucial for real-time systems such as advertising, dynamic pricing, retail inventory management, fraud detection, activity monitoring, online matching, and online gaming. The explosion of data is driving a corresponding growth in the size and complexity of real-time systems. These systems are often geographically distributed across numerous clusters. This distribution is essential for handling the immense volume of requests and maintaining a low-latency response. Furthermore, these systems are designed to scale horizontally, which is vital because they often handle internet-scale traffic with user numbers that can fluctuate dramatically. This also makes them vulnerable to internet-scale threats like DDoS attacks.

Consider the example of Google Ads, a large-scale online advertising system. The Ads serving system is geographically distributed across multiple clusters to handle massive user traffic. Each cluster runs numerous jobs with multiple replicas to ensure high availability and low response time. Some jobs may have thousands of replicas per cluster, while others may have tens of thousands, resulting in hundreds of thousands of replicas globally. Another example of a large-scale real-time system is online shopping serving. They need to be highly scalable to accommodate peak shopping periods when demand can increase dramatically. This requirement for dynamic scalability is common to many real-time systems that serve internet-scale traffic.

Such real-time systems rely on real-time data pipelines to receive updates for their states. These pipelines utilize real-time extractors [5, 6, 11, 56] that depend on a robust message delivery subsystem. This subsystem transports messages containing extracted data to the real-time systems. Depending on the specific needs of the real-time system, the message delivery subsystem must offer certain guarantees. For example, stateful real-time systems that can handle duplicate messages require *in-order* sequencing (messages delivered in the order they were produced) and *at-least-once* delivery (each message delivered at least once to all consumers). Ordered delivery encompasses unordered delivery and at-least-once delivery subsumes at-most-once delivery. While exactly-once delivery is desirable too, it requires two-phase commit, which is challenging for cross-cluster delivery and hence beyond the scope of this paper. Beyond delivery guarantees, low latency is paramount for these messaging systems, as is the ability to handle a massive number of consumers.

Existing Message Delivery System Message delivery systems are available in two variants: *push-based* and *pull-based*. Pioneered by systems like Scribe [42], Siena [12], and Gryphon [49], push-based asynchronous message delivery has found widespread adoption in RabbitMQ [19],

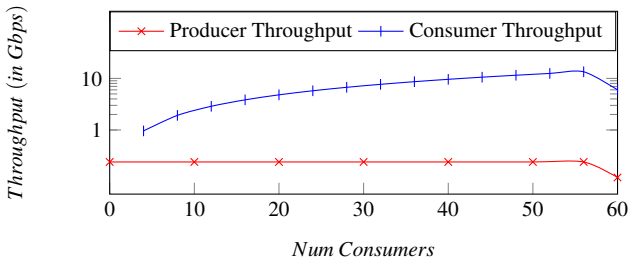


Figure 1: Results for Kafka consumer-scaling experiment. As consumers scaled, the consumer throughput went up to 14 Gbps, after which the performance degraded.

Facebook's Wormhole [44], Google Cloud PubSub [34], and Apache ActiveMQ [14]. However, relying on a push-based model presents challenges for many systems. When a diverse set of consumers, with their varying latency-sensitive backends, needs to be supported, push-based delivery becomes impractical. Handling push messages can block critical CPU cores, leading many systems to prefer pulling messages at their own pace. Moreover, pull-based messaging systems can deliver higher throughput that push-based systems.

Systems such as Apache Kafka [33] have implemented pull-based models, employing brokers to store messages in files on their local file system. Each broker handles one or more *partitions* of a *topic*. Within a partition, messages are totally ordered, while across partitions, they are partially ordered. The throughput of brokers, for both writing and reading, is constrained by the capabilities of the storage medium (like SSDs, currently maxing out at about 4.84 Gbps). Kafka performs well when message volume is high (in the Gbps range) and consumer count is limited. However, it is not optimized for scenarios with a large fan-out; tens of thousands of consumers can lead to potential meltdown, at the very least cause a significant increase in latency. There are ongoing efforts to replace local file system with Hadoop Distributed File System (HDFS) [47]. However, low-latency delivery to several consumers requires enormous tail-reading of file bytes, and so HDFS data node servers with a 64 MB file block size can still face throughput issues. In-memory caching of "hot" messages within brokers offers a potential workaround for storage bottlenecks. However, hot-spotting and network congestion can occur even within a single partition, for instance, when global total order is required.

To demonstrate consumer throughput issues, we conducted an experiment on Google Cloud [15]. The experiment involved a Kafka cluster with the following specifications: 3 brokers in a zone, each configured with 8 vCPUs, 32 GB of RAM, and NVMe SSD; and one topic with 9 non-replicated partitions.The partitioning ensured even distribution of load across brokers, preventing hot-spotting. A producer was initiated, sending 1 KB-sized messages at a rate of 240 Mbps. Then, consumers were scaled up incrementally, adding four at each step. Figure 1 shows the results. Initially, consumer throughput scaled linearly, reaching a peak of 14 Gbps. However, beyond this point, we observed a drop in the write rate due to throughput conflict between writes and reads. The producer byte rate decreased to 120 Mbps, consequently causing a drop in overall consumer throughput. The maximum consumer throughput achieved in this Kafka setup was 14 Gbps, after which performance degraded.

Apache Pulsar [32] is a geo-replicated message delivery system. It uses Apache BookKeeper [21] for storage. BookKeeper persists messages in distributed write-ahead logs, and indexes them in a cache for fast retrieval. These logs are segmented to optimize performance. However, their large segment size (typically 128 MB) can lead to throughput issues when several consumers tail-read from them.

Our Contributions To the best of our knowledge, there are no messaging systems that can handle large consumer fan-out and provide low-latency ordered message delivery. Our contribution lies in the development of such a system.

In summary, our system utilizes files on a distributed file system, namely Colossus [27] (formerly Google File System [22]). Producers append messages to these files, while consumers poll and read them. We introduce an in-memory layer on top of Colossus specifically designed for low-latency message delivery at scale. The system leverages one-sided network primitives, namely Remote Memory Access (RMA), for intra-cluster reads, while utilizing traditional two-sided communication, namely Remote Procedure Call (RPC), for inter-cluster byte transfer. The system provides essential guarantees for message delivery: *in-order* sequencing and *at-least-once* delivery to all consumers. Additionally, it enables consumers to pull messages at their own pace.

Our contributions are fourfold: 1) We demonstrate how RMA-based in-memory caching overcomes tail-read throughput limitations, enabling low-latency message delivery. 2) We show that proper chunking and distribution eliminate hot-spotting, even with global total ordering constraints. 3) We demonstrate that scaling out resources horizontally effectively mitigates network limitations. 4) We highlight the resource efficiencies achieved by leveraging RMA.

Our contribution targets real-time systems with high consumer throughput requirements, rather than systems that need extremely low latency, say, under 50ms. While achieving such low latency is possible with enhanced resources, it can be expensive. The ideas discussed in this paper can be adapted to enhance the consumer scalability of existing systems, leading to improvements in their benchmark performance.

We have materialized the design to build Fast ACS (an abbreviation for Ads Copy Service). Fast ACS has been deployed in production, consistently delivering messages to thousands of consumers across dozens of production clusters within a few seconds or even sub-seconds (p99). Fast ACS is adopted by systems that demand ordering, at-least-once delivery and low-latency on a large scale.

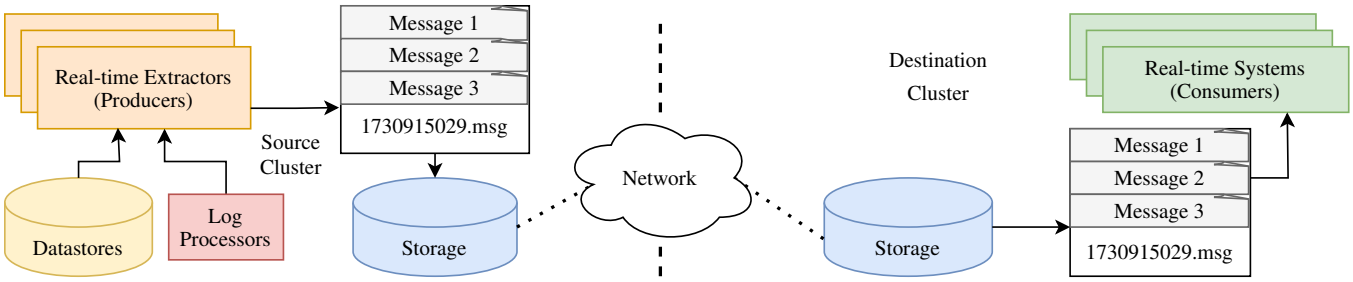


Figure 2: File-based ordered messaging for a real-time systems. Each message stream can have multiple destination clusters based on its consumers. Message files are tail-copied from the source cluster's storage to the destination clusters' storage.

2 Design Overview

We will begin by outlining how our system achieves ordered message delivery through files. Starting from Section 2.2, we will detail our system's core components.

2.1 File-Based Ordered Message Delivery

Files provide a mechanism for ordered message delivery. Producers append messages to *message files* and consumers read from them. To limit their size, the message files undergo periodic rollover. Each message file includes its creation timestamp in its name, enabling chronological sorting.

This file-based messaging approach is agnostic to the underlying message formats, focusing primarily on the storage and delivery of bytes. Consequently, producers and consumers can define their own protocols for serializing and deserializing messages to and from the files.

Producers write messages into distinct *message streams* in the source cluster, each representing an individual bundle of messages, analogous to *topics* in pub-sub systems. Producers can write to multiple message streams, while consumers can read from multiple ones as well. Each message stream is sharded for scalability, with each shard comprising a set of chronological message files containing the messages. A message stream shard resides within a single directory of the file system, simplifying organization. Thus, all ordered messages for a single message stream shard can be found in chronological files under one directory.

To accommodate geographically distributed consumers, message files must be copied from the source cluster's storage to the destination cluster's storage. Consumers in the destination cluster can then tail-read the file from their cluster-local storage system. Figure 2 provides an illustration of our cross-cluster file-based ordered message delivery for a real-time system.

2.2 Multi-Layer Storage

As illustrated in Figure 3, the message files are stored in a multi-layered storage system: Colossus [27], a distributed file system, serves as the primary layer, while a replicated in-memory cache acts as the secondary layer. This additional

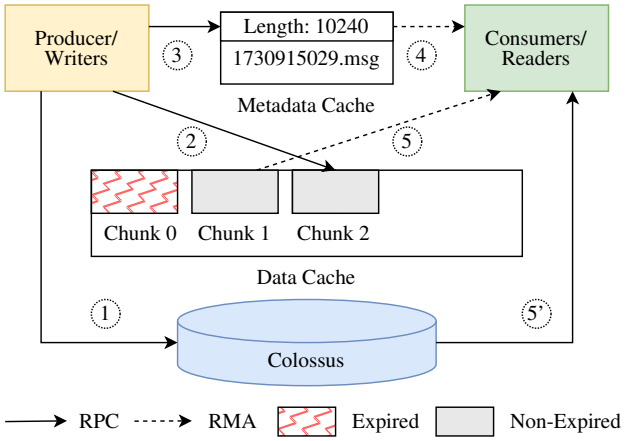


Figure 3: The multi-layer storage design. The producers/writers write the bytes to the Colossus file (1) and shadow the bytes as chunks to the data cache (2), and then update the length of the file in the metadata cache (3). The consumers/readers poll the metadata cache for the latest length of the file (4) and read the chunks from the data cache preferably (5), with Colossus acting as a fallback (5'). All the reads from caches happen over RMA. In this illustration, chunk 0 has expired while chunk 2 is only partially-filled. The length stored in the metadata cache points to the current position within this last, partially-filled chunk.

layer ports minimal Colossus features and operates in parallel to it. All message stream bytes are replicated to both layers. Given its finite capacity, the in-memory cache layer only stores the most recently written, "hot" bytes for the files. The layer absorbs the majority of queries for tail-reads, with Colossus serving as a fallback. This offers a low-latency, high-throughput path for message delivery.

The In-Memory Cache Layer Files are segmented into a series of fixed-size chunks, each 4 KB in size. These chunks are then represented as key-value pairs. The key for each chunk is a hash of two components - the absolute Colossus file path, which is globally unique across Colossus, and the chunk's sequence number. The value comprises the corresponding chunk bytes. The 4 KB chunk size is partly based on the observed message size and partly on the maximum transmission unit in the cluster fabric, allowing a chunk to be read in a single packet.

We employed CliqueMap [4] as the underlying system for the in-memory storage layer. CliqueMap is a remote in-memory key-value cache that facilitates writes through RPC and intra-cluster reads via RMA. The latter proves advantageous given the read-heavy nature of our intra-cluster consumer traffic. Each key-value pair in the cache is subject to garbage collection (GC) upon either the expiration of its Time-To-Live (TTL) or when the cache approaches its storage capacity limit. CliqueMap employs consistent hashing [31] for key distribution among replicas. Within a replica, CliqueMap divides its memory into fixed-size *slabs*, and key hashes are divided into *buckets* via set-associative mapping.

CliqueMap supports r=3.2 replication, which provides resilience against single-point failures. With replication, CliqueMap offers a range of read consistency modes, two of which we leverage: the *consistent* mode, which reads values from at least 2 out of 3 replicas, thereby delivering the most recent value, and the *relaxed* mode, which reads from a random replica that may provide a stale value.

In constructing a file system atop CliqueMap, we employ two instances per cluster: a *data cache* and a *metadata cache*. These are analogous to the Colossus *chunkservers* and *metadata servers* respectively, except lacking processing capabilities. As a result, unlike Colossus, the data and metadata for a file cannot be atomically written to the cache layer. The data cache acts as the repository for file chunks while the metadata cache stores file metadata, particularly (a) the current *length* of the file and (b) the *file lock*. The current length, a monotonically increasing number, reflects the file length of the append-only file, stored as chunks, in the data cache, while the file lock serves to prevent concurrent writers in a best-effort manner. As elaborated subsequently, locks primarily contribute to performance optimization and do not impact the system's correctness. While both file chunks and metadata stored in such volatile caches remains vulnerable to data loss, our system is designed to withstand such occurrences, as detailed in the subsequent sections.

The data cache has two types of chunks for each file: *complete* chunks and a *last, partially filled* chunk (if any). Each chunk in the data cache possesses the following property: reading any part of a chunk will always yield the correct bytes, if present, for the corresponding file positions, even if the chunk is partially filled. This is guaranteed by always writing a chunk starting from its beginning and never from any other position, as detailed later. TTL expiration automatically removes older, non-hot chunks.

2.3 Routing

Directly streaming bytes from the source cluster to all the destination clusters of a message stream would be prohibitively expensive. To address this, we utilize a Prim-based [41] optimizer to construct a minimum spanning tree (MST), hereafter referred to as a *copy tree*, for each message stream, as illustrated in Figure 4. While prioritizing bandwidth cost, the optimizer is also fine-tuned to minimize both, the tree's depth and the fan-out per node. Minimizing the tree's depth is essential, as each hop along the copy tree introduces some delay. The first hop from source typically traverses continental boundaries, followed by a hop across metro boundaries. A typical copy tree has a maximum of 4 hops from the source cluster which can accommodate a few hundred clusters.

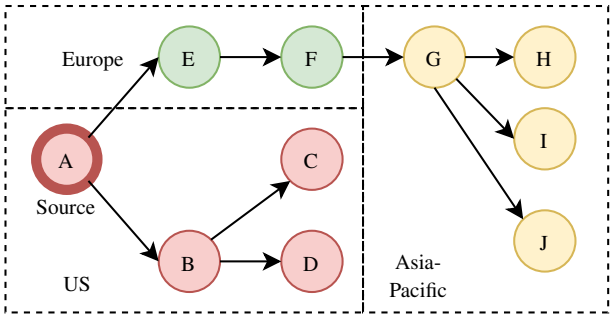


Figure 4: An MST for a message stream. The source cluster is in the US, and the stream is copied to three clusters within the US, two clusters in Europe, and four clusters in the Asia-Pacific region.

Outages in a cluster, often caused by factors such as bad releases, connectivity problems, or bulk maintenance events, can delay all descendant clusters in the copy tree. Such disruptions frequently led to significant Service-Level Objective (SLO) misses, which prompted us to automate the outage handling process. We employ a breadth-first search algorithm to penalize the problematic nodes and links, and then delegate reconstruction of the copy tree to the optimizer. The reconstruction usually sets the problematic nodes as leaf nodes in the copy tree. Consumers in the affected cluster are re-routed to a nearby healthy cluster, sacrificing the ability to perform RMA reads. Similarly, when a new cluster is added to a message stream, it starts as a leaf node with rate-limited delivery and is then promoted to a regular node once the catch-up to the head is complete.

Along each hop of the copy tree, we have a pair of worker jobs - *readers* and *writers*. Readers are responsible for polling, reading, and sending bytes from the current hop to the next hops of the copy tree while writers write the bytes to the destination storage. This design provides the necessary decoupling for the jobs to scale independently. The readers scale based on the egress bandwidth demands, which depends on the fan-out from the current cluster, and the writers scale based on the ingress bandwidth demands.

Each file is tail-copied independently through an *operation*, identified by the Colossus target file path and the storage type - either Colossus or cache. As a result, for each file, the operations for Colossus and cache are independent along the entire copy tree, which also doubles the bandwidth utilization. While pairing them was technically feasible, it led to additional complexity in implementation and introduced avoidable latency. For example, invoking the Colossus *Open* file func-

tion in the write mode is prohibitively expensive as it needs to acquire lock on the file. The cache operation bypasses such costly Colossus operations and is heavily-optimized for parallelization, enabling the fastest possible byte delivery rate.

Finally, operations are scheduled by a cluster-local *scheduler*, which is triggered by a notification from the producer. The schedulers establish a continuous stream along the tree.

2.4 Components

Producers Producers (i.e., real-time extractors) are responsible for generating update files and appending serialized messages to them. Our *File* interface implementation writes the serialized messages to the underlying Colossus file and then asynchronously shadows (copies) the bytes to the data cache. The length of the Colossus file and the length of the file in the data cache are then recorded in the metadata cache. This allows for low-latency retrieval of the file lengths from the metadata cache via RMA, especially for Colossus, where Colossus Bigtable [13] often runs into hot-spotting issues when several consumers poll it simultaneously.

Flushing messages to Colossus establishes a single source of truth, with all other copies (in other Colossus clusters and caches) reflecting this committed state. To amortize disk write costs, producers typically buffer messages before flushing.

In order to shadow the bytes to the data cache, the file implementation tracks the last partially-filled chunk (if any) for the append-only file. When new bytes are flushed to Colossus, the file implementation computes the chunks for the data cache and performs a bulk-write operation on the corresponding key-value pairs. Since a write to a chunk is atomic, slow-growing files often end up overwriting the same bytes to the same chunk multiple times, thereby increasing intra-cluster bandwidth usage unnecessarily. This can be partially resolved by reducing the chunk size to fit the file's byte rate.

Readers For each operation, a reader replica in an upstream cluster initiates a long-lived streaming RPC connection [24] with a writer replica in a downstream cluster. The operation polls the file systems for file length changes and subsequently reads and sends the bytes downstream.

File length is periodically polled from the metadata cache, which stores lengths for both the data cache file and the Colossus file. To reduce overhead, a single background thread performs consistent bulk reads from the metadata cache on behalf of all the operations scheduled on the reader replica. The polling interval for the metadata cache is set to 50ms, and can be reduced to 10ms or less based on latency requirements. Additionally, the Colossus file length is also polled from Colossus Bigtable, albeit less frequently. This serves as a fallback, in case the metadata cache becomes unavailable.

The data cache reads employ a two-step process. Initially, a relaxed read for chunks is performed, followed by consistent reads for those chunks that return fewer bytes than expected — a scenario that may occur when reading from a lagging data cache replica, as illustrated in Figure 5. Empirical observations show that consistent reads are infrequent. Hence, relaxed reads contribute to saving substantial intra-cluster bandwidth and processing resources. An alternative approach could involve using *2xR*, where a quorum on the version number of the key-value pair is first established. However, this necessitates two round-trips, which introduces latency. The relaxed read is hedged after a short delay of 30ms.

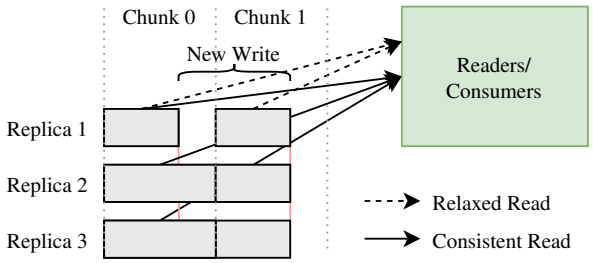


Figure 5: An illustration of relaxed reads followed by consistent reads. Suppose new bytes are written across multiple chunks, filling up the previously partially-filled chunk 0. The new chunk 1 is successfully written to all replicas, but the updated chunk 0 is only written to replicas 2 and 3. When a reader detects a new length from the metadata cache and performs a relaxed read for chunk 0 from replica 1, it will receive fewer bytes than expected. The reader will then perform a consistent read to obtain the complete chunk 0.

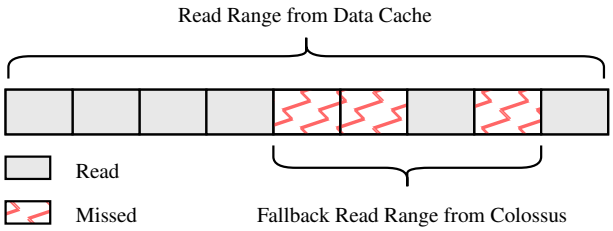


Figure 6: An illustration of fallback reads to Colossus in the event of misses from the data cache. All missed chunks are fetched in a single Colossus read to amortize the RPC cost.

Note that because the data cache is volatile, it is possible to encounter a missing chunk even after a consistent read. A missing chunk is retrieved from Colossus by reading from the positions corresponding to the chunk. In scenarios where multiple chunks are missing, the corresponding read from Colossus spans the combined positions of all the missing chunks as illustrated in Figure 6.

For a Colossus operation, the bytes are retrieved opportunistically from the data cache first and then from Colossus upon a miss, as described above. Opportunistic reads from the data cache in Colossus operations are best-effort but save valuable disk time. The Colossus Flash Cache [53] is ineffective as it only caches immutable blocks, which is not useful for tail-reading growing files. The read bytes are relayed to the downstream cluster sequentially, in line with Colossus's append-only operation support.

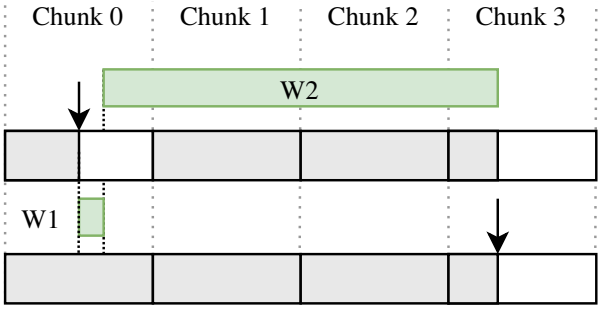


Figure 7: An illustration of writes to data cache. Consider a scenario where the 0th chunk is partially-filled and there are two concurrent write requests: W_1 with a few bytes (not enough to fill the 0th chunk) and W_2 with large bytes (spanning over the 0th, 1st and 2nd chunk fully, and the 3rd chunk partially). It is possible for W_2 to arrive before W_1, leading to the 1st, 2nd chunk and 3rd chunk being written before the 0th chunk. Consequently, the W_2th write for the 0th chunk needs to be serialized after W_1 is received and the chunk is complete. The length, shown as pointer, is updated only after all bytes up to the pointer are filled.

Similarly, for a cache operation, chunks are fetched from the data cache first, followed by a fallback to Colossus upon a miss. However, unlike a Colossus operation, the bytes are read and sent to the writer in parallel, i.e., the processing of the next file length delta does not wait for the previous one to complete.

Astute readers may notice a circular dependency between Colossus and cache operations. However, this circularity is resolved by the fact that actual byte reads are limited to the length of the file in the corresponding storage only. The cache reads by Colossus operations are *opportunity-based* while the Colossus reads by the cache operations are *fallback-based*.

Writers For a Colossus file, the operation employs single-threaded appends, which also updates the metadata in Colossus Bigtable, constituting an atomic step.

For the data cache, the operation differentiates between complete chunks and the last partially-filled chunk. Complete chunks are written in parallel as they will only be written once. On the other hand, writes to partially filled chunks are serialized to ensure that the prefix of bytes in the chunk is always correct, as illustrated in Figure 7. This creates per-chunk dependencies between write requests and the operation maintains a map of such write relationships for each chunk. Once all writes are complete, the length in the metadata cache is then updated to reflect the position up to which all complete chunks and the last partially-filled chunk have been written to the data cache. As noted earlier, unlike a Colossus file, the writes to the data and the metadata cache do not constitute an atomic step.

Note that this out-of-order parallel-write scheme for the data cache interferes with evictions based on TTL where chunks with higher sequence number may be evicted before chunks with lower sequence number. In practice, this has not been a significant problem because consumers rarely fall behind enough to be impacted by chunk expiration.

Algorithm 1 Delayed Colossus Reads for a File

```
1:  maxDelay ← 1s                          ▷ Maximum tolerable read delay.
2:  cfsSizeRec ← queue()                   ▷ Colossus file size historian.
3:
4:  procedure ShouldReadFromColossus
5:      cacheSize ← cache.size()
6:      cfsSize ← colossus.size()
7:      cfsSizeRec.push({time: time(), size: cfsSize})
8:
9:      while !cfsSizeRec.empty() do
10:         if cfsSizeRec.front().size > cacheSize then
11:             break
12:         cfsSizeRec.pop()               ▷ Remove caught-up positions.
13:     if cfsSizeRec.empty() then
14:         return false                   ▷ Cache is caught-up.
15:     if time() > cfsSizeRec.front().time + maxDelay then
16:         cfsSizeRec.clear()
17:         return true                    ▷ Cache is delayed.
18:     return false                       ▷ Cache may catch-up soon.
```

Consumers Consumers (i.e., real-time systems) are responsible for reading bytes from the storage and deserializing the messages in a sequential manner. Our design guarantees eventual progress for the consumers, regardless of the volatile state of either the metadata or data caches. This is due to the persistent, append-only nature of Colossus files.

Like readers, consumers actively poll the metadata cache to get the file length for both the Colossus file and the data cache file. Consumers then have the option to read from either of the two storage layers. A naive approach would be to read the chunks from the storage which is ahead of the other in order to minimize latency. Typically, the data cache file leads the Colossus file, making this strategy suitable for most cases. However, we have observed that Colossus files can often overtake the data cache intermittently, particularly when a data cache stream is interrupted along some copy tree hop. This used to result in a significant reads from Colossus, leading to increased latency penalties when Colossus throttled the requests. To address this, our objective was to prioritize reading from the data cache as long as latency requirements are met. This is achieved by postponing reads from Colossus unless message delivery latency is at risk. The implementation for delayed Colossus reads is illustrated in Algorithm 1. Our *File* interface implementations makes this process transparent to the consumer jobs.

Sometimes, some consumers might start late or fall behind. In such cases, past reads from the data cache will fail as the chunks would have expired, and the consumers are forced to fallback to Colossus until they catch up to the head and can resume reading from the cache. Upon cache miss, the fallback

mechanism is same as that described for readers.

Lastly, each consumer passively rate-limits the reads to cap the read bandwidth in the event that the producer restarts and writes a large backlog to the cache. Without the rate limit, the read amplification would overwhelm the data cache.

3 Implementation Details

Start Position for Cache Copies In essence, the data cache is designed to serve only non-lagging consumers. Consumers who have fallen behind are expected to catch up using Colossus file before reading chunks from the data cache. This simplification allows us to help keep the data cache stream ahead of the Colossus stream. In cases of cache outages and subsequent recovery, the data cache stream jumps to the chunk where the Colossus file is currently positioned, and the length in the metadata cache is updated accordingly.

Dueling Writers The scheduler employs Slicer [2] for sharding, which, like many sharding schemes, provides weak consistency guarantees. While strong consistency could have been achieved by employing a lock service like Chubby [10] or ZooKeeper [28], or by utilizing Slicer's strong consistency mode, these approaches compromise availability. Prioritizing availability, we opted for Slicer's weak consistency model. Consequently, multiple operations can exist for the same target path and storage.

For Colossus files, multiple writers are prohibited since the *Open* operation on a file invalidates all existing file handles used by other processes. This allows a new operation to automatically supersede an older one for a target path.

However, such a mechanism is not available for cache operations as the cache storage layer cannot execute logic. This can result in multiple writers competing, causing clobbering of key-value pairs in both the data and the metadata caches. For instance, complete chunks would be overwritten with partial chunks or the file length would intermittently go backwards. This would negatively impact performance, forcing consumers to frequently fallback to Colossus. Note that the correctness remains unaffected, and appendix A provides a formal specification verifying the system's correctness.

To address the performance issue, we utilize locks stored in the metadata cache. These locks are essentially leases that can be marked as *poisoned*. Each lock is a key-value pair, where the key is derived from a hash of the Colossus file path, and the value is a unique signature for each operation. This signature consists of a unique identifier for the writer replica and a random nonce to distinguish the operation on that replica. A lock for a file is obtained at the beginning of an operation and released when the operation completes. After acquisition, the writer continuously checks the lock for poisoning. When another operation for the same file is scheduled, it *poisons* the existing lock. Upon detecting the poison, the existing operation is then forced to terminate, allowing the new operation to acquire the lock. Additionally, if a writer terminates without releasing the lock, the new operation seizes the lock after a small delay.

Tuning the Caches We tuned both data and metadata caches for low-latency. These caches operate within the standard shared Borg [52] environment, where various jobs compete for resources. Network bandwidth is the most critical resource for the caches due to the high fan out within each cluster. Both data and metadata caches scale horizontally [43] based on the bandwidth and QPS usage.

I. Data Cache: A 1-minute TTL is assigned to the chunks, accommodating most, if not all, consumers. GC is configured to start once the cache reaches 80% of its capacity. The initial RAM for each replica and the minimum number of replica per cluster is configured to accommodate a minute worth of most recent chunks. The slabs and buckets for key-value pairs are pre-allocated to avoid delays caused by resizing. Chaining is enabled to accommodate overflowing buckets to prevent undesired forced evictions. Furthermore, the data cache utilizes a least-recently modified policy, prioritizing the removal of the oldest written chunk.

II. Metadata Cache: The TTL is set to 24 hours which is much larger than the usual rollover time for update files. Since the number of key-value pairs in metadata cache is inherently restricted by the number of files, most capacity and GC settings were deemed unnecessary.

CPU usage on the caches is typically minimal, as most traffic is read-heavy that utilizes a highly efficient software RMA implementation running on Pony Express [38].

To ensure low latency, clients set strict sub-second deadlines on cache operations. Additionally, we tuned the CliqueMap client-side executor for high parallelism by increasing both the number of threads and the buffer space allocated per thread. The underlying execution of operations can be parallelized because we manage concurrency at the application layer. However, as a side-effect, this also increased RAM usage on the consumer side. We then performed manual tuning to optimize for a low resource footprint without impacting latency. In the future, we wish to automate this based on the read traffic pattern.

Finally, to avoid potential server-side resource conflicts with write operations, reads are deliberately configured to never resort to using RPC mode, except for overflowing bucket reads. However, this still does not prevent OS network buffers and bandwidth conflicts between reads and writes.

Fault Tolerance The caches can handle single point failures, though multiple failures may impact performance. Both readers and writers can tolerate multiple failures, but this can create temporary imbalance due to operation rescheduling, potentially slowing down operations and increasing latency. To mitigate this, new operations are throttled and existing operation are periodically shed to restore balance.

The scheduler manages the entire life cycle of an operation.

During transient periods of scheduler unavailability, such as restart, an operation can continue running in an *orphaned* state thereby maintaining stream continuity. Once the scheduler is back online, it can schedule a new operation for the same file and storage, which will force the existing orphaned operation to terminate, as detailed previously. This straightforward design, inspired by the *make-before-break* strategy, ensures that the stream flow remains uninterrupted.

When scheduler and workers are both unavailable, delays are inevitable. A stream might be disrupted and will need to be re-established.

4 Evaluation

In our evaluation, we aim to address several key questions:

I. Ideal Performance: How does the system perform under ideal conditions?

II. Abrupt Load Management: What happens to the system when an abrupt consumer load is exerted on it?

III. Fault Tolerance: How does the system handle faults and ensure resilience?

IV. Scalability: How effectively does the storage layer scale in response to egress bandwidth demands?

V. Backlog Recovery: How does the system recover from large message backlogs?

All evaluations were performed within the shared Borg [52] environment. Each cache replica was allocated 1 CPU core and 8 GB RAM. Consumers were assigned 1 CPU core and 4 GB RAM. Readers and writers were assigned 2 CPU cores and 4 GB RAM, and were configured to scale both vertically and horizontally based on the demand. All jobs were configured to utilize high-priority CPU resources and ran on machines with Pony Express [38], providing highly-efficient software RMA. We used the B4 network [23, 30] for WAN traffic, which is designed for cheaper, low-priority, non-user-facing traffic.

Producers generated messages in data-index format, appending messages into two files: the *data* file, containing message contents, and the *index* file, which primarily stored the offsets of the message contents within the data file. Additionally, producers had a buffering interval of 100ms.

For evaluation purposes, monitoring jobs were turned up alongside consumers. These jobs mimicked consumers and exported performance metrics. The key metric exported by the monitors was *message delivery delay*, calculated as the difference between the producer's time on message production and the consumer's time on message consumption. The delay was comprehensive as it included the time taken by the producer to serialize and write the message, the time taken by the transport layer to copy the message to the destination, and the time taken by the consumer to read and deserialize the message. Although Google's TrueTime [16] could have facilitated cross-cluster measurements, we opted to use smeared UTC [45] instead. This decision was based on the observation

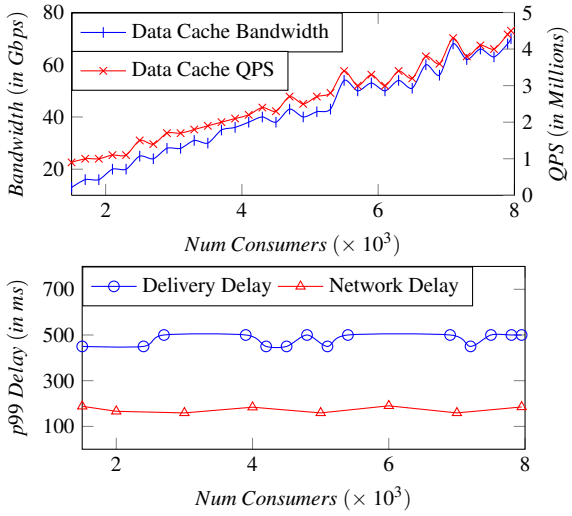


Figure 8: Results for experiment 1(a). The fluctuations in data cache bandwidth and QPS are attributed to repeated reads for last partially-filled chunk and the execution of consistent reads when relaxed reads fail. The message delivery delay exported by monitors remained relatively stable regardless of the number of consumers.

that clock drifts were negligible compared to the actual delays we encountered. All reported delays have a relative error margin of less than 7%. Metrics were exported to Monarch [1] a few minutes into steady-state after startup to avoid capturing noise generated during the initial read up to head.

4.1 Experiment 1

In this experiment, we fixed the number of cache replicas. We turned up 9 replicas for the data cache and 6 replicas for the metadata cache in each cluster. To generate workload, we enabled two message streams originating from clusters in the us-central and euro-west regions. These streams were then replicated to 15 destination clusters. The combined producer write rate was 240 Mbps on average. Using r=3.2 and chunk size of 4 KB, this configuration yielded a data cache write QPS of approximately 22,500 in each cluster. The message streams were 120-way sharded, and each shard contained their own data-index file pairs. Each consumer was assigned to read from 4 shards, resulting in a read rate of 8 Mbps per consumer. The polling interval was set to 50ms for readers and 100ms for consumers.

Experiment 1(a) - Ideal Case We started with 1,500 consumers in one of the leaf clusters of the copy tree, which was 3 hops away from each source, located in asia-east region. We scaled-up the number of consumers to see how it affects the latency. The consumers were scaled up smoothly adding 5 replicas every 5 seconds. Throughout the experiment, all upstream clusters reported stable delays. This represents a fairly ideal scenario.

Observations: The experiments concluded with 7,950 ac-

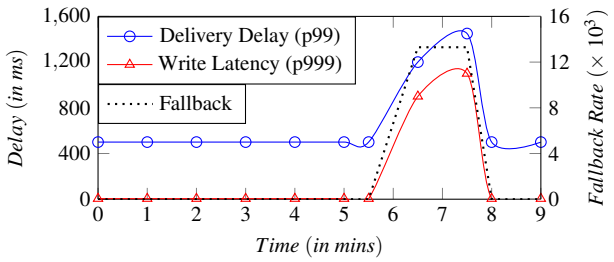


Figure 9: Results for experiment 1(b). The elevated cache write latency led to a substantial increase in fallbacks to Colossus by consumers, consequently impacting delivery delay.

tive consumers. Figure 8 shows the results. Notably, no fallback to Colossus was observed during stable state of consumers. Peak read bandwidth from the data cache reached 70 Gbps, exceeding the anticipated 62 Gbps by approximately 13%. A similar trend was noted for QPS, although the deviation from expectations was more pronounced. The peak QPS to the data cache was 4.5M, surpassing the expected 2M by 2.25x. The elevated bandwidth can be attributed to two factors. First, the last partially-filled chunk is read multiple times until it reaches its full size. Second, the chunks that fail relaxed reads are followed up with consistent reads. The latter results in elevated QPS as well.

Monitors consistently reported a p99 message delivery delay of approximately 500ms. This delay stemmed from several factors: producer-side buffering and serial flushing of data and index (120ms), network transit (180ms), and consumer-side periodic polling followed by serial reading of index and data (100ms). Our system added only around 25ms of processing delay along each hop, which included source read I/O, network transmission/receive queuing, and destination write I/O, with the majority of the delay attributed to I/O.

In the following experiments, we capped the number of consumers at 4,000 to ensure we stayed within the resource constraints allocated for each cache replica.

Experiment 1(b) - Abrupt Consumer Spike Utilizing the same cluster setup as in Experiment 1(a), we abruptly scaled the number of consumers from 0 to 4,000 to observe the system's response. This simulates an atypical scenario, like during incident mitigation, where the consumer size surges unexpectedly, putting the system under stress.

Observations: The data cache experienced a sharp increase in read QPS, reaching as high as 4.7M, due to numerous requests for trailing bytes of files stored in non-expired chunks. This surge in read QPS created back pressure on writes because both read and write operations shared the same OS network buffers and bandwidth (lacking isolation at that level), temporarily increasing write latency. Consequently, cache operations encountered failures, forcing consumers to fallback to Colossus to retrieve the necessary bytes. This resulted in a noticeable delivery delay increase of approximately 1s for about 150 seconds as shown in Figure 9. The system eventually recovered and resumed normal performance.

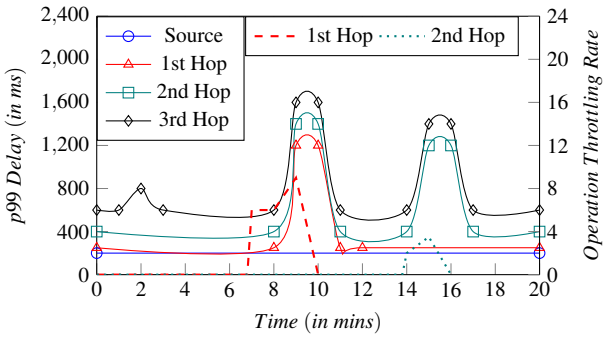


Figure 10: Results for experiment 1(c). The solid lines represent p99 delays and non-solid lines represent throttling caused by imbalance. Delay in a hop impacted the delay in downstream hops.

Experiment 1(c) - Multi-hop Abrupt Consumer Spike We repeated experiment 1(b), but along all the hops of the longest branch of the copy tree of one of the message stream. We turned up 4,000 consumers in 4 clusters, including source.

Observations: The results are shown in Figure 10. We noticed spikes in write latency in the affected clusters, similar to experiment 1(b), which occasionally resulted in operation failures. Although the scheduler rescheduled the operations within 100ms, this rescheduling led to imbalances that caused operation throttling. As a consequence, throttled cache operations triggered Colossus fallbacks. The impact was amplified as a delay in a parent hop affected the entire subtree. The p99 message delivery delay stayed mostly below 600ms, with spikes up to 1.8 seconds coinciding with Colossus fallbacks.

Experiment 1(d) - Fault Tolerance Similar to experiment 1(b), we turned up 4,000 consumers within a single leaf cluster. Additionally, we turned up two backup replicas for both the data cache and the metadata cache. Subsequently, we terminated the cache replicas, followed by the readers and writers, to analyze the resulting impact.

Observations: Data caches demonstrated resilience to single-point failures, maintaining quorum and experiencing no read/write failures when one replica was terminated. However, unavailability and cache operation failures occurred when two replicas of a key-shard were terminated. The backup replica's CPU utilization spiked as it replicated key-value pairs from the remaining replica. This further elongated the unavailability, causing significant Colossus fallbacks of 60,000 reads/seconds and a spike in latency up to 2 seconds as shown in Figure 11. A similar pattern emerged for metadata cache when two replicas of a key-shard were terminated, although recovery was faster due to the significantly smaller number of key-value pairs.

In contrast, terminating readers and writers had a negligible impact on latency (not shown), as operations were swiftly rescheduled by the scheduler within 50ms. However, there was a minor effect on load distribution among the remaining

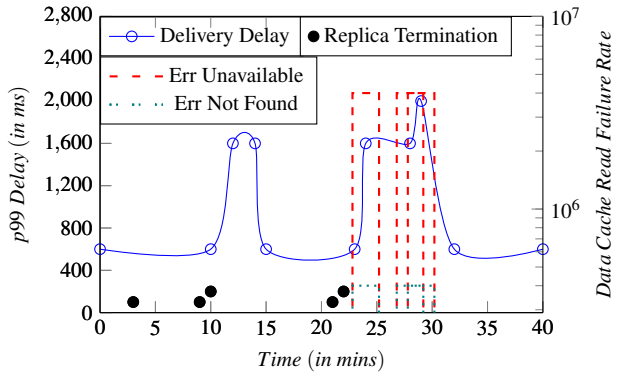


Figure 11: Results for experiment 1(d). Scatter points represent the specific times when a data cache replica for a particular key-shard was terminated. A significant increase in failures was observed, primarily attributed to key-shard unavailability during the repair process.

replicas, which was gradually resolved through load shedding and operation rescheduling.

4.2 Experiment 2

In this experiment, we enabled horizontal scaling for caches, triggered by bandwidth and QPS pressure. We turned up 18 replicas for the data cache and 6 replicas for the metadata cache in one of the leaf cluster. Scaling was configured to trigger whenever a replica breached either 4 Gbps of bandwidth or 400,000 QPS. We simulated a heavier workload by enabling two message streams from clusters in the us-central and euro-west regions. Together, these streams had an average producer write rate of 480 Mbps. The data cache write QPS in each cluster reached approximately 45,000. The message streams were 120-way sharded, and each consumer was assigned 20 shards to read from. Thus, each consumer had a read rate of 80 Mbps. The polling interval was set to 50ms for readers and 500ms for consumers, as a shorter duration led to metadata cache overload.

Experiment 2(a) - Smooth Scaling We began with an initial 1,000 consumers and ramped up to 20,000 consumers, incrementally adding 50 replicas every 10 seconds.

Observations: The maximum read bandwidth observed from the data cache reached 1.8 Tbps as shown in Figure 12. Meanwhile, the metadata cache experienced a peak QPS of 19.2M (20,000 consumers x 2 message streams x 20 shards x 2 files per shard x 2 keys per file x 2 polls per second x 3 replicas). In response to the workload, the data cache scaled up from its initial 18 replicas to 481, surpassing the expected 460 replicas by 4%. This scale-up led to chunk migrations due to re-sharding, causing transient unavailability of the data cache. This, in turn, forced fallbacks to Colossus, resulting in intermittent drops in data cache bandwidth. Despite this, the p99 message delivery delays exported by monitors remained under 2.5 seconds.

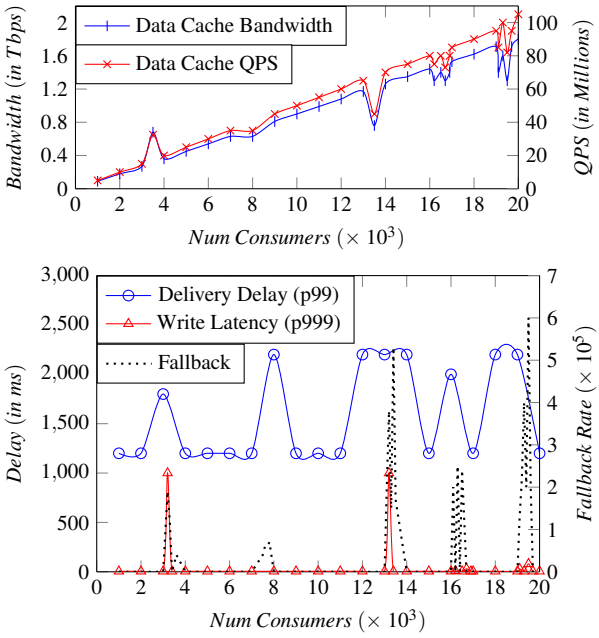


Figure 12: Results for experiment 2(a). Dips were observed in data cache bandwidth and QPS, attributed to two factors. First, the spikes in cache write latency. Second, the transient unavailability of the data cache during scale-ups. These dips were accompanied by fallbacks to Colossus.

The metadata cache also expanded, growing from 6 replicas to 81, which was 68% higher than the anticipated 48 replicas. This steeper-than-expected increase in the metadata cache was due to an uneven distribution of key-value pairs, particularly during scale-up events. This imbalance potentially overloaded certain replicas, triggering further scaling. Increasing the replication factor from r=3 to, say, r=10 could address the metadata cache overload issue, but it would come at the cost of increased tail latency during writes. Unfortunately, CliqueMap's current limitations only allowed for a replication factor of 3.

Another interesting observation is the fact that with 27x more data cache replicas after scale-up, we also increased the chunk storage capacity by 27x and so the TTL of chunks could have been as large as 30 minutes. However, in practice, consumers rarely experience such significant delays. We are exploring strategies to vertically scale-down the caches as they are horizontally-scaled.

Experiment 2(b) - Abrupt Scaling We scaled up the consumers suddenly from 0 to 6,500. While such large-scale increases in consumers are infrequent, they can occur in certain clusters during regional outages.

Observations: The peak data cache bandwidth surged to 645 Gbps, and the metadata cache QPS reached 6.24M. This abrupt increase triggered aggressive scaling in both caches, leading to a significant imbalance that necessitated further scaling. The data cache expanded from 18 to 406 replicas,

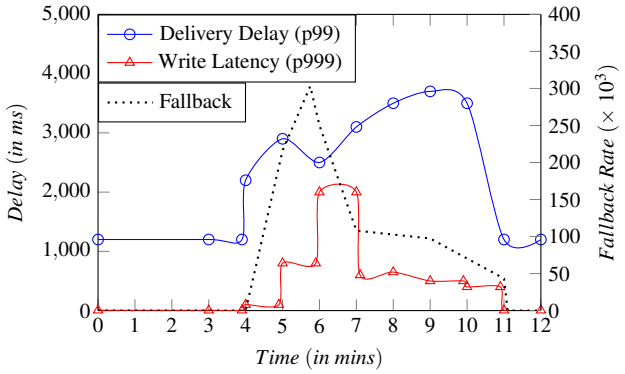


Figure 13: Results for experiment 2(b). The delivery delay was adversely affected for a period of several minutes. A significant number of Colossus fallbacks were observed during this time. Ultimately, the system stabilized and achieved a consistent delivery delay.

while the metadata cache grew from 6 to 29 replicas. During this period, prolonged cache read and write failures were observed, causing a substantial number of fallbacks to Colossus, peaking at 300,000 reads/second. Consequently, the p99 latency exported by monitors spiked beyond 3 seconds as shown in Figure 13. The system eventually recovered after 7 minutes, with delays subsiding to under 1.5 seconds. Once stabilized, the data and metadata caches scaled back down to 171 and 19 replicas respectively, to save resources.

Experiment 2(c) - Backlog Recovery Building on scenario 2(b), we initiated the test with 6,500 consumers, each passively capped at a read rate of 160 Mbps. We then stopped the producer for one of the message stream and started it after a 15-minutes delay. This action resulted in a substantial accumulation of messages awaiting extraction and delivery. Such scenarios can occur when a backlog of messages accumulates due to producer downtime.

Observations: After the producer restart, the peak bandwidth surged to over 1.6 Tbps as shown in Figure 14. This peak was temporary, lasting for only 2 minutes before settling down to 600 Gbps. Due to the transient nature of the spike, the data cache experienced only a minor scale-up from 171 to 184 replicas. The imposed limit on the read rate effectively managed the fan out. However, a significant delivery delay was observed, which eventually resolved once a steady state was reached.

5 Experiences

Development From early design to deployment, the system required an investment of 8 SWE-years. The development process added 17,500 lines of new, non-test C++ code.

The system underwent extensive testing before production deployment. We run regression tests in the pre-production environment before releasing to production. In production, we monitor the metrics outlined in the evaluation and offer

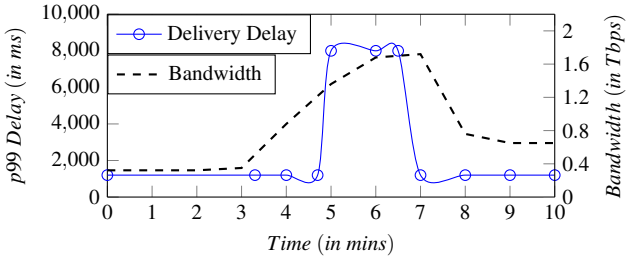


Figure 14: Results for experiment 2(c). A sudden surge in read bandwidth was observed, which lead to high delivery delay.

SLOs on latency and cache byte availability.

We have correctness monitors that ensure the bytes read by consumers exactly match the bytes written by producers. While incorrect bytes in compressed message streams would lead to decompression failures, some message streams are uncompressed where it would be particularly concerning if consumers successfully parse malformed messages.

Rollout Strategy Our system spans dozens of clusters, consisting of numerous jobs (with 3 to 400 replicas each) across scheduler, worker, and storage layers. This complexity necessitates a meticulous rollout strategy. During cache rollouts, we adopt a cautious approach, updating only one replica of a given key-shard at a time. After an update, the replicas are given time to repair their state by retrieving key-value pairs from other replicas.

Worker rollouts cause delay spikes due to throttling and reschedules. Fast rollouts (updating half the replicas at a time) can reduce reschedules but may cause hot-spotting. Conversely, slow rollouts (updating one replica at a time) increases reschedules. We have found that updating one-third of the replicas at once strikes a good balance between hot-spotting and reschedules.

Finally, we never rollout the scheduler and worker jobs together, ensuring scheduler availability during worker rollouts. The rollout strategy discussed so far is slow, sometimes taking several hours. We are continually refining our rollout strategy to strike a better balance between deployment speed and system stability.

Integration with Google Ads The Ads infrastructure relies on an Extract, Transform, and Load [51] pipeline. Extraction and transformation are intra-cluster operations primarily consisting of MapReduce [18, 55] jobs. They extract data from various sources such as transactional databases (e.g., Ads F1 [25, 46]), data warehouses (e.g., Napa [3] and Mesa [26]), and real-time log processors (e.g., Ubiq [48] and Photon [7]), and produce data that is loaded into the serving systems. The extraction and transformation jobs are replicated across a few selected clusters. For instance, each replica of real-time extractors resides in a cluster on a different continent.

The serving systems, on the other hand, are geographically distributed across dozens of clusters. Each job in the Ads

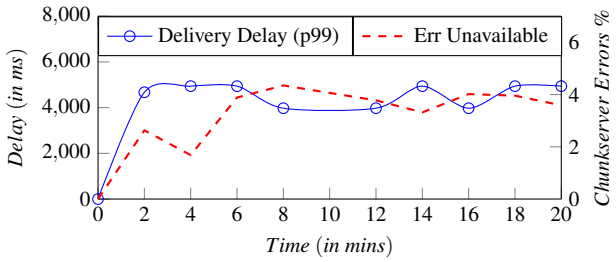


Figure 15: Case of data cache unavailability. Colossus chunkservers throttled read requests resulting in unavailability.

serving system has an in-memory state represented as key-value pairs that is used to serve requests efficiently.

An additional 7,600 lines of non-test C++ code was added for integration with the Ads infrastructure. Each shard of a Google Ads message stream contains two types of entities: *snapshots*, which are compact summaries of a message stream shard up to a logical time [36], and *updates*, which are continuous, ordered sequences of messages on top of the snapshots. The snapshots are used to build an initial server state and the updates are applied on top of it. Each message updates the value for a particular key. Snapshots are atomically copied across clusters by a batch system, while updates are handled by our system. Update files older than the recent snapshots are garbage-collected.

We migrated several Ads message streams to the new stack. The message delivery delay exported by monitors, measured over a day, shows p95, p99, p999, and p9999 latency of 500ms, 630ms, 730ms, and 5.71s respectively. These latency vary depending on the byte rate of the message stream. The largest message stream experienced a p9999 latency of 8.59s over a day. Even within a single message stream, some shards occasionally experience higher delays. The opportunistic reads in Colossus read operations have 96% hit rate, saving disk time on the read path.

We have also noticed latency spikes in certain message streams, some lasting several hours. A few of them were caused by bugs, often introduced due to code complexity. Deadlocks have been particularly challenging to debug because of their latent nature and inability of our cluster-local monitors to distinguish between producer-stopped streams and streams stopped due to worker deadlocks. Occasionally, our jobs were scheduled on bad machines, significantly impacting tail latency [17] without clear symptoms. In such situations, we resort to rescheduling such jobs based on system metrics. Other delays stemmed from infrequent intercontinental network issues caused by link outages. Although the routing process has automations to address this, the passive nature of the actions still results in delays.

Ablation Study In one scenario, the data cache was unavailable, whereas the metadata cache was available. We recorded a substantial QPS to Colossus chunkservers for several hours. Chunkservers throttle based on CPU, memory, and network time usage, in addition to disk time usage. Consumers experienced throttling resulting in unavailability for 3-5% of read requests, thereby resulting in a higher delivery delay (Figure 15). The slight delay intentionally introduced before reads from Colossus (Algorithm 1) proved effective in managing the QPS thereby resulting in a negligible CPU-related unavailability. The vast majority of unavailability (over 99%) stemmed from network congestion.

On another occasion, a faulty configuration push rendered both the data and metadata caches unavailable. We observed latency in the order of minutes, as the Colossus file length was polled directly from Bigtable at a much lower frequency.

Comparison with Predecessor The system before Fast ACS lacked the capability to handle large consumer fanout. With limited fanout, Fast ACS demonstrates greater tail latency stability due to the use of RMA. Furthermore, the adoption of RMA has led to a significant reduction in server-side CPU usage, resulting in cost savings exceeding one-third of the previous total cost.

6 Key Takeaways

This work yielded several valuable insights for designing scalable, low-latency messaging systems:

1. NIC capacity will be the bottleneck of future network-intensive computing: Given the limited near-term prospects for widespread terabit NIC deployment, achieving ordered data delivery at scale requires designs that efficiently utilize existing NIC capabilities. Our system addresses this by horizontally scaling resources to maximize cluster's NIC capacity utilization.

2. Ordered byte delivery does not need in-order transmission across the network: Instead, parallel out-of-order fetching of data chunks, followed by client-side reassembly, can effectively overcome server-side throughput constraints for low-latency delivery. This concept is already employed in web clients that perform parallel download of file chunks from multiple servers. Our system leverages this principle by chunking message files, pulling these chunks out-of-order, and then assembling and delivering them to consumers in the correct sequence. Through consistent chunk distribution, we also avoid hot-spotting in all scenarios.

3. RMA offers a powerful paradigm for building highly efficient software by minimizing server-side processing overhead: Our implementation strategically utilizes RMA on the client side, reserving server-side computation for fallback scenarios, thus providing significant resource savings.

7 Related Works

Distributed storage systems are evolving to leverage modern hardware capabilities. For example, NVM- and RDMA-aware HDFS [29] provides remote SSD reads for file bytes.

Octopus [57] enhances this further with low-latency file metadata access. Colossus also supports similar functionality with SSDs and CliqueMap, too, can use SSDs as the underlying storage. However, SSDs are not ideal for our needs for several reasons. First, their lifespan would be limited due to wear-leveling caused by continuous writes. Second, they offer lower read throughput and higher latency compared to RAM. Third, our hot storage requirements are minimal, not justifying the capacity of SSDs.

Tachyon [37] is an RMA-based in-memory file system available for big data processing. However, it is not optimized for tail reading due to its large block sizes, which can lead to hot-spots on the tail block servers. Reducing the block size can help, but it may lead to latency hits on read path since block metadata access still relies on RPCs. Our approach of distributing 4 KB chunks as key-value pairs significantly reduces hot-spotting during tail reads.

Memcache [40] is a popular productionized distributed in-memory key-value store. It provides low-latency storage and lookup and integrates well with external sharding engines like Slicer [2] to offer scalability. However, we decided to use CliqueMap due to two reasons. First, CliqueMap supports replication out-of-the-box, which makes it resilient to a single point of failure and makes our rollout process simple, unlike Memcache, which requires certain wrappers to support replication. Second, CliqueMap makes use of RMA on the read path, achieving significantly higher throughput and saving valuable server-side CPU while maintaining a comparable median latency. Other RMA key-value systems like FaRM-KV [20] could provide serialized consistent lookups using primary/backup architecture; however, serialization-based consistency was not required in our design, and so we opted for a simpler quorum-based system. Pilaf [39] is one such RMA KV-store with quorum-based support and resembles performance similar to CliqueMap. However, CliqueMap is already productionized at Google and serves as a backend for critical infrastructures in Ads, Maps, and YouTube.

Lastly, while there is extensive research on userspace file systems like FUSE [50] and its extension, extFUSE [8], which provide a VFS-like interface for creating custom file systems, we chose not to use them. First, our needs were limited to dealing with files, not directories. This allowed us to utilize hash-based addressing instead of a hierarchical file system and a key-value store was a natural fit. Second, our caching file system was essentially a reflection of the underlying Colossus file system supporting append-only files. Implementing a full file system was unnecessary.

8 Future Work

To ensure fair and efficient caching for a growing number of message streams, we are exploring key-space isolation to prevent resource contention. Currently, the shared data cache can be overused by individual streams, impacting others. By providing dedicated cache space, each stream can independently manage its usage and chunk expiration, improving overall system stability and fairness.

Each Google cluster has a capacity limit ranging from hundreds to tens of thousands of machines. During experiment 2, we observed that multiple consumers were running on the same machines, sometimes up to 10. One effective technique to limit intra-cluster bandwidth would be to implement a sidecar ambassador process [9] that retrieves messages onto local machines and relays them to consumers. This approach would allow us to scale up to millions of consumers per cluster while enforcing a limit on anticipated data cache bandwidth and metadata cache QPS.

In experiment 2, we observed that the primary factor driving up the latency was the increased consumer polling interval. Reducing the polling interval can help bring down latency but can cause QPS limits breach on the metadata cache. There can be many strategies to mitigate the QPS problem. First, the bandwidth utilization per metadata cache replica is low and so the network is essentially non-constraining; the QPS limits can be raised by increasing the number of software RMA endpoints. Second, because the write QPS is much lower than read QPS, we can combine metadata for multiple files into a single key-value pair. This combination can be automated based on the consumer read pattern.

Finally, we are investigating the potential of RMA and other network enhancements to improve performance in broader scenarios, such as cross-cluster RMA-based reads. However, current network deployment limitations prevent us from supporting this functionality in all cases.

9 Conclusion

This paper presents the design of Fast ACS, a file-based ordered message delivery system. At its core is a multi-layer storage that has been demonstrated to be a valuable component, helping the system achieve Tbps-scale consumer fan-out per cluster and deliver messages with minimal latency. This system has been widely deployed in production.

Acknowledgments

We gratefully acknowledge contributions to the design and implementation of the system from many colleagues at Google. Special mention for Aman Shaikh, Apar Madan, Ayush Jha, Ben T. Harper, Bin Lyu, Chen-Han Ho, Connor Quagliana, Erin Rovelstad, Florentina I. Popovici, Hongchen Li, Lily Chen, Manav Prajapati, Mengjie Xia, Nick Golob, Paul D. Bartlett, Serafi Zanikolas, Sucharitha Vasudevan, Tao Huang, and Youer Pu. We also thank Jeff Mogul, Tao Huang, and Ashish Gupta for their helpful suggestions on the presentation of the work. Finally, we thank the anonymous reviewers and our shepherd, Fernando Pedone, for their feedback.

References

[1] Colin Adams, Luis Alonso, Benjamin Atkin, John Banning, Sumeer Bhola, Rick Buskens, Ming Chen, Xi Chen, Yoo Chung, Qin Jia, et al. Monarch: Google's planet-scale in-memory time series database. *Proceedings of the VLDB Endowment*, 13(12):3181–3194, 2020.

[2] Atul Adya, Daniel Myers, Jon Howell, Jeremy Elson, Colin Meek, Vishesh Khemani, Stefan Fulger, Pan Gu, Lakshminath Bhuvanagiri, Jason Hunter, et al. Slicer:{Auto-Sharding} for datacenter applications. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 739–753, 2016.

[3] Ankur Agiwal, Kevin Lai, Gokul Nath Babu Manoharan, Indrajit Roy, Jagan Sankaranarayanan, Hao Zhang, Tao Zou, Min Chen, Jim Chen, Ming Dai, Thanh Do, Haoyu Gao, Haoyan Geng, Raman Grover, Bo Huang, Yanlai Huang, Adam Li, Jianyi Liang, Tao Lin, Li Liu, Yao Liu, Xi Mao, Maya Meng, Prashant Mishra, Jay Patel, Rajesh S R, Vijayshankar Raman, Sourashis Roy, Mayank Singh Shishodia, Tianhang Sun, Justin Tang, Junichi Tatemura, Sagar Trehan, Ramkumar Vadali, Prasanna Venkatasubramanian, Joey Zhang, Kefei Zhang, Yupu Zhang, Zeleng Zhuang, Goetz Graefe, Divyakanth Agrawal, Jeff Naughton, Sujata Sunil Kosalge, and Hakan Hacıgümüş. Napa: Powering scalable data warehousing with robust query performance at google. *Proceedings of the VLDB Endowment (PVLDB)*, 14 (12):2986–2998, 2021.

[4] Aditya Akella, Amanda Strominger, Amin Vahdat, Arjun Singhvi, Dan Gibson, Harshad Deshmukh, Maggie Anderson, Milo M. K. Martin, Rob Cauble, and Thomas F. Wenisch. Cliquemap: Productionizing an rma-based distributed caching system. 2021.

[5] Tyler Akidau, Alex Balikov, Kaya Bekiroğlu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, and Sam Whittle. Millwheel: Fault-tolerant stream processing at internet scale. *Proceedings of the VLDB Endowment*, 6(11):1033–1044, 2013.

[6] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, et al. The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *Proceedings of the VLDB Endowment*, 8(12):1792–1803, 2015.

[7] Rajagopal Ananthanarayanan, Venkatesh Basker, Sumit Das, Ashish Gupta, Haifeng Jiang, Tianhao Qiu, Alexey Reznichenko, Deomid Ryabkov, Manpreet Singh, and Shivakumar Venkataraman. Photon: Fault-tolerant and scalable joining of continuous data streams. In *SIGMOD '13: Proceedings of the 2013 international conference on Management of data*, pages 577–588, New York, NY, USA, 2013.

[8] Ashish Bijlani and Umakishore Ramachandran. Extension framework for file systems in user space. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 121–134, 2019.

[9] Brendan Burns and David Oppenheimer. Design patterns for container-based distributed systems. In *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*, 2016.

[10] Mike Burrows. The chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 335–350, 2006.

[11] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache flink: Stream and batch processing in a single engine. *The Bulletin of the Technical Committee on Data Engineering*, 38(4), 2015.

[12] Antonio Carzaniga, David S Rosenblum, and Alexander L Wolf. Achieving scalability and expressiveness in an internet-scale event notification service. In *Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing*, pages 219–227, 2000.

[13] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):1–26, 2008.

[14] Binildas Christudas and Binildas Christudas. Activemq. *Practical Microservices Architectural Patterns: Event-Based Java Microservices with Spring Boot and Spring Cloud*, pages 861–867, 2019.

[15] Google Cloud. https://cloud.google.com/. Accessed: 2024-10-03.

[16] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)*, 31(3):1–22, 2013.

[17] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Communications of the ACM*, 56:74–80, 2013.

[18] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[19] David Dossot. *RabbitMQ essentials*. Packt Publishing Ltd, 2014.

[20] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. {FaRM}: Fast remote memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 401–414, 2014.

[21] The Apache Software Foundation. Apache bookkeeper. https://bookkeeper.apache.org/. Accessed: 2024-10-03.

[22] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 20–43, Bolton Landing, NY, 2003.

[23] Ramesh Govindan, Ina Minei, Mahesh Kallahalla, Bikash Koley, and Amin Vahdat. Evolve or die: High-availability design principles drawn from googles network infrastructure. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 58–72, 2016.

[24] gRPC Authors. Core concepts, architecture, and lifecycle | grpc. https://grpc.io/docs/what-is-grpc/core-concepts/#bidirectional-streaming-rpc. Accessed: 2024-10-03.

[25] Ashish Gupta and Jeff Shute. High-availability at massive scale: Building google's data infrastructure for ads. In *Workshop on Business Intelligence for the Real Time Enterprise (BIRTE)*, 2015.

[26] Ashish Gupta, Fan Yang, Jason Govig, Adam Kirsch, Kelvin Chan, Kevin Lai, Shuo Wu, Sandeep Dhoot, Abhilash Kumar, Ankur Agiwal, Sanjay Bhansali, Mingsheng Hong, Jamie Cameron, Masood Siddiqi, David Jones, Jeff Shute, Andrey Gubarev, Shivakumar Venkataraman, and Divyakant Agrawal. Mesa: Geo-replicated, near real-time, scalable data warehousing. In *VLDB*, 2014.

[27] Dean Hildebrand and Denis Serenyi. Colossus under the hood: a peek into google's scalable storage system. https://cloud.google.com/blog/products/storage-data-transfer/a-peek-behind-Colossus-googles-file-system, 2021. Accessed: 2024-08-02.

[28] Patrick Hunt, Mahadev Konar, Flavio P Junqueira, and Benjamin Reed. {ZooKeeper}: Wait-free coordination for internet-scale systems. In *2010 USENIX Annual Technical Conference (USENIX ATC 10)*, 2010.

[29] Nusrat Sharmin Islam, Md Wasi-ur Rahman, Xiaoyi Lu, and Dhabaleswar K Panda. High performance design for hdfs with byte-addressability of nvm and rdma. In *Proceedings of the 2016 International Conference on Supercomputing*, pages 1–14, 2016.

[30] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, et al. B4: Experience with a globally-deployed software defined wan. *ACM SIGCOMM Computer Communication Review*, 43(4):3–14, 2013.

[31] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pages 654–663, 1997.

[32] David Kjerrumgaard. *Apache Pulsar in action*. Simon and Schuster, 2021.

[33] Jay Kreps, Neha Narkhede, Jun Rao, et al. Kafka: A distributed messaging system for log processing. In *Proceedings of the NetDB*, volume 11, pages 1–7. Athens, Greece, 2011.

[34] SPT Krishnan, Jose L Ugia Gonzalez, SPT Krishnan, and Jose L Ugia Gonzalez. Google cloud pub/sub. *Building Your Next Big Thing with Google Cloud Platform: A Guide for Developers and Enterprise Architects*, pages 277–292, 2015.

[35] Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, June 2002.

[36] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. In *Concurrency: the Works of Leslie Lamport*, pages 179–196. 2019.

[37] Haoyuan Li, Ali Ghodsi, Matei Zaharia, Scott Shenker, and Ion Stoica. Tachyon: Reliable, memory speed storage for cluster computing frameworks. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 1–15, 2014.

[38] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Mike Dalton, Nandita Dukkipati, William C. Evans, Steve Gribble, Nicholas Kidd, Roman Kononov, Gautam Kumar, Carl Mauer, Emily Musick, Lena Olson, Mike Ryan, Erik Rubow, Kevin Springborn, Paul Turner, Valas Valancius, Xi Wang, and Amin Vahdat. Snap: a microkernel approach to host networking. In *In ACM SIGOPS 27th Symposium on Operating Systems Principles*, New York, NY, USA, 2019.

[39] Christopher Mitchell, Yifeng Geng, and Jinyang Li. Using {One-Sided}{RDMA} reads to build a fast,{CPU-Efficient}{Key-Value} store. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 103–114, 2013.

[40] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, et al. Scaling memcache at facebook. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 385–398, 2013.

[41] Robert Clay Prim. Shortest connection networks and some generalizations. *The Bell System Technical Journal*, 36(6):1389–1401, 1957.

[42] Antony Rowstron, Anne-Marie Kermarrec, Miguel Castro, and Peter Druschel. Scribe: The design of a large-scale event notification infrastructure. In *Networked Group Communication: Third International COST264 Workshop, NGC 2001 London, UK, November 7–9, 2001 Proceedings 3*, pages 30–43. Springer, 2001.

[43] Krzysztof Rzadca, Pawel Findeisen, Jacek Swiderski, Przemyslaw Zych, Przemyslaw Broniek, Jarek Kusmierek, Pawel Nowak, Beata Strack, Piotr Witusowski, Steven Hand, et al. Autopilot: workload autoscaling at google. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–16, 2020.

[44] Yogeshwer Sharma, Philippe Ajoux, Petchean Ang, David Callies, Abhishek Choudhary, Laurent Demailly, Thomas Fersch, Liat Atsmon Guz, Andrzej Kotulski, Sachin Kulkarni, et al. Wormhole: Reliable {Pub-Sub} to support geo-replicated internet services. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 351–366, 2015.

[45] Michael Shields. Leap-smeared representation of time for high-accuracy applications. 2016.

[46] Jeff Shute, Radek Vingralek, Bart Samwel, Ben Handy, Chad Whipkey, Eric Rollins, Mircea Oancea, Kyle Littlefield, David Menestrina, Stephan Ellner, John Cieslewicz, Ian Rae, Traian Stancescu, and Himani Apte. F1: A distributed sql database that scales. In *VLDB*, 2013.

[47] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *2010 IEEE 26th symposium on mass storage systems and technologies (MSST)*, pages 1–10. Ieee, 2010.

[48] Alexander Smolyanov, Ashish Gupta, Divy Agrawal, Haifeng Jiang, Manish Bhatia, Manpreet Singh, Monica Chawathe Lenart, Namit Sikka, Navin Melville, Scott Holzer, Shan He, Shivakumar Venkataraman, Tianhao Qiu, Venkatesh Basker, Vinny Ganeshan, and Yuri Vasilevski. Ubiq: A scalable and fault-tolerant log processing infrastructure. In *Workshop on Business Intelligence for the Real Time Enterprise (BIRTE)*, 2016.

[49] Robert Strom, Guruduth Banavar, Tushar Chandra, Marc Kaplan, Kevan Miller, Bodhi Mukherjee, Daniel Sturman, and Michael Ward. Gryphon: An information flow based approach to message brokering. *arXiv preprint cs/9810019*, 1998.

[50] Bharath Kumar Reddy Vangoor, Vasily Tarasov, and Erez Zadok. To {FUSE} or not to {FUSE}: Performance of {User-Space} file systems. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 59–72, 2017.

[51] Panos Vassiliadis and Alkis Simitsis. Extraction, transformation, and loading. *Encyclopedia of Database Systems*, 10:14, 2009.

[52] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at google with borg. In *Proceedings of the tenth european conference on computer systems*, pages 1–17, 2015.

[53] Tzu-Wei Yang, Seth Pollen, Mustafa Uysal, Arif Merchant, and Homer Wolfmeister. {CacheSack}: Admission optimization for google datacenter flash caches. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 1021–1036, 2022.

[54] Yuan Yu, Panagiotis Manolios, and Leslie Lamport. Model checking tla+ specifications. In *Advanced research working conference on correct hardware design and verification methods*, pages 54–66. Springer, 1999.

[55] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *2nd USENIX workshop on hot topics in cloud computing (HotCloud 10)*, 2010.

[56] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the twenty-fourth ACM symposium on operating systems principles*, pages 423–438, 2013.

[57] Bohong Zhu, Youmin Chen, Qing Wang, Youyou Lu, and Jiwu Shu. Octopus+: An rdma-enabled distributed persistent memory file system. *ACM Transactions on Storage (TOS)*, 17(3):1–25, 2021.

A Formal Specification for Dueling Writers

We developed a formal specification for the system, which consists of multiple writers for the cache file and a single writer for the corresponding Colossus file. The system was modeled using TLA+ [35]. The goal was to check for correctness of the system, manifested in the following way:

- *Safety*: Consumers receive bytes strictly in the order they were produced. This is formally expressed by the following invariant:

  $Invariant ==$

  $$\wedge\ readBytes = [i \in 1..Len(readBytes) \mapsto kBytes[i]]$$

  where *readBytes* are the bytes read by the consumer and *kBytes* are the bytes written by the producer in source.

- *Eventual Progress*: The consumers eventually receives all the bytes. Formally, the following property is satisfied:

  $Termination ==$

  $$\Diamond\ ((pc = \text{“}done\text{”}) \implies (readBytes = kBytes))$$

  where *pc = "done"* is the termination marker.

The system's safety is intuitively clear: data cache chunks always contain the correct bytes, if present, corresponding to the file positions, and each chunk write is atomic. The corresponding Colossus file serves as a safety net for partially-filled or missing chunks.

Eventual progress is ensured because the Colossus file continually grows in length, eventually forcing consumers to fall back and fetch bytes directly from Colossus.

We used the TLA+ Model Checker [54] to verify the specification under two configurations:

- *Configuration 1*: 4 chunks, each with 4 bytes, and 2 cache writers. This resulted in 85,690,897 distinct states.

- *Configuration 2*: 2 chunks, each with 2 bytes, and 4 cache writers. This resulted in 12,062,699 distinct states.

In both runs, the model checker successfully verified the system's safety and eventual progress.

Poby: SmartNIC-accelerated Image Provisioning for Coldstart in Clouds

Zihao Chang[1,2], Jiaqi Zhu[1,2], Haifeng Sun[3†], Yunlong Xie[1,2], Kan Shi[1,2], Ninghui Sun[1,2], Yungang Bao[1,2], Sa Wang[1,2†]

[1] *SKLP, Institute of Computing Technology, CAS*
[2] *University of Chinese Academy of Sciences* [3] *Peking University*

Abstract

Coldstart introduces a significant latency penalty in cloud computing. While several previous works have proposed mechanisms such as warm start, fast snapshot recovery, lightweight isolation, and fast image download to avoid or mitigate this issue, image provisioning remains underexplored despite being critical.

In this paper, we propose `Poby`, a software-hardware collaborative system that offloads and accelerates critical operations of image provisioning using SmartNICs. Specifically, `Poby` embodies a disaggregated architecture that offloads different image provisioning operations to the appropriate hardware such as embedded CPUs and domain-specific hardware accelerators for optimal performance. It uses a pipeline-based, data-driven workflow to eliminate delays caused by the serial execution of image provisioning operations. Moreover, it contains a distributed image provisioning scheme to alleviate the performance bottlenecks of conventional centralized registries. We implement the entire `Poby` system using BlueField SmartNICs and evaluate its performance using various microservice and FaaS benchmark suites. The results demonstrate that `Poby` outperforms two industry-standard container platforms, containerd and iSulad, with speedups of 13.2× and 8.0×, respectively. In addition, compared to iSulad, it reduces host CPU usage by 87.5%.

1 Introduction

Cloud computing has long promised on-demand scaling since its inception. This feature allows the cloud system to automatically adjust hardware resources to meet the dynamic needs of cloud applications, eliminating the need for human intervention. However, this convenience also presents significant challenges in cloud management. The coldstart problem is one of the prominent performance issues [5, 64, 83]. When an incoming request triggers an on-demand container creation and runtime initialization, the response time can be significantly longer - almost 10× compared against a warm startup [41], which negatively impacts the user experience.

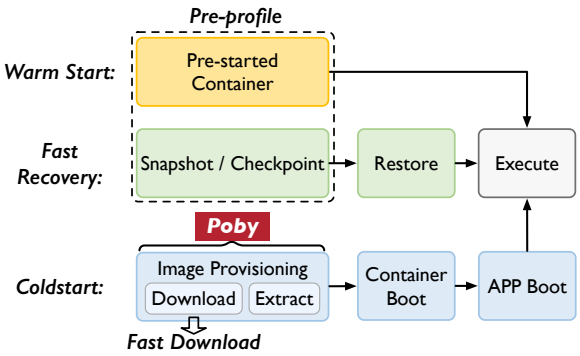


Figure 1: Overview of optimizations for coldstart.

Within coldstart, the most time-consuming step is *image provisioning*, which involves *downloading* and *extracting* large images from remote storage into local memory. Therefore, as shown in Figure 1, state-of-the-art (SOTA) work towards coldstart mainly focuses on the image provisioning part. The *warm start* method [1, 8, 51, 60, 61, 64, 68] prestarts containers before invocation and keeps them running continuously. However, this introduces considerable resource overhead and breaks the promise of on-demand scaling. The *fast recovery* method [3, 11, 19, 54, 62, 73, 80], which relies on snapshots or checkpoints to boot containers, avoids the time-consuming image provisioning process. Still, this approach incurs substantial storage overheads and requires containers to be stateless, which is challenging in production environments.

Both warm start and fast recovery methods attempt to *bypass* the image provisioning part. However, in fact, it is impossible to completely avoid coldstart in the production environment, such as warm start prediction errors, the first request to a newly uploaded function, and so on. In such scenarios, some work directly *accelerates* the image provisioning part, such as accelerating image download through a peer-to-peer approach [10, 39, 72, 76] (*fast download*). Nevertheless, we identify a counter-intuitive issue where *image extraction* takes more time than *image download*. The *extraction* step, which

[†]Corresponding authors.

decompresses and unpacks image files to initiate the container booting process, consumes over 68.8% of the overall image provisioning latency (details in §2.2).

To accelerate the entire image provisioning process for coldstart, we propose a software-hardware collaborative system named Poby. Instead of consuming valuable host CPU resources for compute-intensive tasks such as image compression and extraction, we offload those tasks directly onto modern SmartNICs with specific hardware accelerators for better performance and reduced CPU usage. In addition, most modern SmartNICs natively support zero-copy network protocols such as RDMA [14, 16, 48, 53], thus enabling low-latency and high-bandwidth image downloads.

However, porting image provisioning to SmartNIC is a nontrivial task due to the following three reasons. Firstly, the embedded CPUs of existing SoC-based SmartNIC lack sufficient computing power. Naively offloading the entire image provisioning process to SmartNIC can be counterproductive. Secondly, the sequential execution of the image provisioning operations causes significant waiting delays. This issue worsens on SmartNIC because images must be transferred between the NIC memory and the host memory. Finally, the high-bandwidth image download via RDMA could also overload the centralized image registry, potentially creating a new performance bottleneck.

To address these challenges, we design and implement a *disaggregated architecture* in Poby, which offloads various image provisioning operations, including control of image provisioning, image downloads, image decompression, and image unpacking, to optimal hardware such as the embedded SmartNIC CPU, the RDMA logic, the decompression accelerator, and the host CPU, accordingly. Poby features a *pipeline-based data-driven workflow* to eliminate the waiting delay introduced by serial execution of image provisioning operations. Furthermore, Poby *decentralizes the image registry* to alleviate the potential performance bottlenecks of the conventional centralized registry.

In this paper, we implement Poby using the Nvidia BlueField-2 SmartNIC [14], which is equipped with eight ARMv8 A72 CPUs and various domain-specific hardware accelerators for network and storage. We build a dual-card prototyping system to evaluate the effectiveness, efficiency, and scalability of the proposed design. Our evaluation shows that in comparison to the SOTA implementations containerd [23] and iSulad [55], Poby achieves 13.2× and 8.0× speed-ups, respectively. Furthermore, compared to the SOTA image download works Kraken and FaaSNET, which use high-performance host CPUs, Poby achieves comparable scalability using the embedded SmartNIC CPUs.

To the best of our knowledge, Poby is the first software-hardware collaborative system to accelerate the complete image provisioning process for coldstart. In summary, we make the following contributions:

- We observe that image extraction is critical but time-consuming for image provisioning. Accelerating the entire image provisioning process is essential to mitigate the coldstart overhead.
- We propose a novel software-hardware collaborative system named Poby, which features disaggregated architecture, pipeline-based data-driven workflow, and decentralized image registry to accelerate the end-to-end image provisioning process.
- We prototype and evaluate Poby on BlueField-2 SmartNICs with realistic cloud workloads and demonstrate that Poby achieves up to 13.2× performance improvements over prior works.

2 Background and Motivation

2.1 Coldstart and Prior Optimizations

Coldstart is a classic problem in cloud computing, especially in container-based microservice architecture and serverless computing. For flexible service management, large-scale online service providers deploy single-purpose microservices or fine-grained function instances into containers, which offer a lightweight and isolated virtualization environment. In such cases, a coldstart refers to the delay caused by inactivity when a microservice instance or function is first invoked. In addition, a surge in service requests will also trigger a large number of coldstarts. A complete coldstart process consists of three steps: (i) preparing the container images (including download, decompress and unpack), (ii) launching the container, and (iii) initializing the application. Unfortunately, the coldstart process can be time-consuming and may potentially bottleneck the overall system performance.

The container image encompasses all the necessary components for a container to operate. According to the OCI Image specification [27], the image is composed of an image manifest and a collection of filesystem layers. The manifest is a general description and a list of layers contained in the image. The layers themselves are compressed filesystems and changes of the filesystem. These layers can be serialized to a container filesystem according to the manifest.

Limitations of prior optimizations. As shown in Figure 1, various optimization technologies have been proposed to alleviate the coldstart overhead. However, these solutions still suffer from different limitations:

- *Fast download.* Some works [39, 72, 76] focus on optimizing the image download. They apply a peer-to-peer (P2P) approach to accelerate image download and improve the scalability. Although image download is one of the most time-consuming operations within coldstart, they ignore the overhead of image extraction, i.e., decompressing and unpacking the downloaded images before launching the container. Note that image extraction is also a critical phase in coldstart (see details in §2.2). Works like [10, 39, 76]

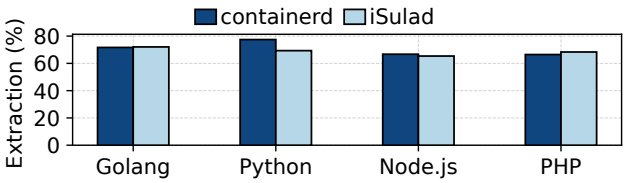


Figure 2: Relative extraction time in image provisioning.

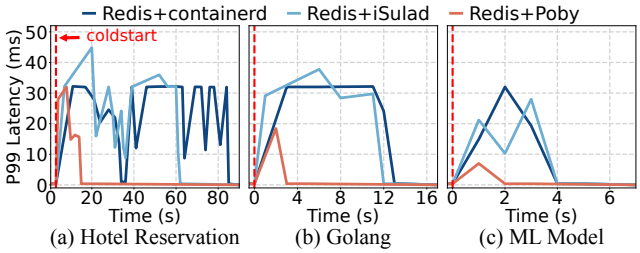
adopt new image formats to enable on-demand downloading and decompression, but this makes them completely incompatible with traditional image distribution systems.

- *Fast recovery.* Some studies maintain the *snapshot / checkpoint* of a booted container in memory [11, 54, 80–82] or storage [3, 19, 33, 62, 73] for *fast recovery*. In this way, these systems bypass the image provisioning when starting a previously launched container. However, the diversity of microservices and function instances poses considerable challenges for the management of numerous snapshots. Firstly, it requires a substantial amount of memory or storage to create and maintain snapshots of booted containers. Secondly, the application in the container snapshot must be *stateless*. This means that it cannot retain any information about previous interactions. Because recovering stateful data (e.g., timestamps, statistics, and request data) from the snapshot can introduce anomalies and privacy concerns.
- *Warm start.* Prior works [6, 13, 49, 75] adopt the *keep-alive* strategy to avoid the coldstart penalty. However, such approaches incur additional overhead for running containers even when there are no requests. On the one hand, such overhead is unacceptable for platforms that manage a large number of containers. On the other hand, it breaks the fundamental promise of *pay-as-you-go* in cloud computing. To reduce the keep-alive overhead, several studies [1, 8, 61, 64] predict the occurrence of coldstart and preemptively start containers. But this approach requires much effort to learn coldstart patterns for high prediction accuracy, and is ineffective for newly deployed images. Other works [51, 60, 68] maintain a shared container pool to reduce overhead, while this approach is not applicable to user-provided images.

2.2 Motivation

When looking into the coldstart process, the following three observations motivate us to completely address this issue.

① **Coldstart from image provisioning is inevitable.** Existing approaches such as fast recovery and warm start aim to bypass image provisioning at the cost of massive memory caching. However, (i) it is not cost-effective to cache *all* containers in memory. For example, CodeCrunch [6], a SOTA warm start work, achieves an 80% warm start rate with surprisingly over 90 TB peak memory usage. Thus it is impractical and inefficient to cache all the containers in limited memory resources, since requests are concentrated on a subset of hot containers [64, 76, 81]. (ii) Some works [1, 8, 61, 64] try to predict the upcoming needs for containers and provide on-demand



(a) Hotel Reservation (b) Golang (c) ML Model

Figure 3: Tail latency of Redis when colocated individually with containerd, iSulad, and `Poby` during the coldstart.

memory caching to reduce memory costs. However, these predictions are by no means completely accurate, especially in production. According to the trace of Alibaba Cloud [17], the average coldstart frequency for each container in the platform still exceeds 130 times in one day.

② **Image provisioning is the most time-consuming step of coldstart.** According to the statistics [76] of 712,295 coldstart operations for containerized functions within 15 days from Alibaba Cloud, over half of function requests spend at least 72% of the total time on image provisioning. And more than 57% of these operations take at least 45 seconds. Although factors such as image size, compression rate, and the number of layers may impact the processing time, image provisioning remains the most time-consuming step of coldstart [51, 68, 74].

③ **Recent efforts ignore the image extraction process.** Existing works only focus on accelerating the image download [39, 72, 76], overlooking the extraction process of image provisioning. Remarkably, we find that image extraction (i) is not only the de facto performance bottleneck in image provisioning (ii) but also introduces severe performance interference to other applications.

- *Performance bottleneck.* Figure 2 evaluates the image extraction time of two well-known container solutions, containerd [23] and iSulad [55], across language runtimes of different sizes. Notably, the image extraction takes more than 68.8% of the image provisioning time in all kinds of runtimes.
- *Performance interference.* Figure 3 evaluates the performance interference of Redis when colocated with containerd and iSulad, respectively. We measure the latency of Redis during the extraction of three image sizes (see Table 1). The latency of Redis increased by over 110 × due to colocation. This is attributed to the resource-intensive nature of image extraction, which involves decompression and file writing. Firstly, decompression is CPU-intensive, as it executes intricate decoding operations to reverse complex encoding patterns. Secondly, the unpacking operation requires writing a large number of small files in images, which demands frequent CPU involvement.

Thus, we argue that the image extraction should be carefully re-examined and accelerated.

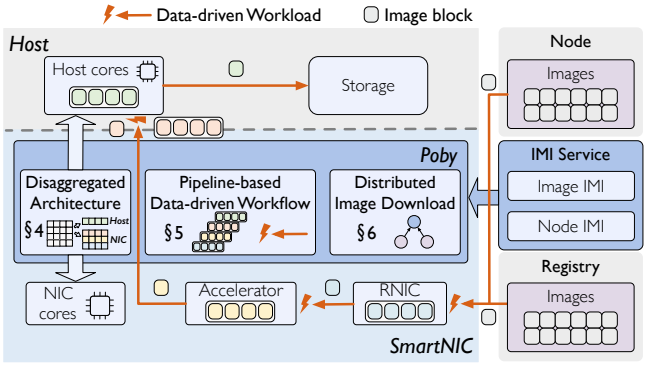


Figure 4: Overview of the Poby system.

3 Poby Overview

Poby is a novel software-and-hardware collaborative system designed to accelerate image provisioning of the coldstart process. The overall architecture is shown in Figure 4. As can be seen, a key feature of Poby is its ability to offload critical tasks related to image provisioning onto modern SmartNICs, thus achieving hardware acceleration using domain-specific accelerators, as well as saving valuable host CPU resources from processing cloud infrastructure tasks.

We define the following main ***goals*** for Poby:

- ***Performance*:** Poby aims to complement existing mitigation methods by accelerating the overlooked decompression process, and fundamentally resolve the coldstart issue.
- ***Resource*:** Poby offloads coldstart to SmartNIC for reducing resource contention and datacenter *tax* [29].
- ***Compatibility*:** Poby provides an optional auxiliary acceleration capability for coldstart, it does not refactor the original design and software stack of container platforms.

In recent years, SmartNICs have been widely adopted in data centers [21, 28, 69] and many researches apply SmartNIC to offload applications such as storage [38, 50], distributed system [31, 63], and network protocol [4, 32, 52, 65]. In this work, we take advantage of several key features of modern SmartNICs. For example, they natively support zero-copy network protocols such as RDMA, potentially enabling low-latency and high-bandwidth image download. In addition, SmartNICs embody hardware accelerators and computing engines for various scenarios including data compression and management, thus ideally suitable for accelerating image extraction and addressing resource contention issues.

3.1 Challenges

Although accelerating the image provisioning by the SmartNIC seems intuitive, it is non-trivial to put into practice due to the following **C**hallenges.

C1: Offloading dilemma. The strawman solution that offloads the complete image provisioning onto the SmartNIC cannot achieve good performance. The reasons accounting for this include two aspects. Firstly, the on-NIC CPU of existing SoC-based SmartNICs is wimpy, which is only suitable for offloading applications with low IPC (instructions per cycle) or high cache MPKI (misses per kilo-instruction) [31, 44, 78]. That is to say, the performance of image extraction on the NIC is much lower than on the host. Secondly, even if we offload the image extraction to the specific hardware accelerator or customized FPGA module, the transmission of numerous extracted files becomes the new bottleneck. Here, most container images are comprised of many layers.

C2: Serial execution of image provisioning. The serial execution of image provisioning operations is a typical store-and-forward workflow, which results in significant waiting delays. Specifically, each operation must suffer a delay of waiting for the previous operation to complete, even if the preceding operation is a time-consuming operation like downloading and decompression. This issue is particularly severe for latency-sensitive and large containers. In addition, offloading image provisioning to the SmartNIC introduces an additional operation of transferring image data from the NIC memory to the host memory, which exacerbates the problem.

C3: Centralized image registry. Although RDMA can boost image download performance, the classic centralized image registry remains a bottleneck. Our tests reveal that concurrent downloads from eight nodes to one registry cause a 10× and 6× increase in latency with TCP and RDMA, respectively. Traditionally, cloud providers mitigate this by scaling registry nodes and using the load balancer. However, prior research shows that requests to a single service can spike dramatically, incurring hundreds of nodes to pull the same images simultaneously [10, 64, 76]. This necessitates deploying numerous additional registries, contradicting our design goal. Moreover, the centralized registry struggles with RDMA scalability. The RDMA NIC (RNIC) of the registry has to frequently switch the queue pair information in its cache to transmit images to different hosts, which seriously impacts the performance.

3.2 Poby Design

As shown in Figure 4, Poby integrates three solutions to accelerate image provisioning. The key insight of Poby is to orchestrate appropriate hardware resources of the entire cluster to accelerate different operations within image provisioning. In a nutshell, Poby executes the fine-grained operations of image provisioning atop the RNICs, on-NIC CPU, hardware accelerators, and host CPU. In addition, Poby assumes that images are compressed in gzip format, which is the most widely used format, accounting for as much as 96.3% of all images in Docker Hub [85]. The design of Poby is orthogonal to the compression format and compatible with other formats.

Disaggregated architecture (§4). Poby devises a *disaggregated architecture* to offload different operations of image provisioning to appropriate places (**C1 addressed**). Firstly,

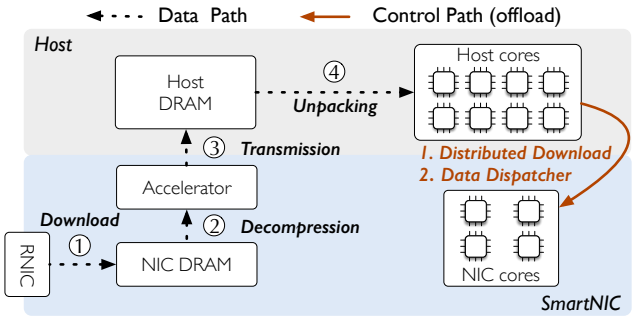


Figure 5: Disaggregated architecture of Poby.

we offload the control of the entire image provisioning to the on-NIC CPU, whose computing overhead is acceptable for its wimpy CPU. Secondly, we download the image via RDMA to fully utilize the high-bandwidth transmission capability of modern SmartNICs. Thirdly, we decompress the downloaded images at the hardware decompression accelerator to achieve better performance. Finally, we transmit the decompressed images from the NIC to the host via PCIe and unpack them by the host CPU, which eliminates the performance bottleneck of transmitting numerous small files.

Pipeline-based data-driven workflow (§5). Poby adopts a *pipeline-based data-driven workflow* to eliminate the waiting delay introduced by the classic sequential execution of coldstart (**C2 addressed**). To enable the pipeline-based data path, images are downloaded, decompressed, transmitted, and unpacked at the granularity of blocks (e.g., 16 MB) in Poby. Thus, the download, decompression, transmission, and unpacking for different data blocks can be executed at the same time, improving the efficiency of image provisioning. However, the finer-grained data path introduces more control information to manage all pipeline blocks. For example, a 37-layer image with a total size of 2.45G can be divided into 189 pipeline blocks (16MB). Poby further utilizes data-driven workflow to reduce the overhead of additional control information. Specifically, we integrate the control command into data blocks. Then, each component in Poby executes the subsequent operations based on the control command in its received data block. This minimizes the transmission of control command and execution status.

Distributed image download (§6). Poby decentralizes the image registry to alleviate the performance bottlenecks of the conventional centralized registry (**C3 addressed**). Specifically, Poby adopts a *best-effort* strategy to currently download the image from multiple nodes who happen to own the required images. In this way, Poby avoids the significant storage overhead of distributed nodes *actively caching images*. To enable this, we first devise the *image metadata index* (IMI) to hold the available image locations and node transmission workload. Poby makes the download plan based on the IMI to ensure the load balance between distributed nodes. Here, our best-effort strategy allows Poby to download images from the original registry when no node contains the required image or the retrieved IMI is inconsistent.

Discussion. The design of Poby, based on workflow accelerating, is orthogonal to most existing studies as it focuses on optimizing the inherent processes of coldstart. For example, the disaggregated architecture and pipeline-based data-driven workflow can accelerate *warm start* and subsequent processes of *fast download*. Even for *on-demand download* works that adopt new image formats, Poby can still accelerate image processing for the lazy download process.

4 Disaggregated Architecture

Poby proposes a disaggregated architecture that orchestrates resources of the entire cluster to accelerate image provisioning for coldstart. Figure 5 illustrates the details of the architecture, composed of a *data path* (§4.1) and a *control path* (§4.2).

4.1 Data Path

The key idea behind Poby data path is to put different operations into the right hardware components to maximize performance gains. Existing container solutions are typically monolithic, with a single server responsible for the entire image provisioning process. In contrast, Poby disaggregates this monolithic architecture into separate components and offloads them to specialized hardware units, including on-NIC CPU, decompression accelerator, and host CPU. Furthermore, we decouple the image extraction into *image decompression* and *image unpacking*. Along with the image transmission between the NIC and host memory (discussed in **C2**), the entire data path of image provisioning currently includes four operations: image download, image decompression, image unpacking, and image transmission. Here, this decoupling between image decompression and image unpacking brings three benefits.

① **New execution order bypassing bottleneck.** The decoupling enables reordering the execution of image unpacking and image transmission. Specifically, Poby executes the data path in the following order. Firstly, Poby downloads the images via RDMA to its SmartNIC memory. Secondly, Poby decompresses the downloaded images using the hardware accelerator. Thirdly, Poby directly transmits the decompressed images from the accelerator to the host memory. Finally, Poby unpacks the decompressed images on the host CPU.

Note that this execution order bypasses the performance bottleneck discussed in **C1**. On the one hand, image unpacking on the host can avoid the overhead of unpacking on the weak on-NIC CPU. On the other hand, transmission for a large decompressed file is much more friendly to PCIe than numerous small unpacked files. Because each file writing operation involves multiple PCIe transactions (such as DMA transfers and interrupt requests). When unpacking a file with numerous small files, writing these files to disk can saturate

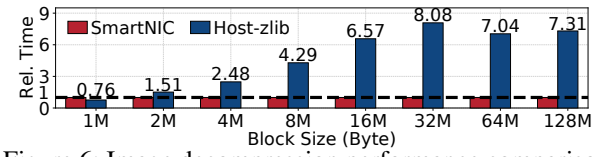


Figure 6: Image decompression performance comparison.

the PCIe bus with a large number of transactions and small data, thus reducing the effective bandwidth utilization.

② **Network-wide collaborative enhancement.** This decoupling permits the architecture to be further disaggregated across multiple nodes. Therefore, we can perform image decompression at the hardware accelerator of remote hosts and then download the decompressed images via RDMA. Here, our on-NIC controller can distinguish whether the received image data is raw or decompressed according to block_status (detail in § 5.2). It then dispatches the data to the local hardware accelerator or directly to the host memory depending on the data type. This collaborative architecture can be applied to the following two scenarios: (1) the local hardware accelerator is busy or occupied by other tasks; and (2) nodes in the cluster are only equipped with RDMA NICs but no SmartNICs. This approach is in line with our design principle, i.e., orchestrating hardware resources of the entire cluster to accelerate image provisioning. As a result, this further reduces the demands of `Poby` for SmartNIC and increases the utilization of accelerators.

Although decompressing the image first greatly increases the amount of data transferred, this is acceptable for two reasons. Firstly, current RDMA hardware offers up to 200 Gbps of transmission bandwidth with μs-scale latency, which significantly reduces the transmission latency of images. Secondly, the hardware accelerator decompresses images much faster than the host (Figure 6), reducing the additional overhead introduced by transmitting decompressed images.

③ **Performance benefits for decompression.** The image decompression itself can benefit from the hardware decompression accelerator of existing SmartNICs. Figure 6 compares the time that decompresses images of different sizes using the hardware accelerator and the classic `zlib` [47] on the host. As the block size increases, the decompression time of `zlib` increases significantly, far exceeding that of the accelerator. Furthermore, for blocks larger than 16 MB, the accelerator reduces the decompression latency by 7× on average. Note that the decompression time of blocks varies depending on the data pattern, leading to fluctuations (e.g., 32 MB).

4.2 Control Path

The control path of `Poby` coordinates the execution workflow of image provisioning. Specifically, the control path includes an *application programming interface (API) agent* running on the host and an *on-NIC controller*.

API agent. The API agent ensures compatibility with the original interface of the existing container service (e.g., Docker API). When the host issues a command to perform a coldstart, our API agent sends this request to the on-NIC controller. When the image provisioning ends, the agent returns the control back to the container service to launch the container.

On-NIC controller. The on-NIC controller is responsible for two functionalities. (1) It manages the distributed image download, including the maintenance of RDMA contexts, the retrieval for IMIs, and the decision-making (i.e., download layer from which host) for distributed downloads (see §6). The on-NIC controller adopts a lazy destruction strategy to reuse existing RDMA connections, minimizing the overhead of frequent connection creation and teardown. (2) It dispatches different types of downloaded image data to the corresponding places in the data path (see §4.1).

The offloaded on-NIC controller brings the following three benefits. Firstly, it decreases host resource usage, allowing hosts to run more business applications. Secondly, the on-NIC controller further reduces the transmission delay of commands for the control path. Finally, the operations of the control path are not compute-intensive, offloading them to the weak on-NIC CPU will not cause overload.

5 Pipeline-based Data-driven Workflow

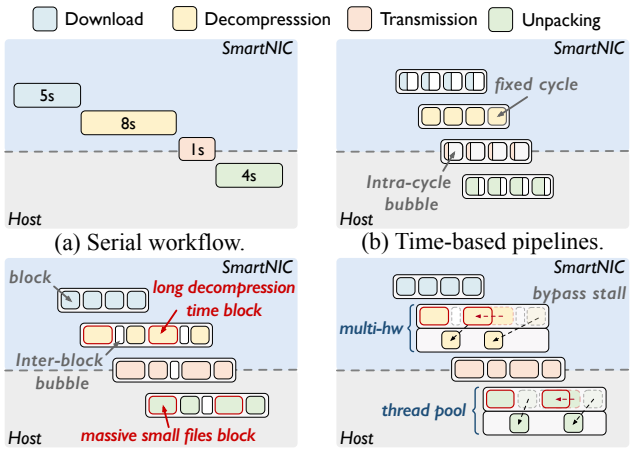
`Poby` introduces a pipeline-based data-driven workflow to enhance the efficiency of classical serial execution (Figure 7a). This section investigates two critical questions: (i) What is the best way to split the pipeline? (§5.1) and (ii) How should the pipeline execution be scheduled? (§5.2).

5.1 Pipeline-based Data Path

Strawman solutions. `Poby` decomposes the original data path into four operations: download, decompression, transmission, and unpacking. These operations support streaming processing, where the partial output of one operation can serve as the input for the next. Thus, `Poby` executes them in a pipeline-based manner to enhance parallelism. However, common pipeline designs face significant performance issues when applied to image provisioning.

- *Layer-based pipelines:* Common in prior studies [23, 55, 72], they parallelize image provisioning at the layer granularity. The small number of image layers and the substantial size variations between them result in overly coarse granularity, limiting parallelism.
- *Time-based pipelines:* This approach segments processing into fixed-length time periods (Figure 7b). Variability in hardware performance introduces pipeline bubbles, as some stages must wait for others to finish. For example, the 100 Gbps RDMA wastes most of each cycle waiting for the 30 Gbps decompression accelerator.

Block-based redundant pipelines. `Poby` proposes block-based redundant hardware pipelines to enhance pipeline par-



(a) Serial workflow. (b) Time-based pipelines.

(c) Block-based pipelines. (d) Redundant pipelines (Poby).

Figure 7: Different designs of pipeline-based workflow.

allelism and address pipeline bubbles. Poby divides images into fixed-size data *blocks*, which are simple subdivisions of layers, requiring no additional processing. This enables parallel processing of blocks within a layer, tackling the low parallelism of layer-based pipelines. However, this approach still faces two significant issues.

Issue 1: Pipeline bubbles. Block-based pipelines still encounter pipeline bubbles. The primary factors contributing to these bubbles (Figure 7c) are:

- Differences in compression ratios cause varying sizes of decompressed blocks, leading to inconsistent decompression times. Figure 8 shows that the decompression time can vary by over 3.5× for 128 MB blocks from different images.
- Data blocks with massive small files take considerably longer to unpack, increasing processing time compared to blocks with fewer files.

To alleviate these bubbles, Poby constructs *redundant* hardware pipelines. Figure 7d illustrates the redundant hardware mechanism. For the decompression stage, Poby prioritizes the round-robin strategy to schedule blocks to local accelerators. When no local accelerators are available, Poby notifies the sender via the existing connection to decompress blocks before transmitting. For the unpacking stage, Poby maintains a dedicated unpacking thread pool on the host side and exploits the kernel scheduler to manage it. If a new block arrives while the previous one is still being unpacked, the new block will be scheduled to available cores.

Issue 2: Block size. Block-based pipelines face the classic pipeline cycle decision dilemma.

- A small block size wastes most of the time switching and setting environments, reducing pipeline efficiency. Figure 8 shows that the accelerator incurs a 6.9 ms startup time.
- A large block size reduces the parallelism of pipelines or even degenerates into layer-based pipelines.

To achieve high efficiency, Poby adopts an empirical value, 16 MB, as the block size. Firstly, a 16 MB block can amortize the startup time of decompression (Figure 8), which we speculate is related to hardware configuration and the minimum data processing granularity of the accelerator. Secondly, over 80% of the small layers are less than 16 MB [85], a 16MB block strikes a balance between the pipelining of large layers and the switching overhead of small blocks. In addition, the efficiency of the 16 MB block is also confirmed by *Exp#5*.

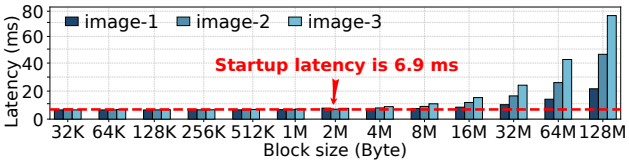


Figure 8: The decompression latency of SmartNIC with different block sizes and images.

5.2 Data-driven Workflow

Issues with control-driven workflows. The block-based pipeline introduces a significant amount of control information into the conventional *control-driven workflow*. Figure 9 illustrates such a workflow, where the centralized on-NIC controller coordinates the execution of each decomposed image provisioning operation. The controller uses control messages to interact with hardware components, determining the start, completion, and downstream of operations. In block-based pipelines, images are split into numerous blocks, each requiring control information for orchestration. This consumes substantial resources and limits the concurrency of pipelines.

Data-driven workflow in Poby. Poby proposes a *data-driven workflow* to schedule the execution of block-based pipelines, eliminating the overhead of control messages. Instead of relying on centralized control, Poby integrates state information into data blocks and dynamically updates the information during execution. Each block contains two pieces of state information. (i) The block_index specifies the position of a block in the layer and image, along with the total number of blocks in the corresponding layer. (ii) The block_status indicates whether the block is compressed or not.

The complete pipeline-based data-driven workflow of Poby is as follows:

(1) The API agent parses the requests from users and sends them to the on-NIC controller.

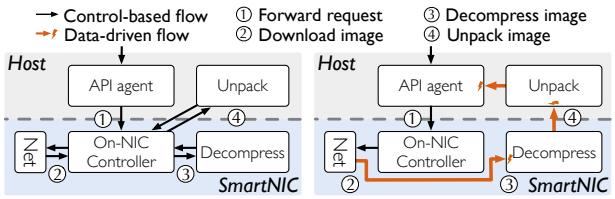
(2) The on-NIC controller retrieves the locations of the image and establishes network connections to get image blocks.

(3) The download operation receives the block and assigns it based on block_status to (i) available decompression queues (round-robin strategy) and (ii) transmission queue.

(4) The accelerator decompresses the blocks in its queue and assigns them to the transmission queue when completed.

(5) The transfer engine sends the blocks in its task queue to the host side and notifies the unpacking thread pool.

(6) The unpacking thread pool assigns worker threads to process blocks, updates progress and completion to the API



(a) Control-driven workflow. (b) Data-driven workflow (Poby).

Figure 9: Different designs of workflow models.

agent according to the `block_status`.

(7) All operations perform as above until its queue is empty.

This data-driven workflow of Poby provides two major advantages. (i) The pipeline becomes self-driven, removing dependency on the on-NIC controller. In this workflow, hardware components track block statuses and pass them to subsequent stages upon completion. As shown in Figure 9b, the pipeline autonomously operates without requiring control-path intervention after initialization. (ii) Each block is uniquely identified, enabling precise tracking of provisioning. Firstly, Poby leverages `block_index` to assess provisioning progress. For example, when the final block of a layer is processed, the corresponding image step is considered complete. Secondly, using the `block_status`, Poby can bypass the decompression accelerator by sending remotely decompressed blocks (see §4.1) directly to the host for unpacking.

6 Distributed Image Download

The distributed image download of Poby is transparent to the host and incurs no additional overhead. Existing distributed image download studies [7, 39, 72, 76, 86] effectively address the performance bottlenecks of centralized registries. However, these studies introduce significant synchronization and storage overhead to actively cache images. As noted in [85], the total size of all publicly accessible images already exceeded 47 TB in 2017, and this number has undoubtedly grown larger since then. Go beyond existing works, Poby proposes *best-effort distributed download* and *image metadata index* to minimize cache overhead. Furthermore, Poby offloads distributed image management to SmartNIC, freeing the host from concerns about the management overhead.

Best-effort distributed download. The best-effort design only allows coldstart requests to download images from nodes that happen to contain the required image. Note that images on other nodes are from previous runs, rather than proactive caching. Only when no node contains the required image, Poby downloads it from the registry. Additionally, our distributed download operates at the image layer granularity, coldstart can download different layers from other nodes concurrently. The best-effort strategy offers two benefits: (1) It avoids the significant storage overhead of proactively caching. (2) It reduces the network bandwidth consumption needed for frequent synchronization of image updates across nodes. Note that Poby does not introduce additional coldstart latency,

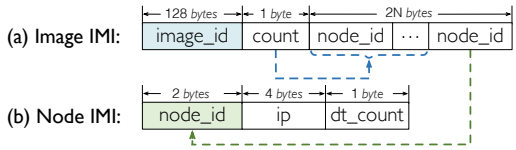


Figure 10: Format of IMI.

since it retains the original centralized image registry download scheme as a fallback. Once any error occurs during the processing of a layer, Poby immediately hands over the task to the original process without blocking the execution flow.

Image metadata index (IMI). To enable best-effort distributed image download, we propose the image metadata index (IMI). On the one hand, the IMIs track the available locations of each image and the image transmission workload of each node. Here, we use the number of active image download tasks on each node to quantify the transmission workload. On the other hand, the IMIs associate the image digest with the image manifest to obtain all necessary image metadata (e.g., the number of image layers). Leveraging the IMI, the node initiating the coldstart can arrange the most efficient download plan by considering image locations, the number of layers, and each node's workload. In short, it prioritizes downloading layers from nodes with fewer download tasks.

IMI format. As shown in Figure 10, we devise two data structures to hold the image and node information, respectively.

- *Image_IMI.* The key of the Image_IMI is the image digest `image_id`, which uniquely identifies images. According to the OCI Image Format Specification [27], we use the SHA-512 algorithm to generate the 128-byte digest. The Image_IMI value contains the available image locations referenced by the `node_id`. Each Image_IMI holds up to *N* `node_id` slots, where *N* is a pre-configured constant. We use the `count` field to track the number of valid `node_id`.
- *Node_IMI.* Each node in Poby corresponds to a Node_IMI, identified by the `node_Id` field. The Node_IMI values include the node IP address (`ip`) and the current number of active download tasks on this node (`dt_count`). Each time the IMI service responds to an IMI query, it increments the `dt_count` in the associated Node_IMIs by one to indicate a new download task. When the download task is over, the node initiating the coldstart must notify the IMI service to decrement the `dt_count` in the corresponding Node_IMIs.

It is trivial to keep all IMIs in memory. Consider a system consisting of 50,000 nodes and 1,000,000 images. An image_IMI records require 139 bytes if N is set to 5. Thus, it will take 0.35 MB for Node_IMIs and 139 MB for Image_IMIs in total.

IMI management. The IMI service manages all the IMI data, including IMI querying, node load tracking, and image location tracking. (1) For IMI querying, the IMI service returns the manifest and IMI of the required image to the node initiating the coldstart. (2) For node load tracking, the IMI service automatically increments the `dt_count` in the Node_IMIs when responding to an IMI query, indicating a new download

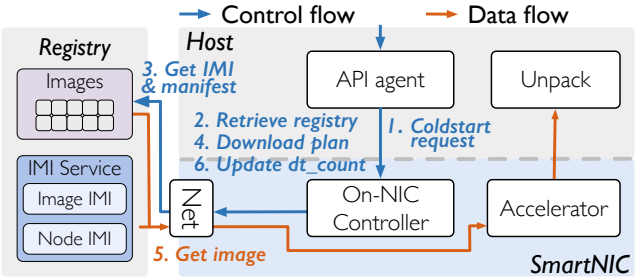


Figure 11: Workflow of distributed image download.

task on these related nodes. Upon receiving a completion notice from a node, it updates the `dt_count` accordingly. (3) For image location tracking, the IMI service simultaneously adds the node to the available locations in the Image_IMI. If the number of available nodes exceeds N, we use the least recently used algorithm to update the `node_id` list in Image_IMI. If a node deletes a local image, it notifies the IMI service to remove its `node_id` from the corresponding Image_IMI. Here, our IMI service is colocated with the original registry, which just provides the required images when unavailable on any node (best-effort strategy). Since the IMI is far smaller than images, this prevents the IMI service from becoming a bottleneck in transmission and managing IMI.

Workflow. Figure 11 shows the complete workflow of the distributed image download. (1) The on-NIC controller receives the coldstart request from the API agent on the host. (2) The on-NIC controller sends the request for the desired image (image name and tag) to the registry. (3) The registry looks up the image digest based on the provided name and tag. Then, it retrieves the manifest and IMI using the image digest and returns them as a response. (4) The on-NIC controller makes the download plan based on Algorithm 1. (5) The pipeline-based data path downloads data blocks within each image layer from the planned nodes. (6) The on-NIC controller notifies the IMI service on the registry of the completion of the download task to decrease the `dt_count` of the corresponding Node_IMIs.

Algorithm 1 presents the pseudo-code to generate the most efficient distributed download plan. The algorithm assigns a provider for each layer according to the response (manifest and IMIs) from the IMI service. It achieves load balancing through the following steps: (1) Sort nodes containing the images in ascending order according to dt_count (line 2). (2) Assign the node with the lowest image download load to the current layer (lines 5-6). (3) Switch to the next node when the load of the current node is greater than the next node (lines 7-10). (4) Repeat *steps 2-3* to assign nodes for all layers (lines 4-10).

Discussion. We explore how `Poby` handles unexpected cases. Firstly, `Poby` can function even with inconsistent IMI. If a node deletes a local image and this change is not synchronized with the IMI service, inconsistencies may arise. In this case, the on-NIC controller verifies with planned nodes during

Algorithm 1 Download plan for distributed image download

Input: *manifest, image_imi, node_imi[list_len]*
Output: *download_plan<layer_id, ip>*

```
1:  function GENERATE_DOWNLOAD_PLAN( )
2:      nodes = sort_by_dtcount_ascend(node_imi)
3:      node_id = 0
4:      for all layer in manifest.layers do
5:          download_plan[layer.id] = node_imi[node_id].ip
6:          node_imi[node_id].dt_count ++
7:          if node_id == list_len − 1 then
8:              node_id = 0
9:          else if nodes[node_id].dt_count > nodes[node_id + 1]
    then
10:             node_id ++
11:     return download_plan
```

RDMA connection setup that they have the required image. If a node is missing the image, the controller updates the download plan or pulls the image from the registry if none of the nodes have it. Secondly, `Poby` can handle unexpected image deletions. If a host deletes the image during the transmission, and the NIC remains unaware, corrupted data may be transmitted. `Poby` checks the checksum for each downloaded layer, and if a mismatch is detected, the controller identifies that node as faulty and re-downloads the layer from another node.

7 Implementation

We implement a prototype of `Poby` with NVIDIA BlueField-2 SmartNIC [14] and open-source it*. We implement the disaggregated architecture with cross-hardware pipelines and a data-driven framework. These two works are mainly deployed on SmartNIC. For the IMI management, we integrate with existing components.

SmartNIC. BlueField-2 consists of three parts: (i) RDMA NIC, (ii) multi-core SoC, and (iii) hardware accelerators. According to the specifications, the total maximum power consumption of BlueField-2 is 150W. Considering the low-power characteristics of ARM cores and dedicated hardware accelerators, its power efficiency exceeds that of general-purpose host CPUs. Besides, the hardware vendor provides a software library called DOCA to support interactions with these hardware components. We encapsulate the library to build the disaggregated architecture of `Poby`.

Cross-hardware pipelines. The pipeline of `Poby` consists of two parts: data storage and processing. We implement the two parts with memory pool and thread pool.

- *Memory pool.* `Poby` pipelines the data path at block granularity. We implement a memory pool to manage blocks. It supports adjusting the size and number of blocks dynamically. `Poby` configures them according to image characteristics. The image characteristics include layer size and the

*Available at: https://github.com/ACS-Innov/ATC2025-Poby.

Table 1: Workload images adopted from benchmark suites.

	Name	Specification	Source
Micro Service	Social Network (*SN*)	9 layers, 90.6MB	DeathStar Bench [24]
	Media Service (*MS*)	23 layers, 286.6MB	
	Hotel Reservation (*HR, small*)	18 layers, 39.6MB	
FaaS	Python (*PY*)	30 layers, 155.1MB	OpenWhisk [22]
	Node.js (*JS*)	21 layers, 356.3MB	
	Golang (*GO, medium*)	20 layers, 281.6MB	
	PHP (*PP*)	37 layers, 192.7MB	
	pyaes (PS)	33 layers, 169.9MB	Function Bench [30]
	linpack (*LP*)	33 layers, 214.6MB	
	chameleon (*CL*)	33 layers, 170.1MB	
	matmul (*MT*)	33 layers, 214.6MB	
	ML Model (*ML, large*)	37 layers, 2.45GB	

number of layers. Poby gets them from the image manifest, which is the initial download in the entire workflow.

- *Thread pool.* To improve the efficiency of pipelines, Poby concurrently performs time-consuming steps such as downloading and unpacking. Specifically, Poby implements a thread pool for unpacking on the host with folly [20], a high-performance C++ library that provides a thread-pool API. The workers in the thread pool process different layers concurrently. Downloads also employ layer-grained concurrency. The scheduling of all pipeline threads is managed by the scheduler of Linux.

Data-driven workflow. We implement an RDMA-based remote procedure call (RPC) framework for data-driven workflow. The workflow uses two-sided RDMA verbs to support gRPC-like communication, and epoll to monitor multiple events occurring in pipelines (e.g., RDMA events, compression hardware events). To transmit data blocks with control messages, we place control messages into RPC parameters.

IMI management. The IMI service is a centralized component for managing IMI data and supporting retrieval. To ensure availability, we integrate it into the registry server and use leveldb [25] to store the IMIs. For the coldstart node, we implement the IMI logic into the on-NIC controller. When the on-NIC controller receives requests to coldstart or deletes images, it notifies the update of images to IMI service.

8 Evaluation

We evaluate the performance benefits of Poby with various images, and aim to answer the following questions.

- **Q1 (End-to-end performance):** How much can Poby speed up coldstart (§8.2)?
- **Q2 (Concurrency and scalability):** How does the latency of Poby change as the number of nodes and concurrent requests increase (§8.3)?
- **Q3 (Latency breakdown):** How much do the individual processes of Poby contribute to overall latency? (§8.4)?
- **Q4 (Resource demand):** What is the resource demand of Poby (§8.5)?
- **Q5 (Architecture optimization):** Is it possible to disaggregate Poby to different nodes (§8.6)?

8.1 Experimental Methodology

Testbed. We deploy our Poby prototype on two servers. Each server is equipped with two Intel Xeon Gold 6226R CPUs, 256 GB of memory, and an NVIDIA BlueField-2 SmartNIC. These servers run Ubuntu 22.04 LTS with the 5.15.0 Linux kernel. In addition, we use another server with a dual-port 100Gbps NVIDIA ConnectX-5 NIC [15] to run the image registry and IMI service.

Baselines. We compare Poby with two baselines.

- *containerd* [23] is the runtime of Docker, the most popular open-source containerization platform. It is an industry-standard container runtime and supports managing the life cycle of containers.
- *iSulad* [55] is a new container solution that has the advantage of being light, fast, and applicable to multiple architectures. The performance of iSulad is much better than containerd for pulling images, including download, decompression, and unpacking.

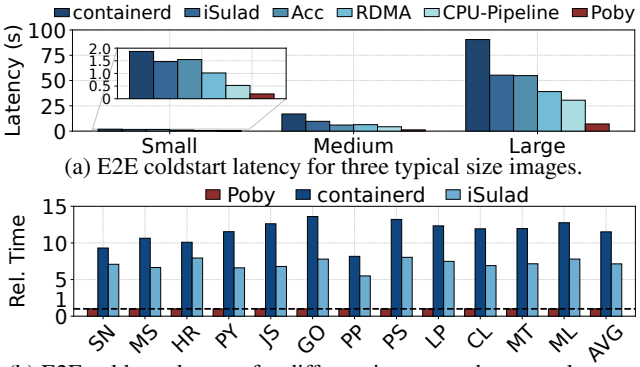
Workloads. We use a collection of images from real-world applications and two cloud benchmark suites as workloads for evaluation. Firstly, we use DeathStarBench [24] to represent microservice images. Secondly, we select Functionbench [30] images and four widely used language runtime images from OpenWhisk [22] as FaaS workloads. Thirdly, we choose three images of typical sizes from those images for detailed analysis. They are hotel-reservation (*small*), OpenWhisk-Golang (*medium*), and ML Model (*large*), with compressed sizes of 39.6 MB, 281.6 MB, and 2.45 GB, respectively. Table 1 displays all the information of workloads in detail.

Latency metrics. In this work, we measure the end-to-end coldstart latency to evaluate the performance of different systems. Poby focuses on image provisioning and all evaluated works adopt the same container runtime. Hence, the end-to-end latency does not take the boot time of coldstart into account, as the boot time is the same across all systems.

8.2 End-to-end Performance

This experiment evaluates the performance of Poby with different images. We choose latency as a key performance metric and configure Poby with 2 unpacking threads (see §8.4). These images, which cover small, medium, and large sizes, can effectively illustrate the performance of Poby for different sizes of images.

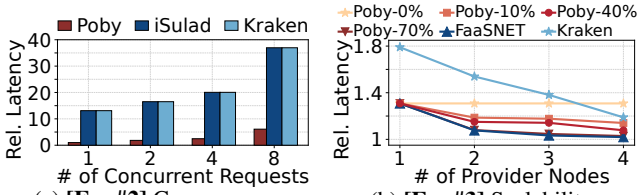
[Exp#1] Performance improvement: Poby can speedup coldstart by 11.5× and 7.1× than containerd and iSulad on average. Figure 12a shows the coldstart latency of the three typical-size images on different systems. Poby reduces the latency of coldstart by up to 92.7%, and it outperforms two



(a) E2E coldstart latency for three typical size images.

(b) E2E coldstart latency for different images and average latency.

Figure 12: **[Exp#1]** E2E coldstart latency.



(a) **[Exp#2]** Concurrency. (b) **[Exp#3]** Scalability.

Figure 13: Latency of Poby and baselines with different (a) concurrent numbers and (b) provider nodes. Poby-N% represents Poby with a N% distributed download hit ratio.

industry-standard container platforms, containerd and iSulad, with maximum speedups of 13.2× and 8.0×, respectively. Containerd takes more than 90 seconds for large images due to serial execution, while iSulad still takes 55.3 seconds with multi-threads. To further evaluate the acceleration of Poby for various containers, we compare the latency of containerd, iSulad, and Poby for all the images (Table 1) in Figure 12b. Poby is 11.5× and 7.1× faster than containerd and iSulad on average. The values of containerd and iSulad are relative latencies to Poby.

To detail the performance improvements of our design compared to the original host CPU solution, we implemented two prototypes with RDMA (RDMA) and hardware accelerator (Acc) based on the original workflow. Furthermore, we implement a host CPU-based pipeline prototype (CPU-Pipeline) based on the RDMA version to demonstrate the performance gains from pipelines. Figure 12a shows that the direct utilization of hardware features (Acc and RDMA) only reduces coldstart latency by 20% compared to the host CPU solution, while pipeline (CPU-Pipeline) further reduces it by 34%. However, they still have a significant gap compared to Poby. Because the disaggregated pipelines of Poby efficiently leverage RDMA and accelerators on SmartNIC while addressing the issue of pipeline bubbles and small file transfer.

8.3 Concurrency and Scalability

We evaluate the concurrency and scalability of Poby and compare them with two baselines.

- *Kraken* [72] is a P2P-powered registry designed for image managementand distribution in cloud environment. Kraken has been in production at Uber, which can distribute 20K 100MB-1G blobs in 30 seconds.
- *FaaSNET* [76] is a SOTA fast download work. Due to the lack of open-source access and unavailable details of the new image format, we simulate a simplified version without on-demand fetching using the distributed image download part in its paper.

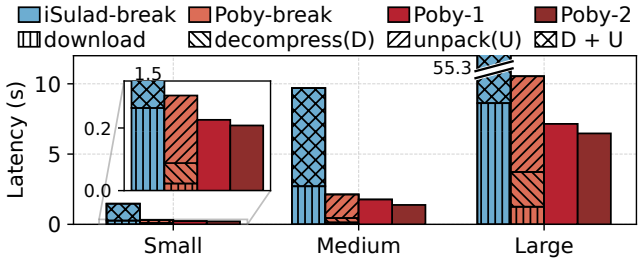
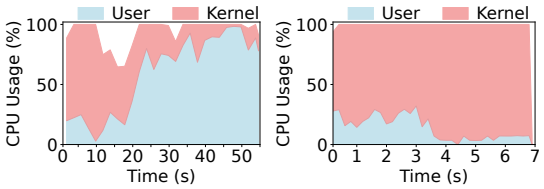


Figure 14: **[Exp#4]** Latency and statistics of each process for iSulad and Poby. The overall latency includes unpacking with one (Poby-1) and two (Poby-2) threads. Poby-break represents the time accumulated by each step of the poby-1.

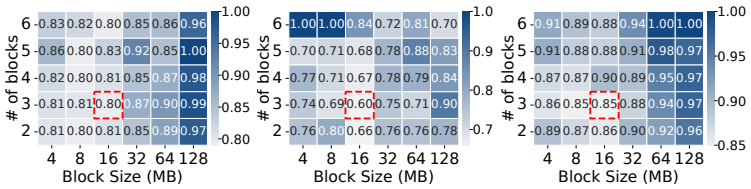
[Exp#2] Concurrency: ***Poby is 4.8× faster than Kraken under 8 concurrent coldstart requests.*** We simultaneously generate coldstart requests on the same machine to evaluate the concurrency performance of Poby and iSulad. Figure 13a shows the average relative latency of Poby, iSulad, and Kraken under different numbers of concurrent requests. All latencies in the figure are relative to the latency of Poby under one request. In this experiment, Kraken gets the same latency as iSulad. Since in the scenarios of concurrent coldstart multiple images, Kraken cannot take advantage of P2P download. Meanwhile, Poby remarkably outperforms iSulad and Kraken from one to eight concurrent requests. As the number of concurrent connections increases to eight, the latency of Poby increases by 6×. Because the network bandwidth limits the performance of disaggregated pipelines of Poby. However, the latency of Poby is still 4.8× faster than that of iSulad and Kraken. It is worth noting that we do not limit the CPU resource (up to 96 cores) of iSulad. Because network bandwidth and the non-parallelizable portion (e.g., each layer) limit concurrency and infinite time reduction.

[Exp#3] Scalability: ***Poby performs much better than Kraken at a distributed download rate of only 10%, and achieves comparable scalability to FaaSNET at 70%.*** This experiment evaluates the scalability of Poby with our RDMA cluster, which consists of 5 nodes. We compare the average coldstart latency of Poby, FaaSNET, and Kraken. In each experiment, all 5 nodes simultaneously initiate one coldstart request, with the number of image providers ranging from 1 to 4. For example, the N providers experiment of Poby indicates that 5 nodes simultaneously initiate coldstart request once through Poby to download the image from N providers. Since Poby employs a best-effort distributed download strategy, with the hit rate balancing coldstart latency and storage overhead. This hit ratio represents the probability that Poby



(a) iSulad. (b) Poby.

Figure 15: **[Exp#5]** CPU usage of iSulad and Poby for coldstarting the large image.



(a) Small. (b) Medium. (c) Large.

Figure 16: **[Exp#6]** Latency of Poby for small, medium, and large images with different sizes and numbers of blocks.

downloads the image from multiple nodes in one experiment. We measured the latency of Poby under different distributed download hit rates (e.g., Poby-10%). Figure 13b shows the latency of these systems in different cases. The values in this figure are the relative values of the corresponding system and images compared to Exp#1. The latency of Poby-0% shows that 5 concurrent requests exceeded the capacity of one provider, resulting in a 30% increase in latency. The decentralized image download alleviates image provision bottlenecks, and even a single additional provider can improve coldstart performance by 22%.

8.4 Latency Breakdown

We conduct latency analysis to quantify the execution time of each step for iSulad and Poby. Since Poby adopts pipelines to accelerate the data path, we cannot directly obtain the time consumption of each step. Here, we accumulate the time to process blocks for each step as the total time (Poby-break), though the time of these steps overlaps. Note that the decompression time of Poby consists of two parts: decompression on SmartNIC and data transmission to the host. This is because Poby integrates these two steps into a single SmartNIC operation to improve performance. iSulad does not require cross-hardware transmission, but the decompression and unpacking operations are integrated (e.g., libarchive). Therefore, we measure the total time of the two operations (iSulad-break).

[Exp#4] Latency analysis: ***Poby can reduce the extraction time by 76.1%, and adding an additional unpacking thread can further reduce it by 11.0%.*** Figure 14 shows the latency breakdown for the typical-size images, where image extraction is the most time-consuming step for both iSulad and Poby. Poby reduces the extraction time by 76.1% than iSulad on average. This indicates Poby achieves an efficient architecture on the SmartNIC (§5). In addition, the unpacking process of Poby retained on the host still occupies an average of 71.6% of the total time. To further evaluate Poby with concurrent unpacking (storage backend support), we measure the latency of Poby with one (Poby-1) and two (Poby-2) unpacking threads. The 2-thread unpacking reduces the latency by 9.1% on average. Even for a small image, which is only 39.6 MB, it still reduces the latency by 4.4%. While for medium and large images, 2-thread reduces the latency by more than 11.0%. Besides, we measure Poby with more unpacking threads, but it does not gain further improvement. Because 2-threads unpacking already reached the bandwidth bottleneck of storage.

8.5 Resource Demand

In the experiments, we evaluate the CPU and memory resource demand of Poby. The software and hardware environment for this experiment is the same as § 8.2.

[Exp#5] CPU usage: ***Poby offloads 45.3% of user-mode CPU usage on the host side to SmartNIC, and reduces CPU usage by 87.5%.*** Poby is designed to minimize the kernel-mode execution. Specifically, Poby reduces the overall host-side CPU workload by offloading intensive user-mode tasks, such as decompression, to the SmartNIC. As a result, only the non-offloadable file-writing operations remain in kernel mode. For CPU usage, we measure the CPU utilization of Poby and iSulad on the host side. Figure 15a and 15b details the CPU usage of iSulad and Poby in kernel and user mode. Poby takes 6.8s for coldstart of the large image while iSulad takes 54.6s. Poby reduces 47.8s (87.5%) of CPU time than iSulad. The low utilization of iSulad from 14s to 19s is due to waiting for a large layer (> 1 GB) to finish downloading. Besides, Poby spends only 14.1% of CPU time in user mode, while iSulad spends 59.4%. It offloads 45.3% of user-mode tasks to SmartNIC, including control path and decompression. This indicates that Poby offloads most tasks to SmartNIC (except for unpacking), leaving only 5% of user-space CPU usage on the host side.

Additionally, offloading image provisioning to the SmartNIC similarly encounters performance interference issues. Poby delegates the time- and resource-consuming decompression to the hardware accelerator, leaving the on-NIC CPU solely responsible for the control path. We evaluate the on-NIC CPU demands of Poby during the coldstart of all workload images, observing a peak on-NIC CPU usage of only 16.3% and an average usage as low as 3.0%. Given the low resource demand, we believe it feasible to reserve resources on the on-NIC CPU for Poby, effectively mitigating potential interference from other applications.

[Exp#6] Memory configuration: ***Poby achieves the lowest coldstart latency when configured with three 16MB blocks.*** Poby maintains a memory pool on the host side, which includes several pre-allocated blocks for receiving the decompressed data. The size of the blocks in the memory pool is

consistent with the block-based pipelines. To obtain an optimal memory pool configuration, we measure the coldstart latency of Poby with different sizes and numbers of blocks. As we mentioned in §5, blocks larger than 2 MB can overcome the startup latency of accelerators on SmartNIC. Here, we vary the block size from 4 MB to 128 MB and the number of blocks in the memory pool from 2 to 6. Figure 16 illustrates the latency of coldstart for small, medium, and large images with different memory configurations. The following two observations confirm the analysis in § 5.1 and guide our configuration of the block size. For the small image, it aligns better with smaller block sizes. Because a single small block is sufficient to accommodate layers of the small image. Increasing the block size and number does not improve performance and may even lead to performance degradation due to worse data locality (blocks larger than the cache). For the medium and large images, 16 MB is an efficient block size for pipelined execution. As for the number, three blocks are sufficient for 2 unpacking threads, and increasing the number of blocks brings no additional benefit.

8.6 Architecture optimization

As mentioned in §4, Poby can alleviate local SmartNIC contention via network-wide offload. This experiment evaluates the performance of disaggregating Poby to different nodes.

[Exp#7] Network-wide offload: ***Poby achieves network-wide decompression offload with only 18.0% latency increase.*** This experiment configures one node as the coldstart requester and the other node as the decompression assistant. The requester sends image requests to the assistant for decompressing the image. Figure 17 compares the coldstart latency of local and remote decompression for Poby. It reveals that disaggregating Poby to remote nodes with RDMA is feasible for SmartNIC contention scenarios. The remote decompression only introduces 18.0% latency overhead compared to local decompression. Moreover, although the average compression ratio of images is as high as 2.7, the remote decompression only results in 2.2 Gbps network bandwidth overhead. Because Poby employs a block-based pipeline, the capability to decompress and unpack limits the excessive use of network bandwidth. As a result, the network bandwidth overhead of remote decompression is amortized across the entire coldstart process, preventing a surge in network bandwidth demand.

9 Related Work

Lightweight isolation. Some efforts propose lightweight isolation mechanisms to accelerate the container startup with coldstart [2,9,34,46,66]. We divide these works into two categories. Firstly, some systems [2,34] employ sensitive data protection techniques to share isolation environments. Secondly, other approaches utilize fine-grained isolation techniques (e.g., application- [46], language- [66], and thread- [34] level).

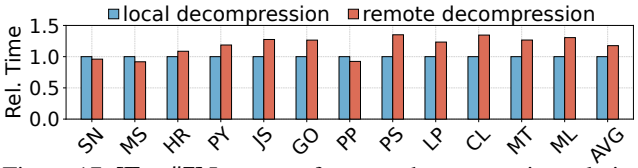


Figure 17: **[Exp#7]** Latency of remote decompression relative to local decompression for Poby.

These methods completely update the existing isolation mechanism of cloud platforms, effectively mitigating the coldstart issue. Poby optimizes the pre-processing steps of creating isolation environments, which is orthogonal to lightweight optimization works and can be integrated together.

SmartNIC offload. Previous works [70,79] characterize the performance of SmartNIC. Due to the integration of RDMA network cards, extensive researches continue on leveraging SmartNIC to enhance network performance. However, with the introduction of programmable hardware, most of the researches [4,21,32,37,43,52,56–59,65] focus more on offloading tasks such as network functions, network protocol stacks, and package processing to SmartNIC. In addition, some works attempt to offload a broader range of applications. AlNiCo [40] accelerates transaction processing within a single node, while Xenix [63] applies to distributed scenarios. Similarly, LineFS [31] offloads distributed file systems with pipeline parallelism. Moreover, other studies apply SmartNIC to storage [36,50,57,69,84], microservice [12,44,45], remote function call [35,67], and others [18,26,28,42,71,77]. Specifically, Poby offloads FaaS coldstart to SmartNIC and uses on-NIC CPU to manage RDMA links and hardware acceleration to decompress layers.

10 Conclusion

Existing approaches ignore the most time-consuming step of coldstart. We complete the coldstart puzzle with SmartNIC offloading and data-driven pipelines. We propose Poby, a software-hardware collaborative system that leverages emerging hardware SmartNICs to accelerate image provisioning directly. Poby disaggregates the architecture of coldstart and orchestrates appropriate hardware resources of the entire cluster to accelerate different operations within image provisioning. It achieves efficiency with data-driven pipelines. Our experiment with BlueField-2 shows that Poby performs well with different benchmark images.

Acknowledgments

We sincerely thank our shepherd Sibin Mohan and anonymous reviewers for their valuable comments. This work is supported in part by the Strategic Priority Research Program of Chinese Academy of Sciences under grant number XDA0320000 and XDA0320300, the National Natural Science Foundation of China (Grant No. 62090022, U24B6012 and 62172388).

References

[1] Siddharth Agarwal, Maria A. Rodriguez, and Rajkumar Buyya. A reinforcement learning approach to reduce serverless function cold start frequency. In 2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid), pages 797–803, 2021.

[2] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. SAND: Towards High-Performance serverless computing. In 2018 USENIX Annual Technical Conference (USENIX ATC 18), pages 923–935, Boston, MA, July 2018. USENIX Association.

[3] Lixiang Ao, George Porter, and Geoffrey M. Voelker. Faasnap: Faas made fast using snapshot-based vms. In Proceedings of the Seventeenth European Conference on Computer Systems, EuroSys '22, page 730–746, New York, NY, USA, 2022. Association for Computing Machinery.

[4] Mina Tahmasbi Arashloo, Alexey Lavrov, Manya Ghobadi, Jennifer Rexford, David Walker, and David Wentzlaff. Enabling programmable transport protocols in High-Speed NICs. In 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20), pages 93–109, Santa Clara, CA, February 2020. USENIX Association.

[5] Ioana Baldini, Paul Castro, Kerry Chang, Perry Cheng, Stephen Fink, Vatche Ishakian, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, Aleksander Slominski, et al. Serverless computing: Current trends and open problems. Research advances in cloud computing, pages 1–20, 2017.

[6] Rohan Basu Roy, Tirthak Patel, Rohan Garg, and Devesh Tiwari. Codecrunch: Improving serverless performance via function compression and cost-aware warmup location optimization. In Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1, ASPLOS '24, page 85–101, New York, NY, USA, 2024. Association for Computing Machinery.

[7] Soeren Becker, Florian Schmidt, and Odej Kao. Edgepier: P2p-based container image distribution in edge computing environments. In 2021 IEEE International Performance, Computing, and Communications Conference (IPCCC), pages 1–8. IEEE, 2021.

[8] Vivek M Bhasi, Jashwant Raj Gunasekaran, Prashanth Thinakaran, Cyan Subhra Mishra, Mahmut Taylan Kandemir, and Chita Das. Kraken: Adaptive container provisioning for deploying dynamic dags in serverless platforms. In Proceedings of the ACM Symposium on Cloud Computing, pages 153–167, 2021.

[9] Sol Boucher, Anuj Kalia, David G. Andersen, and Michael Kaminsky. Putting the "micro" back in microservice. In 2018 USENIX Annual Technical Conference (USENIX ATC 18), pages 645–650, Boston, MA, July 2018. USENIX Association.

[10] Marc Brooker, Mike Danilov, Chris Greenwood, and Phil Piwonka. On-demand container loading in AWS lambda. In 2023 USENIX Annual Technical Conference (USENIX ATC 23), pages 315–328, Boston, MA, July 2023. USENIX Association.

[11] James Cadden, Thomas Unger, Yara Awad, Han Dong, Orran Krieger, and Jonathan Appavoo. Seuss: Skip redundant paths to make serverless fast. In Proceedings of the Fifteenth European Conference on Computer Systems, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery.

[12] Sean Choi, Muhammad Shahbaz, Balaji Prabhakar, and Mendel Rosenblum. λ- nic: Interactive serverless compute on smartnics. In Proceedings of the ACM SIGCOMM 2019 Conference Posters and Demos, SIGCOMM Posters and Demos '19, page 151–152, New York, NY, USA, 2019. Association for Computing Machinery.

[13] Alibaba Cloud. Faq about function execution. https://www.alibabacloud.com/help/en/fc/support/is-the-runtime-environment-released-after-a-function-returns-a-response-can-i-reuse-the-cached-state-or-resources-from-a-previous-invocation, 2020.

[14] NVIDIA Corporation. Nvidia bluefield-2 dpu: Data center infrastructure on a chip. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/documents/datasheet-nvidia-bluefield-2-dpu.pdf, 2021.

[15] NVIDIA Corporation. Nvidia connectx-5 infiniband adapter cards. https://nvdam.widen.net/s/pkxbnmbgkh/networking-infiniband-datasheet-connectx-5-2069273, 2021.

[16] Advanced Micro Devices. Alveo sn1000 smartnic accelerator card. https://www.xilinx.com/products/boards-and-kits/alveo/sn1000.html, 2024.

[17] ds2 lab. The function cold start traces from alibaba cloud function compute. https://github.com/ds2-lab/FaaSNet?tab=readme-ov-file#dataset-downloading, 2021.

[18] Dong Du, Qingyuan Liu, Xueqiang Jiang, Yubin Xia, Binyu Zang, and Haibo Chen. Serverless computing on heterogeneous computers. In Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '22, page 797–813, New York, NY, USA, 2022. Association for Computing Machinery.

[19] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting. In Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '20, page 467–481, New York, NY, USA, 2020. Association for Computing Machinery.

[20] Facebook. Folly: Facebook open-source library. https://github.com/facebook/folly, 2024.

[21] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, Harish Kumar Chandrappa, Somesh Chaturmohta, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, and Albert Greenberg. Azure accelerated networking: SmartNICs in the public cloud. In 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18), pages 51–66, Renton, WA, April 2018. USENIX Association.

[22] Apache Software Foundation. Openwhisk: Warmed containers. https://github.com/apache/openwhisk/blob/master/docs/warmed-containers.md, 2024.

[23] Cloud Native Computing Foundation. containerd. https://github.com/containerd/containerd, 2024.

[24] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, et al. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, pages 3–18, 2019.

[25] Google. Leveldb. https://github.com/google/leveldb, 2023.

[26] Stewart Grant, Anil Yelam, Maxwell Bland, and Alex C. Snoeren. Smartnic performance isolation with fairnic: Programmable networking for the cloud. In Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication, SIGCOMM '20, page 681–693, New York, NY, USA, 2020. Association for Computing Machinery.

[27] Open Container Initiative. Oci image format specification. https://github.com/opencontainers/image-spec?tab=readme-ov-file, 2024.

[28] Houxiang Ji, Mark Mansi, Yan Sun, Yifan Yuan, Jinghan Huang, Reese Kuper, Michael M. Swift, and Nam Sung Kim. STYX: Exploiting SmartNIC capability to reduce datacenter memory tax. In 2023 USENIX Annual Technical Conference (USENIX ATC 23), pages 619–633, Boston, MA, July 2023. USENIX Association.

[29] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. Profiling a warehouse-scale computer. In Proceedings of the 42nd Annual International Symposium on Computer Architecture, ISCA '15, page 158–169, New York, NY, USA, 2015. Association for Computing Machinery.

[30] Jeongchul Kim and Kyungyong Lee. Functionbench: A suite of workloads for serverless cloud function service. In 2019 IEEE 12th International Conference on Cloud Computing (CLOUD), pages 502–504, 2019.

[31] Jongyul Kim, Insu Jang, Waleed Reda, Jaeseong Im, Marco Canini, Dejan Kostić, Youngjin Kwon, Simon Peter, and Emmett Witchel. Linefs: Efficient smartnic offload of a distributed file system with pipeline parallelism. In Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles, SOSP '21, page 756–771, New York, NY, USA, 2021. Association for Computing Machinery.

[32] Taehyun Kim, Deondre Martin Ng, Junzhi Gong, Youngjin Kwon, Minlan Yu, and KyoungSoo Park. Rearchitecting the TCP stack for I/O-Offloaded content delivery. In 20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23), pages 275–292, Boston, MA, April 2023. USENIX Association.

[33] Sumer Kohli, Shreyas Kharbanda, Rodrigo Bruno, Joao Carreira, and Pedro Fonseca. Pronghorn: Effective checkpoint orchestration for serverless hot-starts. In Proceedings of the Nineteenth European Conference on Computer Systems, EuroSys '24, page 298–316, New York, NY, USA, 2024. Association for Computing Machinery.

[34] Swaroop Kotni, Ajay Nayak, Vinod Ganapathy, and Arkaprava Basu. Faastlane: Accelerating Function-as-a-Service workflows. In 2021 USENIX Annual Technical Conference (USENIX ATC 21), pages 805–820. USENIX Association, July 2021.

[35] Nikita Lazarev, Shaojie Xiang, Neil Adit, Zhiru Zhang, and Christina Delimitrou. Dagger: Efficient and fast rpcs in cloud microservices with near-memory reconfigurable nics. In Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '21, page 36–51, New York, NY, USA, 2021. Association for Computing Machinery.

[36] Bojie Li, Zhenyuan Ruan, Wencong Xiao, Yuanwei Lu, Yongqiang Xiong, Andrew Putnam, Enhong Chen, and Lintao Zhang. Kv-direct: High-performance in-memory key-value store with programmable nic. In Proceedings of the 26th Symposium on Operating Systems Principles, SOSP '17, page 137–152, New York, NY, USA, 2017. Association for Computing Machinery.

[37] Bojie Li, Kun Tan, Layong (Larry) Luo, Yanqing Peng, Renqian Luo, Ningyi Xu, Yongqiang Xiong, Peng Cheng, and Enhong Chen. Clicknp: Highly flexible and high performance network processing with reconfigurable hardware. In Proceedings of the 2016 ACM SIGCOMM Conference, SIGCOMM '16, page 1–14, New York, NY, USA, 2016. Association for Computing Machinery.

[38] Huaicheng Li, Mingzhe Hao, Stanko Novakovic, Vaibhav Gogte, Sriram Govindan, Dan R. K. Ports, Irene Zhang, Ricardo Bianchini, Haryadi S. Gunawi, and Anirudh Badam. Leapio: Efficient and portable virtual nvme storage on arm socs. In Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '20, page 591–605, New York, NY, USA, 2020. Association for Computing Machinery.

[39] Huiba Li, Yifan Yuan, Rui Du, Kai Ma, Lanzheng Liu, and Windsor Hsu. DADI: Block-Level image service for agile and elastic application deployment. In 2020 USENIX Annual Technical Conference (USENIX ATC 20), pages 727–740. USENIX Association, July 2020.

[40] Junru Li, Youyou Lu, Qing Wang, Jiazhen Lin, Zhe Yang, and Jiwu Shu. AlNiCo: SmartNIC-accelerated contention-aware request scheduling for transaction processing. In 2022 USENIX Annual Technical Conference (USENIX ATC 22), pages 951–966, Carlsbad, CA, July 2022. USENIX Association.

[41] Zijun Li, Linsong Guo, Quan Chen, Jiagan Cheng, Chuhao Xu, Deze Zeng, Zhuo Song, Tao Ma, Yong Yang, Chao Li, and Minyi Guo. Help rather than recycle: Alleviating cold startup in serverless computing through Inter-Function container sharing. In 2022 USENIX Annual Technical Conference (USENIX ATC 22), pages 69–84, Carlsbad, CA, July 2022. USENIX Association.

[42] Jiaxin Lin, Adney Cardoza, Tarannum Khan, Yeonju Ro, Brent E. Stephens, Hassan Wassel, and Aditya Akella. RingLeader: Efficiently offloading Intra-Server orchestration to NICs. In 20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23), pages 1293–1308, Boston, MA, April 2023. USENIX Association.

[43] Jiaxin Lin, Kiran Patel, Brent E. Stephens, Anirudh Sivaraman, and Aditya Akella. PANIC: A High-Performance programmable NIC for multi-tenant networks. In 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20), pages 243–259. USENIX Association, November 2020.

[44] Ming Liu, Tianyi Cui, Henry Schuh, Arvind Krishnamurthy, Simon Peter, and Karan Gupta. Offloading distributed applications onto smartnics using ipipe. In Proceedings of the ACM Special Interest Group on Data Communication, SIGCOMM '19, page 318–333, New York, NY, USA, 2019. Association for Computing Machinery.

[45] Ming Liu, Simon Peter, Arvind Krishnamurthy, and Phitchaya Mangpo Phothilimthana. E3: Energy-Efficient microservices on SmartNIC-Accelerated servers. In 2019 USENIX Annual Technical Conference (USENIX ATC 19), pages 363–378, Renton, WA, July 2019. USENIX Association.

[46] Xuanzhe Liu, Jinfeng Wen, Zhenpeng Chen, Ding Li, Junkai Chen, Yi Liu, Haoyu Wang, and Xin Jin. Faaslight: general application-level cold-start latency optimization for function-as-a-service in serverless computing. ACM Transactions on Software Engineering and Methodology, 2023.

[47] Jean loup Gailly and Mark Adler. zlib: A massively spiffy yet delicately unobtrusive compression library. https://www.zlib.net/, 2024.

[48] Marvell. Marvell liquidio iii. https://www.xilinx.com/products/boards-and-kits/alveo/sn1000.html, 2020.

[49] Microsoft. Azure functions hosting options. https://learn.microsoft.com/en-us/azure/azure-functions/functions-scale#cold-start-behavior, 2023.

[50] Jaehong Min, Ming Liu, Tapan Chugh, Chenxingyu Zhao, Andrew Wei, In Hwan Doh, and Arvind Krishnamurthy. Gimbal: Enabling multi-tenant storage disaggregation on smartnic jbofs. In Proceedings of the 2021 ACM SIGCOMM 2021 Conference, SIGCOMM '21, page 106–122, New York, NY, USA, 2021. Association for Computing Machinery.

[51] Anup Mohan, Harshad Sane, Kshitij Doshi, Saikrishna Edupuganti, Naren Nayak, and Vadim Sukhomlinov. Agile cold starts for scalable serverless. In 11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19), Renton, WA, July 2019. USENIX Association.

[52] YoungGyoun Moon, SeungEon Lee, Muhammad Asim Jamshed, and KyoungSoo Park. AccelTCP: Accelerating network applications with stateful TCP offloading. In 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20), pages 77–92, Santa Clara, CA, February 2020. USENIX Association.

[53] Netronome. Agilio cx smartnics. https://netronome.com/agilio-smartnics/, 2024.

[54] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. SOCK: Rapid task provisioning with Serverless-Optimized containers. In 2018 USENIX Annual Technical Conference (USENIX ATC 18), pages 57–70, Boston, MA, July 2018. USENIX Association.

[55] openEuler. isulad. https://gitee.com/openeuler/iSulad, 2024.

[56] Phitchaya Mangpo Phothilimthana, Ming Liu, Antoine Kaufmann, Simon Peter, Rastislav Bodik, and Thomas Anderson. Floem: A programming system for NIC-Accelerated network applications. In 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18), pages 663–679, Carlsbad, CA, October 2018. USENIX Association.

[57] Boris Pismenny, Liran Liss, Adam Morrison, and Dan Tsafrir. The benefits of general-purpose on-nic memory. In Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '22, page 1130–1147, New York, NY, USA, 2022. Association for Computing Machinery.

[58] Salvatore Pontarelli, Roberto Bifulco, Marco Bonola, Carmelo Cascone, Marco Spaziani, Valerio Bruschi, Davide Sanvito, Giuseppe Siracusano, Antonio Capone, Michio Honda, Felipe Huici, and Giuseppe Siracusano. FlowBlaze: Stateful packet processing in hardware. In 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19), pages 531–548, Boston, MA, February 2019. USENIX Association.

[59] Yiming Qiu, Jiarong Xing, Kuo-Feng Hsu, Qiao Kang, Ming Liu, Srinivas Narayana, and Ang Chen. Automated smartnic offloading insights for network functions. In Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles, SOSP '21, page 772–787, New York, NY, USA, 2021. Association for Computing Machinery.

[60] Francisco Romero, Gohar Irfan Chaudhry, Íñigo Goiri, Pragna Gopa, Paul Batum, Neeraja J. Yadwadkar, Rodrigo Fonseca, Christos Kozyrakis, and Ricardo Bianchini. Faa$t: A transparent auto-scaling cache for serverless applications. In Proceedings of the ACM Symposium on Cloud Computing, SoCC '21, page 122–137, New York, NY, USA, 2021. Association for Computing Machinery.

[61] Rohan Basu Roy, Tirthak Patel, and Devesh Tiwari. Icebreaker: Warming serverless functions better with heterogeneity. In Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, pages 753–767, 2022.

[62] Divyanshu Saxena, Tao Ji, Arjun Singhvi, Junaid Khalid, and Aditya Akella. Memory deduplication for serverless computing with medes. In Proceedings of the Seventeenth European Conference on Computer Systems, EuroSys '22, page 714–729, New York, NY, USA, 2022. Association for Computing Machinery.

[63] Henry N. Schuh, Weihao Liang, Ming Liu, Jacob Nelson, and Arvind Krishnamurthy. Xenic: Smartnic-accelerated distributed transactions. In Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles, SOSP '21, page 740–755, New York, NY, USA, 2021. Association for Computing Machinery.

[64] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In 2020 USENIX Annual Technical Conference (USENIX ATC 20), pages 205–218. USENIX Association, July 2020.

[65] Rajath Shashidhara, Tim Stamler, Antoine Kaufmann, and Simon Peter. FlexTOE: Flexible TCP offload with Fine-Grained parallelism. In 19th USENIX Symposium on Networked Systems Design and Implementation

(NSDI 22), pages 87–102, Renton, WA, April 2022. USENIX Association.

[66] Simon Shillaker and Peter Pietzuch. Faasm: Lightweight isolation for efficient stateful serverless computing. In 2020 USENIX Annual Technical Conference (USENIX ATC 20), pages 419–433. USENIX Association, July 2020.

[67] David Sidler, Zeke Wang, Monica Chiosa, Amit Kulkarni, and Gustavo Alonso. Strom: Smart remote memory. In Proceedings of the Fifteenth European Conference on Computer Systems, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery.

[68] Arjun Singhvi, Arjun Balasubramanian, Kevin Houck, Mohammed Danish Shaikh, Shivaram Venkataraman, and Aditya Akella. Atoll: A scalable low-latency serverless platform. In Proceedings of the ACM Symposium on Cloud Computing, pages 138–152, 2021.

[69] Shangyi Sun, Rui Zhang, Ming Yan, and Jie Wu. Skv: A smartnic-offloaded distributed key-value store. In 2022 IEEE International Conference on Cluster Computing (CLUSTER), pages 1–11, 2022.

[70] Lasse Thostrup, Daniel Failing, Tobius Ziegler, and Carsten Binnig. A dbms-centric evaluation of bluefield dpus on fast networks. In Rajesh Bordawekar and Tirthankar Lahiri, editors, International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures, ADMS@VLDB 2022, Sydney, Australia, September 5, 2022, pages 1–10, 2022.

[71] Maroun Tork, Lina Maudlej, and Mark Silberstein. Lynx: A smartnic-driven accelerator-centric architecture for network servers. In Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '20, page 117–131, New York, NY, USA, 2020. Association for Computing Machinery.

[72] Uber. Kraken: A p2p-powered docker registry that focuses on scalability and availability. https://github.com/uber/kraken, 2024.

[73] Dmitrii Ustiugov, Plamen Petrov, Marios Kogias, Edouard Bugnion, and Boris Grot. Benchmarking, analysis, and optimization of serverless function snapshots. In Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '21, page 559–572, New York, NY, USA, 2021. Association for Computing Machinery.

[74] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at google with borg. In Proceedings of the Tenth European Conference on Computer Systems, EuroSys '15, New York, NY, USA, 2015. Association for Computing Machinery.

[75] Tim Wagner. Understanding container reuse in aws lambda. https://aws.amazon.com/cn/blogs/compute/container-reuse-in-lambda/, 2014.

[76] Ao Wang, Shuai Chang, Huangshi Tian, Hongqi Wang, Haoran Yang, Huiba Li, Rui Du, and Yue Cheng. Faasnet: Scalable and fast provisioning of custom serverless container runtimes at alibaba cloud function compute. In 2021 USENIX Annual Technical Conference (USENIX ATC 21), pages 443–457. USENIX Association, July 2021.

[77] Zeke Wang, Hongjing Huang, Jie Zhang, Fei Wu, and Gustavo Alonso. FpgaNIC: An FPGA-based versatile 100gb SmartNIC for GPUs. In 2022 USENIX Annual Technical Conference (USENIX ATC 22), pages 967–986, Carlsbad, CA, July 2022. USENIX Association.

[78] Xingda Wei, Rongxin Cheng, Yuhan Yang, Rong Chen, and Haibo Chen. Characterizing off-path SmartNIC for accelerating distributed systems. In 17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23), pages 987–1004, Boston, MA, July 2023. USENIX Association.

[79] Xingda Wei, Rongxin Cheng, Yuhan Yang, Rong Chen, and Haibo Chen. Characterizing off-path SmartNIC for accelerating distributed systems. In 17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23), pages 987–1004, Boston, MA, July 2023. USENIX Association.

[80] Xingda Wei, Fangming Lu, Tianxia Wang, Jinyu Gu, Yuhan Yang, Rong Chen, and Haibo Chen. No provisioned concurrency: Fast RDMA-codesigned remote fork for serverless computing. In 17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23), pages 497–517, Boston, MA, July 2023. USENIX Association.

[81] Yanan Yang, Laiping Zhao, Yiming Li, Shihao Wu, Yuechan Hao, Yuchi Ma, and Keqiu Li. Flame: A centralized cache controller for serverless computing. In Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4, ASPLOS '23, page 153–168, New York, NY, USA, 2024. Association for Computing Machinery.

[82] Hanfei Yu, Rohan Basu Roy, Christian Fontenot, Devesh Tiwari, Jian Li, Hong Zhang, Hao Wang, and Seung-Jong Park. Rainbowcake: Mitigating cold-starts in serverless with layer-wise container caching and sharing. In Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1, ASPLOS '24, page 335–350, New York, NY, USA, 2024. Association for Computing Machinery.

[83] Tianyi Yu, Qingyuan Liu, Dong Du, Yubin Xia, Binyu Zang, Ziqian Lu, Pingchao Yang, Chenggang Qin, and Haibo Chen. Characterizing serverless platforms with serverlessbench. In Proceedings of the 11th ACM Symposium on Cloud Computing, pages 30–44, 2020.

[84] Jie Zhang, Hongjing Huang, Lingjun Zhu, Shu Ma, Dazhong Rong, Yijun Hou, Mo Sun, Chaojie Gu, Peng Cheng, Chao Shi, and Zeke Wang. Smartds: Middle-tier-centric smartnic enabling application-aware message split for disaggregated block storage. In Proceedings of the 50th Annual International Symposium on Computer Architecture, ISCA '23, New York, NY, USA, 2023. Association for Computing Machinery.

[85] Nannan Zhao, Vasily Tarasov, Hadeel Albahar, Ali Anwar, Lukas Rupprecht, Dimitrios Skourtis, Amit S. Warke, Mohamed Mohamed, and Ali R. Butt. Large-scale analysis of the docker hub dataset. In 2019 IEEE International Conference on Cluster Computing (CLUSTER), pages 1–10, 2019.

[86] Chao Zheng, Lukas Rupprecht, Vasily Tarasov, Douglas Thain, Mohamed Mohamed, Dimitrios Skourtis, Amit S Warke, and Dean Hildebrand. Wharf: Sharing docker images in a distributed file system. In Proceedings of the ACM Symposium on Cloud Computing, pages 174–185, 2018.

Burst Computing: Quick, Sudden, Massively Parallel Processing on Serverless Resources

Daniel Barcelona-Pons[†‡] Aitor Arjona[†] Pedro García-López[†]
Enrique Molina-Giménez[†] Stepan Klymonchuk[†]
[†]*Universitat Rovira i Virgili* [‡]*Barcelona Supercomputing Center*

Abstract

We present burst computing, a novel serverless solution tailored for burst-parallel jobs. Unlike Function-as-a-Service (FaaS), burst computing establishes job-level isolation using a novel group invocation primitive to launch large groups of workers with guaranteed simultaneity. Resource allocation is optimized by packing workers into fewer containers, which accelerates their initialization and enables locality. Locality significantly reduces remote communication compared to FaaS and, combined with simultaneity, it allows workers to communicate synchronously with message passing and group collectives. Consequently, applications unfeasible in FaaS are now possible. We implement burst computing atop OpenWhisk and provide a communication middleware that seamlessly leverages locality with zero-copy messaging. Evaluation shows reduced job invocation and communication latency for a 2× speed-up in TeraSort and a 98.5% reduction in remote communication in PageRank (13× speed-up) compared to standard FaaS.

1 Introduction

The cloud offers compute infrastructure on demand, but provisioning, adjusting, and managing these resources for large-scale data processing applications is an arduous task, especially for non-experts. Furthermore, when the load is unpredictable, dynamic, with varying volumes of data, user-driven, and sometimes interactive, finding the right scale to avoid misprovisioning [25, 39, 54] becomes very complex.

Function-as-a-Service (FaaS) has gained traction as a solution to the resource provisioning problem as it offers rapid, on-demand, no-ops scaling and a pay-as-you-go billing model at very fine granularity (MB per ms). Users do not need to set up a cluster, but the service simply accepts function invocations and fully manages the rest. Moreover, its resource *burstability* has set FaaS aside from traditional engines like Spark or Dask, allowing to start thousands of short-lived functions in seconds instead of minutes (see Table 1). Several research works [15, 16, 23, 44] have used FaaS for a myriad of data- and compute-intensive tasks.

Table 1: Time to provision cloud compute resources on different services and technologies.

Technology	Total vCPUs	Nodes[a]	Start-up time
EMR Spark	96	6	296 s
		24	431 s
Dataproc	96	6	95 s
		24	113 s
Dask	128	8	184 s
		64	253 s
Ray	128	8	187 s
		64	229 s
Knative (Kubernetes)	960	960	54 s
OpenWhisk	960	960	21 s
AWS Lambda (2 GiB)	960	960	6 s
Burst Computing	960	960	1.7 s

[a] AWS EMR Spark and GCP Dataproc use m5 and E2-standard VM families, respectively. Dask and Ray are deployed on user-managed m6i family EC2 VMs. Knative, OpenWhisk, and Burst deployed on 20 c7i.12xlalrge VMs but start 960 functions/workers.

This has brought a new concept in cloud computing that refers to the ability to quickly respond to sudden, parallel workloads without provisioning a cluster in advance. Fouladi et al. [16] talk about a "burstable supercomputer-on-demand" and a "burst-parallel swarm of thousands of cloud functions, all working on the same job." However, literature admits that the current FaaS model is too narrow and precluding for massively parallel data processing programs (MPP) [24].

In this work, we highlight that the key issue of FaaS hindering burst-parallel jobs is its lack of group awareness. Indeed, FaaS users need multiple independent service calls to spawn a fleet of workers, which become strongly isolated from each other. We note that such fine-grained isolation is damaging and unnecessary for collaborative jobs, and thus propose to raise the multi-tenant boundaries to the job level.

We present *burst computing*, a new cloud computing model to deal with quick, sudden, massively parallel workloads, which we call *bursts*. To this end, we offer a *group invocation* primitive to handle the whole job as a unit. To the best of our knowledge, we are the first to implement this feature

in a FaaS platform, clearly differing from all other research efforts that suffer the burden of handling and orchestrating individual function invocations. A group invocation allows to optimize resource allocation, ensure worker parallelism, and perform *packing*: running multiple workers co-located in the same environment. In addition to speed up worker start-up latency, this enables *worker locality* and *simultaneity*, which can be exploited to improve code and data loading, and to aid powerful worker-to-worker communication patterns (e.g., broadcast, all-to-all) that seamlessly leverage shared memory channels with zero-copy mechanisms.

From the user side, a burst spawns a fleet of workers that communicate with message-passing, a simple but very powerful abstraction that creates a novel serverless substrate versatile to many applications beyond what is feasible in FaaS. This gives extensive control of the job to advanced users and allows to design compute engines or frameworks on top (e.g., DAG-based) to simplify development, manage the execution, and handle failures. Massively parallel computations are prime burst applications, especially when sudden, unpredictable, and user-driven in nature. Batch-like jobs are also good candidates when run interactively. Some examples are data processing, analytics, and machine learning workloads like exploratory model tuning, SQL, *k*-means, and large-scale sorting. Bursts may be stateless (e.g., grid search or Monte Carlo simulations) or stateful (e.g., table joins and aggregations).

We make the following contributions:

- We present burst computing, a novel cloud service model for short, sudden, massively parallel jobs (bursts). We believe that no cloud vendor or research effort has created the necessary substrate to support them.
- Burst computing evolves FaaS with a key novel group invocation primitive (a flare) that raises multi-tenant isolation from a single function invocation to the whole job. In consequence, the system launches massive process groups faster, with guaranteed parallelism, and packs workers together to exploit locality.
- We implement a burst computing platform by extending OpenWhisk, a state-of-the-art FaaS system. Our implementation includes a specialized Rust worker runtime and a burst communication middleware that seamlessly leverage worker locality with collective code/data loading and zero-copy messaging.
- Under evaluation on several burst-parallel workloads against FaaS, burst computing improves job invocation latency (up to 11.5× faster), worker simultaneity (up to 26.5× lower median absolute deviation), and group communication (up to 98% in a broadcast), for a speed-up of 13× in PageRank and 2× in TeraSort.

2 Motivation: in search of burstability

Many works are leveraging serverless services for massive data processing [3, 7, 9, 15, 16, 23, 40, 60] despite current (FaaS) hindrances [5, 18, 24] due to the resource *burstability*

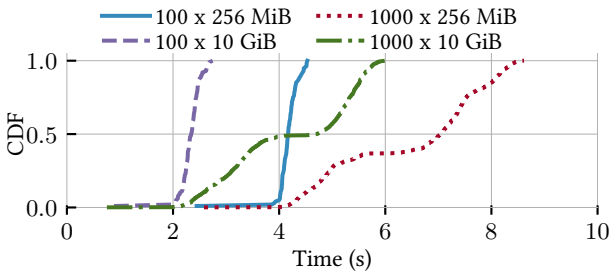


Figure 1: Start-up time (cold start) of 100 and 1000 FaaS functions in AWS Lambda for two memory sizes.

of this model [17, 38]. Applications benefit from quick, on-demand, no-ops resources at very fine granularity, and pay precisely for what they need, when they need it.

This has brought what we call *bursts*, massively parallel processing (MPP) workloads that appear suddenly and process large, variable volumes of data in a very short time (under 1 or 2 minutes). Such applications have dynamic resource needs that cannot be predicted easily, thus serverless burstability becomes essential [49, 50, 52, 56]. Consider, e.g., an interactive, scientist-driven workflow in a Jupyter Notebook, where the user dynamically explores large datasets and modifies parameters that significantly impact the workload size. Although they may resemble batch jobs, they are characterized by their sudden, sporadic occurrence, highly dynamic and unpredictable data volumes, and the expectation of low-latency execution. Representative use cases include interactive model tuning via grid search, exploratory data analysis with SQL queries and algorithms such as logistic regression, and data preparation operations like filtering and sorting. The MilliSort and MilliQuery benchmarks [31] exemplify such short-running workloads. Additional scenarios include real-time data stream or video feed processing, where both data volume and analytical complexity may fluctuate dramatically over time.

Current data processing solutions such as Spark, Dask, Flink, or Ray fail to support bursts. A long-lived deployment of these engines is impractical, as it would easily become misprovisioned. They are not offered *as a service* by any cloud either, which could palliate the issue by multiplexing jobs from multiple tenants, and thus forces per-user deployments that are too slow to set up [38], even on cloud-managed offerings (e.g., Amazon EMR). Table 1 shows that starting one of these technologies is intolerable for critical sporadic or dynamically sized applications.

In contrast, FaaS services provide a large-scale compute substrate much faster. Fig. 1 shows that AWS Lambda may spawn a fleet of 1000 functions in 6 s; a much more appropriate time range for bursts.[1] FaaS is also more attractive than Container-as-a-Service (CaaS) or managed Kubernetes ser-

[1]Note that small functions (256 MiB) incur higher invocation latency than large ones (10 GiB). Also found on other providers (e.g., GCP), it is likely due to the overhead of scheduling finer-grained resources.

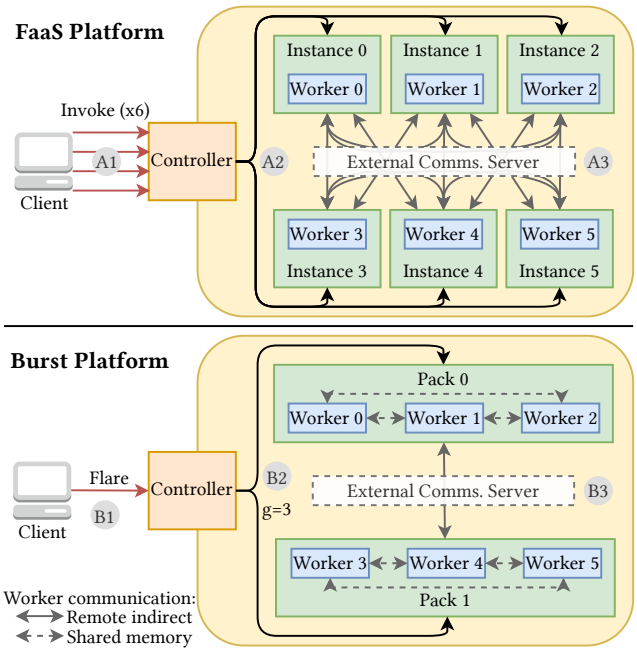


Figure 2: Running a data processing job of 6 workers in FaaS and burst computing with granularity (g) 3.

vices due to simpler abstractions [26] and quicker resource allocation. For example, Knative, a Kubernetes-based FaaS-like implementation, is noticeably slower in spawning workers than dedicated FaaS platforms (Table 1).

2.1 FaaS is holding us back

A review of the literature will show us that running bursts atop FaaS brings many challenges [18, 24, 38]. We highlight three friction points: (**F1**) worker isolation, (**F2**) job fragmentation with complex orchestration, and (**F3**) huge data movement.

To illustrate them, Fig. 2 follows the execution of a parallel job on a FaaS platform. It shows a parallel job with 6 workers. The job could be embarrassingly parallel (stateless) such as a data filtering, or require the workers to coordinate at some point (stateful) such as a table join or, more intensively due to its iterative nature, PageRank.

F1 appears because multi-tenant isolation is at the level of a function invocation. FaaS spawns function instances independently, one at a time, requiring multiple HTTP requests A1 to obtain the 6 workers. Besides the added latency of several requests, this is an issue for parallel jobs because the platform is not aware of these workers being collaborators, and thus cannot guarantee their parallelism. This creates delays or skews between workers that potentially harm job execution. Take, for instance, Fig. 1, where the last function starts up to 6 s after the first one.[2] Even more, the platform populates identical environments (instances) for each invocation A2, which stresses the system with code, dependency, and data[3] loading that creates memory duplication [41, 53].

F2 occurs when workers need to coordinate. For instance, TeraSort *à la* MapReduce includes a data shuffle amidst the job, and PageRank iteratively globally aggregates a vector. Workers cannot communicate effectively because they may not exist at the same time (**F1**). Instead, workers read and write intermediate data asynchronously through an external storage solution. This pattern (depicted in Fig. 3) creates job fragmentation (function stages) and complicates its orchestration, especially in iterative algorithms like PageRank (unfeasible with this approach [6]). First, it increases data movement and requires worker recreation at each stage (adding code and data loading overhead). Second, it needs an active workflow orchestration process to monitor the state of workers and oversee the overall job progress.[4] Centralized solutions add a mostly-idle driver component while decentralized task scheduling adds a layer of complexity to applications [9, 29, 32]. Neither can solve the underlying problem of worker isolation.

These issues are emphasized by **F3**. With so many tiny isolated workers, most communication patterns (e.g., a shuffle) require numerous remote connections A3. In data processing workloads, this may result in very large data transfers, precluded by the (FaaS) lack of direct communication [10, 35].

3 Burst Computing

Burst computing is a novel paradigm for running *bursts* in the cloud. It overcomes the above frictions with two key principles that evolve FaaS: group awareness and locality exploitation. Fig. 2 shows how this changes job invocation.

FaaS hinders worker collaboration because multi-tenant isolation is at the level of a single function (**F1**). Because a job belongs to a single tenant, it makes sense to raise isolation to the job level and handle all its workers as a group. To this end, burst computing provides a group invocation primitive, which we call *flare* B1, to instantly launch massive process groups with guaranteed parallelism. To the best of our knowledge, we are the first to implement this kind of primitive in a serverless system. Flares bring group awareness to the service, which is key to perform worker *packing* B2, i.e., running multiple workers of a job in the same isolated environment. Packing establishes *worker locality* and enables several optimizations discussed below.

In a flare, all workers have guaranteed parallelism and access to the job context (e.g., the burst size, IDs, or locality), which allows them to communicate synchronously in patterns unfeasible in FaaS, such as worker-to-worker message passing and collectives, that simplify job orchestration and avoid **F2**. This difference is depicted in Fig. 3. **F3** is addressed because communication B3 can seamlessly exploit locality and use shared memory mechanisms between workers in the same pack, which reduces remote transfers.

[2] Further evaluation (not shown in the plot) reveals that this dispersity may increase to 44 s in GCP, or 20 s in an OpenWhisk deployment.

[3] For instance, hyperparameter tuning uses the same data in all workers.

[4] This can be painful since FaaS does not provide monitoring mechanisms.

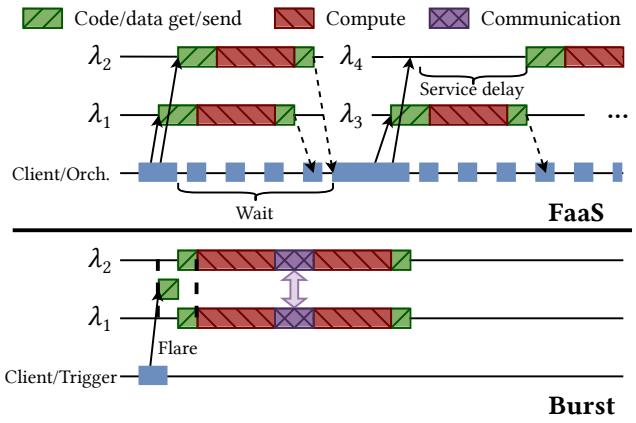


Figure 3: Timeline comparison of a parallel job with FaaS and burst computing.

3.1 Worker packing and communication

Worker packing To run a flare, the burst platform allocates n workers into m packs; we say that n is the *burst size*. The number of workers per pack is the burst's *granularity* ($g = n/m$). Thus, Fig. 2 shows a burst of size $n = 6$ where, by setting $g = 3$, the platform only spawns 2 packs, each with 3 workers. The higher g, the lower m, reducing the number of environment creations, which is a critical part of function invocation time in FaaS. Then, worker code and dependencies are loaded only once per pack and shared by all co-located workers. This further helps with initialization time (especially when dependencies are large) and optimizes resource usage (e.g., avoiding memory duplication [41]). A similar reasoning applies for data loading: workers processing the same data (like in hyperparameter tuning) download it just once per pack and utilize their aggregated resources to speed up the transfer (i.e., with parallel downloads).

Choosing g is a trade-off between ease of system management and locality maximization. To illustrate that, we identify three strategies[5] for worker packing: (i) *heterogeneous*, where workers are placed in containers as big as possible in the underlying system machines; (ii) *homogeneous*, where workers are placed in fixed-size containers; and (iii) *mixed*, where workers are put in fixed-size packs, but if multiple packs fall onto the same machine, they are merged into a single container. The first approach maximizes locality, but it can become a resource scheduling problem, as it is prone to fragmentation. The homogeneous packing mitigates that issue, but it restricts worker locality. The third strategy is the compromise that allows a fast and flexible management while still maximizing locality (see §5). Given this complexity, we argue that the responsibility for setting the granularity should lie with the platform rather than the user, enabling better control over resource scheduling and providing a more streamlined and user-friendly service.

[5] Strategies must consider how many resources we assign to each worker. For simplicity, this paper considers only vCPUs and applies 1 vCPU per worker, but the strategies work for any such assignment.

Worker communication Burst applications are *elastically distributed and collaborative*. They are coded as a single function run by all workers that accepts any worker multiplicity transparently. Then, because workers are guaranteed to be parallel, they may coordinate synchronously by sending messages and with common communication patterns.

To simplify this, burst computing includes a worker-to-worker, message-passing communication middleware readily available to workers. The middleware seamlessly identifies messages between workers placed in the same pack for local communication (zero-copy). Only messages between packs are transferred remotely, and the middleware optimizes these connections (e.g., a broadcast only sends one message per pack). Remote delivery may be implemented with several technologies. Our contributions are independent of this choice because burst computing reduces any remote communication through packing. In this work, we follow the usual approach in FaaS and only consider indirect solutions using an external communication server B3.

4 Design and implementation

We put the above ideas into a prototype burst computing platform and communication middleware. Here we provide the design details and implementation. Fig. 4 shows an overview of the main components and their interactions.

The burst platform extends the design of a FaaS platform to implement group invocation and worker packing. Built atop Apache OpenWhisk, our platform shares its components with important modifications (see §4.4). The *controller* manages user interaction with the platform, it handles inbound HTTP requests to deploy and invoke bursts, oversees system resources, and performs worker packing. A *database* stores the burst definitions and configuration, as well as the results and execution metadata. Computational resources in the platform are provided by the *invokers*, a set of machines with capacity for burst *packs*. Packs are run in containers that isolate a custom runtime environment to run workers.

Our burst communication middleware (BCM) has two main components: the core communication library and the remote backends. The *library* exposes message-based communication to workers, and it is extensible with *backends* to use different remote message delivery solutions.

4.1 Life cycle overview

Fig. 4 depicts the life cycle of the system. To deploy a new burst definition, the user first sends (1) a `deploy` HTTP request. The controller receives it and registers (2) the new definition in the database. Later, when the user desires to trigger the execution of the burst, they send (3) a `flare` HTTP request with specific parameters. The controller handles the invocation and decides worker allocation (4) based on the current state of the invoker machines. The affected invokers

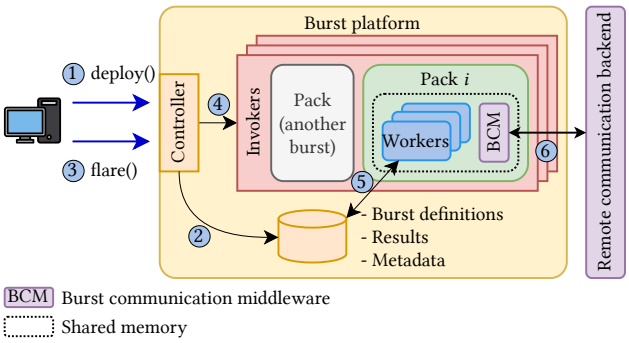


Figure 4: Burst computing platform overview.

receive the task to spawn the required runtime environments (packs) with space for as many workers as needed. When the environments boot, their host invoker tells them which burst definition and parameters to load (5) from the database. Then, each pack spawns its workers internally, which will execute the user-defined function (`work` in Table 2) in parallel. Workers may use the BCM to coordinate and share data. This seamlessly uses shared memory or remote connections to communicate workers (6) in the same or a different pack, respectively. Additionally, workers may read or write data to external storage systems (e.g., object storage) or produce a result that is stored back to the database, where it may be retrieved later by users through another HTTP request.

4.2 Developing and running bursts

User experience is key for burst computing. As a serverless service, all resource management remains hidden. Users interact with the service through a simple interface that allows to define bursts with resource-agnostic code and to schedule their execution. This is similar to FaaS services that allow users to upload their function definitions and then set up triggers or invoke them as needed. The interface and abstractions are summarized in Table 2.

Deployment Similar to functions in FaaS, developers package and upload their burst definitions (code) to the cloud, giving them a name and configuration. The configuration includes runtime parameters and worker characteristics (such as language and memory size).

Invocation Burst definitions are triggered for execution like functions in FaaS: an event or HTTP request notifies the intent to execute a burst with specific input parameters. We call each burst invocation a *burst flare* (Table 2). The main difference with FaaS is that a flare will spawn a group of parallel workers (instead of a single function instance). The service ensures that all workers run simultaneously and applies packing. In our prototype, the burst size is explicit on the size of the *inputParams* array. Hence, users have direct control over it. We believe this to be important because parallelism is strictly application-specific and depends on data volume (e.g., ETL tasks), data content (e.g., dimensionality or sparsity), or algorithm configuration (e.g., the number of clusters in *k*-means). Smart burst sizing is left for future work, i.e., the platform may automatically calculate the number of workers based on application and data information.

Table 2: Burst computing abstractions and API.

Interface	Functions
Burst Service	**deploy**(*defName*, *package*, *conf*) upload and deploy a burst definition **flare**(*defName*, [*inputParams*]) invokes a burst
Burst Function	abstract **work**(*inputParams*, *burstContext*) function to run on each worker
Burst Context	**workerID** → unique ID of this worker within the flare **burstSize** → number of total workers in the flare **packID** → unique ID of the current pack **packSize** → number of workers in the current pack **numPacks** → number of packs within the flare **belongToPack**(*workerID*) → packID returns the pack ID to which a worker belongs to **isPackLeader**() → bool returns true if this worker is its pack's leader
Comm. Primitives	**send**(*data*, *dest*) → none **recv**(*source*) → data **broadcast**(*data*, *root*) → data **allToAll**(*[data]*) → [data] **reduce**(*data*, *f*(*data*, *data*) → *data*) → data

Coding Burst definitions are coded as a single function that is run by each worker in the burst (`work` in Table 2). This function must be programmed elastically so that it accepts and runs correctly for any burst size. The code is also agnostic to the packing performed by the service. To that end, the `work` function receives a burst context object through which each worker may obtain information about the worker distribution within the particular flare. For example, a worker can query its own unique ID, the burst size, granularity, or which workers belong to each pack (*Burst Context* in Table 2). With this information (provided by the platform invoker), the code can implement logic to apply locality optimizations at the pack and burst levels (see an example in §5.4.1). This context object also gives access to the BCM.

Communication interface The BCM offers simple yet powerful worker-to-worker communication through message passing similar to MPI. The abstractions are elastic (adapt to the burst size) and available through the burst context. Burst computing programs make use of two basic primitives to connect workers: send and receive. These primitives enable point-to-point communication between workers and are designed to send arbitrary volumes of data efficiently within the burst. To facilitate common communication patterns in parallel jobs, bursts may also use group collectives. As listed in Table 2, our prototype implements broadcast, all-to-all, and reduce. Primitives and collectives are locality-aware, although the

```rust
fn work(params: Input, burst: &BurstContext) -> Output {
  let num_nodes = params.num_nodes;
  let mut page_ranks = vec![1.0 / num_nodes; num_nodes];
  let mut sum = vec![0.0; num_nodes];
  let adjacency_matrix = get_adjacency_matrix(&params);
  while err < ERROR_THRESHOLD {
    page_ranks = burst.broadcast(page_ranks, ROOT_WORKER);
    for (node, links) in graph {
      for link in links {
        sum[*link] += page_ranks[*node] / out_links(*node);
      }
    }
    let reduced_ranks = burst.reduce(sum, |vec1, vec2| {
      vec1.zip(vec2).map(|(a, b)| a + b).collect()
    });
    if burst.worker_id == ROOT_WORKER {
      err = calculate_error(&page_ranks, &reduced_ranks);
      page_ranks = reduced_ranks;
    }
    err = burst.broadcast(err, ROOT_WORKER);
    reset_sums(&mut sum);
  }
  Output { page_ranks }
}
```

Figure 5: Simplified source code of the PageRank *work* function for burst computing. The accesses to the burst context to obtain the worker ID or communicate are highlighted.

programs remain agnostic to it, i.e., co-located workers (same pack) communicate on shared memory and only remote workers hit the network.

4.3 Application example

Fig. 5 shows an example in Rust code (simplified) of the `work` function that implements the PageRank application. The algorithm consists of an iterative process in which each worker holds a portion of the adjacency graph (relating links between web pages). In each iteration, the new global ranks are computed in parallel, aggregated, and reduced in a tree structure, then broadcasted from the root worker to the rest of them. The algorithm runs until it converges past a threshold or reaches a limit of iterations.

Similarly to the MPI computing model, all workers execute the same code but perform different logic based on the worker ID (the *rank* in MPI). The example highlights the worker accesses to the `BurstContext` object to perform collectives and obtain information about the current flare. For example, it is used to perform a collective *broadcast* to share the updated ranks vector, and later a *reduce* to aggregate the partial ranks computed among the workers. It also shows how a worker checks its ID when it needs to calculate the convergence, since this is only done by the root worker after collecting the aggregated vector in the reduce.

4.4 Burst platform implementation

The prototype implementation is built on top of the popular Apache OpenWhisk platform (v1.0.0). We used OpenWhisk as the basis because it is a well-known, open-source, production-tested FaaS implementation and provides higher burstability than other platforms like Knative (Table 1). Our changes amount to approximately 2K SLOC. They affect the main components of the platform, including the controller, the invoker, and the runtime environment.

The controller now supports two new HTTP endpoints for bursts: `deploy` and `flare`. It also implements the logic to handle them (§4.1). This includes the packing strategy in the three flavors (§3): heterogeneous, homogeneous, and mixed. Granularity can be configured. In any case, the controller calculates the number and size of the packs based on the specific burst size and the resources available in the invokers.

Invokers run a new monitoring logic that can be adjusted to report their load to the controller based on CPU instead of RAM. Our prototype is set to assign 1 vCPU per worker because bursts tend to be compute-intensive jobs and we do not consider parallelism within a worker,[6] but other configurations are possible. Invokers also implement new logic to support the creation and execution of packs, spawning Docker containers of the appropriate size for each burst (by specifying resource limits) and telling each container/runtime the number of workers to run, plus their IDs and context. Containers are currently not reused across bursts.

For the runtime, we adapted the official OpenWhisk Rust environment, but it is possible to support others. The new logic allows to spawn multiple workers within it as requested by its host invoker. In particular, the Rust runtime spawns one thread per worker to provide parallelism. Finally, the runtime also includes our BCM built-in.

4.5 BCM implementation

The burst communication middleware (BCM) is coded in Rust in about 5K SLOC. It is readily available for our custom Rust runtime and we are working on a binding for Python.[7] It enables the transmission of intra-pack (zero-copy) and inter-pack (via remote backend) messages. The BCM is instantiated by the runtime (once per pack) and made available to workers as a parameter (in the `work` function as shown in Table 2).

For local communication, BCM uses in-memory queues to send and receive data between workers in the same pack. In the Rust runtime, workers are threads and reside in the same memory space, so shared memory mechanisms are not necessary (e.g., `shm_open` or `mmap`). Instead, workers just pass memory pointers between them. Thanks to Rust's memory safety guarantees, access to shared data is thread-safe. Rust also provides a reference-counting mechanism for immutable data, so shared data is released when it is no longer used at runtime. For example, the root worker in a *broadcast* sends a read-only memory pointer to its local workers, and they safely access the message concurrently. To modify the data, one may use mechanisms such as copy-on-write.

For remote communication, each pack has a shared connection pool to the remote backend, which allows each worker within the pack to send and receive messages concurrently, with the goal of maximizing the container's bandwidth. This

[6] The burst size (number of workers) determines total job parallelism.

[7] Other languages may be supported through bindings (Java, C++, Go…).

is especially useful in primitives like *all-to-all*, where all workers must open channels with all the others. For large messages, the data is split into smaller chunks that are sent and received concurrently. This maximizes network utilization and allows readers to start receiving data from the first chunk, instead of waiting for the full message to be available at the backend.

The BCM is extensible, allowing the implementation of more remote backends. Currently, we support Redis, DragonflyDB, RabbitMQ, and S3. The backend interface differentiates between sending direct messages (one-to-one) and broadcast messages (one-to-many). The reason is that direct messages are read only once, while broadcast messages multiple times, so we want to optimize this particular case. For instance, in RabbitMQ, one-to-one messages use direct brokers, while one-to-many use fan-out brokers.

To ensure that no messages are lost (*at-least-once* delivery semantics), the BCM relies first on the backend delivery guarantees (e.g., RabbitMQ uses durable queues to avoid dropping messages). Additionally, the BCM keeps a count of direct messages sent between each pair of workers, and for each collective operation. The middleware handles duplicate and/or out-of-order messages. For that, messages include a header with the source and destination worker, collective type, counter, and, if chunked, the number of chunks and chunk number. Messages with a counter lower than the expected value are ignored and assumed as already processed. Those with a counter greater than expected are cached locally until needed. For chunked messages received out-of-order, a memory region is reserved for the total payload and chunks are written to their respective offset as they come in.

5 Evaluation

Our evaluation aims to assess burst computing against current FaaS on the three friction points described in §2.1. Importantly, we show and analyze the effects of worker packing and locality. All experiments run on Amazon Web Services (AWS) in the `us-east-1` region.

5.1 Burst group invocation

Group invocation is the key element against friction **F1**. Here we evaluate how job-level isolation improves worker readiness time (invocation latency), ensures their simultaneity, and provides locality for collaborative code and data loading.

Setup: The burst platform runs on an Amazon EKS cluster, with the control plane on a `t4i.xlarge` VM (4 vCPUs and 16 GB RAM), and the invokers on up to 20 `c7i.12xlarge` VMs (48 vCPUs and 96 GB RAM). This gives us space to accommodate up to 960 workers with 1 vCPU each.

Impact on burst invocation latency First, we use the homogeneous packing policy to evaluate how assigning different granularity (g) affects burst invocation latency. The exploration is depicted in Fig. 6 for two bursts of sizes 48 (left) and 960 (right).[8] It is quickly apparent that as g increases (up to

[8]We conducted experiments with similar results for burst sizes in-between.

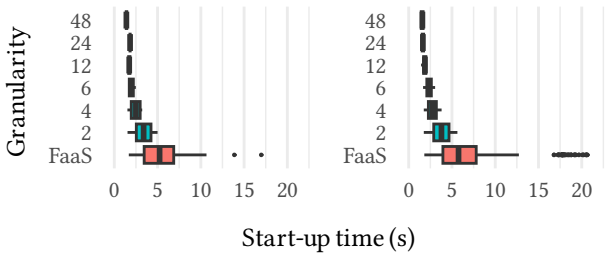


Figure 6: Worker start-up latency distribution within one job for burst computing with different packing granularity and FaaS (equivalent to $g = 1$). Left and right show, respectively, burst sizes of 48 and 960.

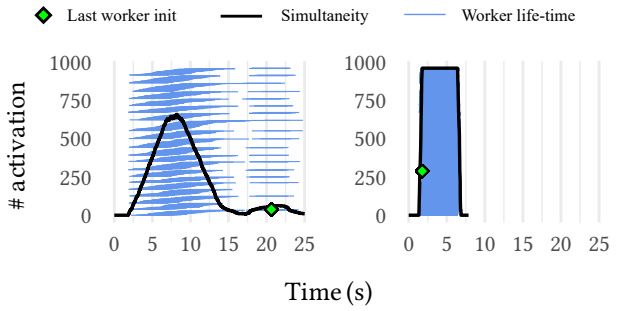


Figure 7: Simultaneity (number of workers running at an instant) in FaaS (left) and Burst with $g = 48$ (right). Each bar represents the life-time of a worker.

48 in both cases), the start-up time decreases, and generally becomes more consistent across workers for all burst sizes. For instance, the latency of having all workers ready in a burst of size 960 reduces by 11.5× from $g = 1$ (FaaS) to $g = 48$. We found that container creation dominates invocation latency, hence higher g performs best. This proves that creating the biggest possible containers, and thus the less amount of them (heterogeneous packing), achieves the best start-up latency, since it creates a single container per invoker per flare. By extension, the mixed packing strategy exhibits the same results, but allows the system to manage resources more effectively in small portions to facilitate allocation and avoid resource fragmentation. To assess the impact of granularity, the rest of the evaluation uses homogeneous packing.

Impact on worker simultaneity We run a burst with size 960 on FaaS against burst computing with $g = 48$. For demonstration purposes, each worker performs a 5-second sleep and we plot their execution timeline in Fig. 7. The plot shows that burst computing achieves faster resource allocation and quicker readiness of workers. This ensures worker parallelism. Analyzing dispersity of worker start-up time (also in Fig. 6), the FaaS execution evinces a range of 18.8 s between the start of the first worker and that of the last one, with a median absolute deviation (MAD) of 2.65 s. In contrast, the range

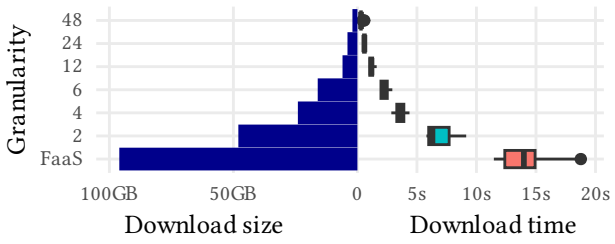


Figure 8: A burst of 96 workers loading the same 1 GiB object from S3 with different granularity.

with $g = 48$ is just 0.44 s (MAD is 0.1 s). Compared, the range is 43× lower in burst computing, with MAD showing 26.5× lower dispersity than FaaS. Dispersity in worker start-up latency precludes FaaS to achieve full parallelism (all workers running simultaneously from start to finish), while burst guarantees it.

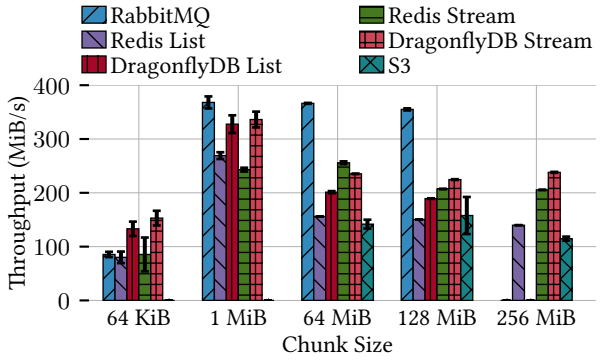
Impact on data loading Burst computing mitigates the FaaS problem of loading the same data on all functions (§2.1), e.g., in a grid search. We can leverage worker access to locality information to optimize this problem and download the data only once per pack, trivially reducing data ingestion. Specifically, each worker in a pack retrieves a part of the data based on calculations from pack information in the burst context (Table 2). Then they recreate the full data in a local shared memory region. This allows to parallelize the download and complete the process faster than choosing a leader to perform it (also possible through the `isPackLeader` function in the context, which returns true for the worker with lowest ID within its pack). We evaluate this approach on multiple g and present it in Fig. 8. Burst optimizations achieve a download time speed-up of 32.6× with $g = 48$ compared to FaaS.

Takeaway Flares eliminate friction **F1** through faster worker group initialization (11.5×) and ensured simultaneity (43× less dispersed workers) that enables locality with packing. In turn, locality may accelerate data download in applications (32.6×), tackling friction **F3**.

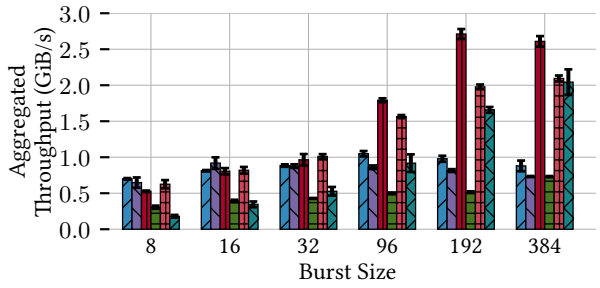
5.2 Burst inter-pack communication

Before we evaluate the effects of the BCM on frictions **F2** and **F3**, we want to ensure that an indirect communication model is feasible and to find a backend that sustains the load of bursts at scale. For this, we measure the throughput of several indirect communication backends. Specifically, we test Redis, DragonflyDB (a Redis-compatible multi-threaded alternative), RabbitMQ, and S3. Redis and DragonflyDB evaluate two flavors: using lists or streams.

Message chunk size The BCM chunks messages into several blocks to optimize network utilization and allow parallel read/write. The optimal chunk size is a trade-off between latency to first byte and operation overhead, and it varies



(a) Throughput between two remote workers sending a 1 GiB payload chunked in different sizes.

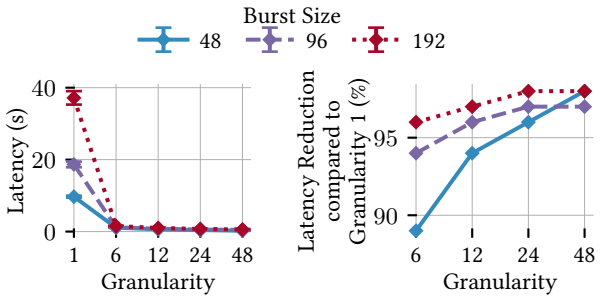


(b) Aggregate throughput of two remote packs, A and B, of varying size (g =burst size/2), where each worker from pack A sends a 256 MiB payload to another worker from remote pack B.

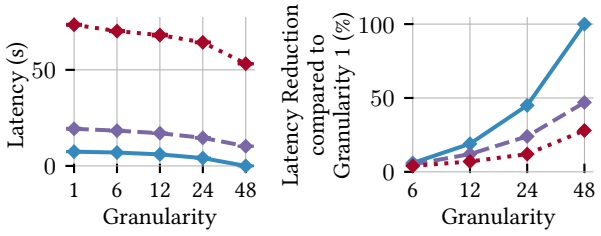
Figure 9: Throughput experiments for the different BCM backends. Median values with standard deviation (10 runs).

for each communication backend. To find the optimal configuration, we measure the throughput of sending a 1 GiB message between two remote workers. The workers run on two `c7i.large` machines (4 vCPUs, 8 GB) and we deploy a `c7i.16xlarge` (64 vCPUs, 128 GB) for the intermediate server. Fig. 9a plots the results. RabbitMQ offers a constant throughput for larger chunk sizes, but does not allow payloads larger than 128 MiB due to AMQP protocol limitations. Redis and DragonflyDB work best at 1 MiB, the latter being slightly superior. S3 offers the lowest throughput because object stores are not designed for small files (1 MiB or less exceeds the allowed service request rate limits).

Maximum throughput To understand how the different backends scale under parallel load, we measure the aggregated throughput between several pairs of workers communicating simultaneously. In this experiment, we launch a group of workers (burst size from 8 to 384) split into two remote groups. Each worker in a group A sends a fixed message (256 MiB) to a worker in the other, remote group B. As the burst size increases, so does the total data volume sent. Each backend uses the optimal chunk size assessed in the micro-benchmark above. Workers run on two VMs scaled to the burst size (from `c7i.xlarge` for 8 workers to



(a) Broadcast



(b) All-to-All

Figure 10: Latency and its reduction percentage with respect to $g = 1$ of two collectives, for varying g and burst size.

`c7i.48xlarge` for 384), and the communication server runs on one `c7i.48xlarge` instance. The results are shown in Fig. 9b. We observe that RabbitMQ does not scale beyond 1 GiB/s. For the in-memory stores, the approach with lists performs better than streams. Like RabbitMQ, Redis does not scale with parallelism because it is single-threaded. In contrast, DragonflyDB does scale and achieves the highest throughput, surpassing 2.5 GiB/s for large burst sizes. S3 also scales with parallelism but remains slower.

Takeaway The BCM achieves high throughput even with indirect communication, and some backends sustain up to the evaluated 384 workers with individual connections, suggesting a feasible approach. In view of the results, the rest of the evaluation uses DragonflyDB List with 1 MiB chunks.

5.3 Burst group collectives

We assess the impact of locality on group collectives as a means to face friction **F3**. We measure end-to-end latency, i.e., the total time it takes for all workers to complete the collective, as we vary g. We present results for *broadcast* and *all-to-all*. *Reduce* behaves similar to *broadcast* because they follow the same data movement pattern.[9]

Setup: We employ one, two and four `c7i.12xlarge` VMs (48 vCPUs, 96 GB) for bursts of sizes 48, 96 and 192, respectively. g varies from 1 to 48. Each worker uses 256 MiB of data for each collective call. The backend server runs on one `c7i.48xlarge` (192 vCPUs, 384 GB).

[9]Other collectives like *gather* and *scatter* are similar to *all-to-all*.

Overall, Fig. 10 shows that latency decreases as the granularity increases. This is because remote communication is the main bottleneck of collective operations and its volume decreases as g increases and more data movement becomes local. The cost of local communication is insignificant compared to the remote one. Broadcast sends the message once, but reads it once per pack, i.e., remote data movement is directly proportional to the number of packs: if we halve the packs, we move half the data. Thus, latency quickly decreases as we increase g; near 98% reduction with $g = 48$ (Fig. 10a). All-to-all is more intensive in data traffic because all workers have a message of 256 MiB for each of the other workers (48 GiB total with 192 workers). This means that even if we only have two packs, half the data traffic is remote. This is clearly evident in Fig. 10b. Considering $g = 48$, burst sizes 48, 96, and 192 create one, two, and four packs; thus latency reduction is ca. 100%, 50%, and 25%, respectively.

Takeaway Locality-aware group collectives heavily mitigate friction **F3** by seamlessly reducing remote data traffic.

5.4 Burst applications

We evaluate three real-world bursts: hyperparameter tuning, PageRank, and TeraSort. These (or similar) applications are commonly used in the literature to assess the performance and show the limitations of FaaS platforms for parallel jobs and serverless data analytics [4, 7, 15, 16, 28].[10] These applications clearly show all friction points while providing an overall view of the effects of burst computing compared to FaaS-based implementations. Further, they are representatives of short jobs a scientist may want to run interactively in a dynamic analysis session: sudden and quick (under 2 min).

Our baseline for comparison is thus the current FaaS paradigm available in public clouds as used in the serverless data analytics literature. Since MPI-like communication is not supported on any serverless service, we opt to use unmodified OpenWhisk and AWS Lambda as our baselines, employing external storage for communication and data sharing to align with the state of practice.

5.4.1 Hyperparameter tuning

Grid search is a machine learning technique in which a set of hyperparameter values are evaluated to find the combination that yields the best performance for a given model. This evaluation is done in parallel, with each worker processing a full copy of the training dataset. Since there are no dependencies between parallel tasks, FaaS seems suitable for this job. However, each function would download a copy of the data, regardless of whether there are functions co-located on the same node. This results in a waste of bandwidth and memory due to duplicate data downloads. Burst computing provides an optimization opportunity by exploiting locality (see impact on data loading in §5.1).

[10]We do not use FaaS benchmarks (e.g., FunctionBench [27]) because they focus on applications where FaaS already does a good job (independent function invocations and stateless or embarrassingly parallel workloads).

Table 3: Time to start 96 workers and gather input data in hyperparameter tuning for different granularity.

Granularity	1 (FaaS)	6	12	24	48	96
Ready time (s)	17.51	5.65	3.64	3.18	2.96	2.57

Setup: The grid search is applied to a stochastic gradient descent model in a `sklearn` Python application and distributed to 96 workers. We use a 500 MiB Amazon reviews dataset (CSV), available at Kaggle[11] and stored in an S3 bucket. AWS Lambda is the baseline (denoted as $g = 1$), with a memory configuration of 1769 MiB, which provides a full vCPU. The burst platform uses a `c7i.24xlarge` instance.

Table 3 collects the "ready time", meaning the time elapsed from client-side job invocation until the input data is available to all workers and they are ready to compute. We see how burst computing quickly reduces this time as g increases. This effect has two causes. First, the group invocation primitive speeds up invocation time compared to FaaS, from approx. 4 to 1.5 s with $g = 96$. Second, data download can be optimized as assessed in Fig. 8. While FaaS has to download a copy of the data on each worker, workers co-located in a pack collaborate in downloading the input in parallel. Hence, as g increases, the input download time decreases, going from 14 s in FaaS to 1 s with $g > 48$.

5.4.2 PageRank

PageRank is a well-known data analytics workload with intensive worker coordination. It involves iterative and heavy data aggregation for large datasets, which is of interest for benchmarking worker communication in burst computing. We adapted PageRank for burst computing from the Hi-Bench suite [19] (MapReduce approach). The implementation is detailed in §4.3. In this case it is not possible to make optimizations regarding data ingestion (like in §5.4.1), since each worker takes a different partition of the dataset. We skip reporting the MapReduce version atop FaaS because the number of (short) stages necessary to perform the iterative aggregations make it obviously slower. Spark has a similar problem [7]; evaluation on AWS EMR with an equal-sized deployment (needing 5 min to start up) shows that this application takes over an hour.

Setup: This experiment uses four `c7i.16xlarge` VMs (64 vCPU, 128 GB). The graph dataset is generated with Hi-Bench consisting of 50M nodes (ca. 30 GiB) in 256 partitions. The algorithm runs over 10 iterations, with a burst size of 256, varying the granularity between 1 and 64.

Fig. 11 shows the total execution time of all iterations split into phases: the time dedicated to download input data (from S3), compute the ranking, and communicate (collectives) between workers. Times for each phase are averaged across workers, and all iterations added. Table 4 shows the network traffic and % of reduction compared to $g = 1$. Communication accounts for the majority of the execution time because the rank vector must be aggregated and shared at each iteration. Our configuration uses a vector of 40 MiB that is sent, received, and aggregated in a tree pattern across the workers, and then broadcast from the root worker to the rest of them. As g increases, the remote portion of this movement decreases. For example, with $g = 2$, only the first level of the (binary) reduction tree is local (the leaves), and the rest communicate remotely. With $g = 64$, there are 4 packs, so remote communication occurs only in the last 2 tree levels. With this setup, we achieve a 98.5% reduction in data traffic and a 13× speed-up compared to $g = 1$.

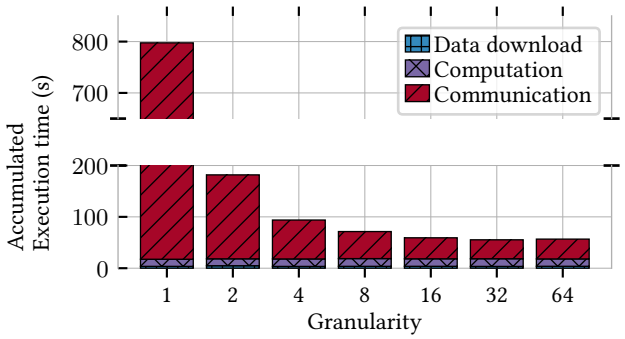


Figure 11: PageRank execution time by phase, varying g.

Table 4: Aggregated network traffic volume and percentage of traffic reduction compared to $g = 1$, varying g in PageRank.

Granularity	**1**	**2**	**4**	**8**	**16**	**32**	**64**
Traffic (GiB)	3068	1532	764	380	188	92	44
% Reduction	n/a	50.0	75.0	87.6	93.8	97.0	98.5

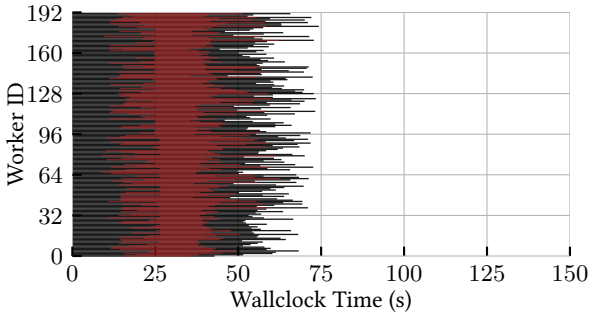
5.4.3 TeraSort

We have implemented TeraSort based on the MapReduce version in Hi-Bench suite [19]. TeraSort is of particular interest because it involves a heavy data shuffle phase. We want to compare a TeraSort following the serverless MapReduce approach [33, 40, 46], with a single-stage burst computing version where we exploit locality for the shuffle phase. The main advantages of burst are: (i) the MapReduce version requires two rounds of function invocations (map and reduce), while the burst model requires a single flare, and (ii) the MapReduce version shuffles data through object storage, while burst employs the (locality-aware) *all-to-all* collective.

Setup: This experiment sorts a 100 GiB dataset generated with Hi-Bench with 192 partitions. The burst platform runs on EKS with two `m7i.24xlarge` (96 vCPUs, 384 GB) invokers and a `c7i.xlarge` controller. The input data is in an Amazon S3 bucket located in the same region. For reference, an equal-sized Spark deployment solves this problem in 106 s average but needs 5 min to start up the cluster.

[11] https://www.kaggle.com/bittlingmayer/amazonreviews

(a) Serverless MapReduce



(b) Burst computing

Figure 12: TeraSort timeline comparison between (12a) serverless MapReduce and (12b) burst computing. MapReduce comprises two function rounds (map and reduce), with data exchange via object storage. Burst uses a single flare, exchanging data through the *all-to-all* collective.

Fig. 12 shows the timeline of two executions comparing serverless MapReduce and burst. The execution time of each worker is shown in horizontal black bars, stacked by *Worker ID* on the vertical axis. Superimposed in red, we see the time elapsed for the shuffle phase of the TeraSort algorithm. In the MapReduce version (Fig. 12a), we highlight (i) the dispersity in function start-up time, as we have seen in Fig. 7; (ii) a gap where no functions are running, caused by splitting the workload into two phases (map and reduce), with an externally-managed synchronization phase between them, adding further overhead; and (iii) an outlier in the map phase (worker #121), which slows down the entire workflow.

All these points are addressed in burst computing (Fig. 12b). First, the group invocation packs all workers into two containers of 96 workers, making start-up faster, and ensuring parallelism, which eliminates (latency-induced) outliers. Second, worker-to-worker collectives avoid splitting the job into two phases. Finally, remote communication is reduced thanks to locality (see §5.3). To wit, we achieve a 2× speed-up for this particular execution—1.91× mean across six executions.

For completeness, we also run TeraSort on Spark using AWS EMR on a cluster of equivalent resources. Best results are achieved with 74 executors of 4 GB of memory each. The total run time ranges between 100 and 110 seconds, placing it between the FaaS and Burst implementations. Although the execution times are relatively close, there are substantial differences that make a comparison and analysis of results complex. To start, the programming language changes (Scala to Rust), and Spark's execution model and its peer-to-peer communication capability impact application performance in different ways: Rust may be more performant, the burst execution model simpler and more effective, but Spark's direct communication is faster. Moreover, it is worth noting that cluster creation time on AWS EMR took over 5 minutes, adding additional overhead for an sporadic workload. Neither FaaS nor Burst require the user to set up a cluster.

Takeaway Frictions **F1**, **F2**, and **F3** appear in real-world applications. Hyperparameter tuning shows duplication in worker initialization due to friction **F1**, which also slows down worker start-up in PageRank and TeraSort. PageRank and TeraSort evidence the issues of friction **F2**: the iterative nature of PageRank makes it unfeasible in FaaS due to excessive stages, and TeraSort is hindered considerably due to slower coordination. Burst computing mitigates this by allowing workers to coordinate and share data in a single stage, instead of requiring multiple stages orchestrated externally that communicate asynchronously through storage. PageRank and TeraSort also emphasize friction **F3**: Page-Rank with iterative, large communication to aggregate the vector and TeraSort with a single, large data shuffle.

6 Discussion and limitations

Burst as a cloud service One potential concern regarding the adoption of burst computing as a public cloud service is the added scheduling complexity of packing multiple workers, along with its implications for billing. Giving burst granularity control to the provider addresses this by enabling more effective resource management. The mixed packing policy we propose further enhances scheduling efficiency by targeting small resource gaps and maximizing locality through co-located resource consolidation. Burst computing aligns with the trend toward "Big Lambda" architectures, seen in platforms such as AWS Lambda, Google Cloud Functions, and CaaS offerings like IBM Code Engine. For example, AWS already supports functions with up to 6 vCPUs, indicating that its infrastructure can already schedule resource bundles at this scale. Setting a burst granularity of 6 would thus mirror current AWS capabilities. Implementing burst computing requires the concurrent allocation of multiple such high-capacity functions, similar to existing multi-invocation FaaS handling, while optimizing locality. Though this may introduce slightly higher latency than a single function invocation, it remains more efficient than coordinating numerous remote HTTP calls. Our evaluation on OpenWhisk confirms the viability of this model, demonstrating that it preserves internal resource management and auto-scaling typical of FaaS.

We see burst computing as a natural evolution of FaaS: despite the constraint of parallel provisioning, bursts are ephemeral and may be bounded in time or concurrency, clearly distinct from long-running serverful processes. Thus, we argue that burst support in public clouds is not prohibitively complex and can follow FaaS-like billing models. Moreover, there is clear evidence that users are already willing to pay for "Big Lambdas" or containers when the workloads justify it [18].

Burst applications The suitability of burst computing versus traditional FaaS depends on an application's communication patterns, performance demands, and tolerance to execution imbalances. Burst computing is designed for massively parallel workloads with collaborating workers, enabling co-location and synchronized execution. Applications with strong locality requirements and intensive data sharing—such as PageRank, SQL queries, or large-scale sorting—are prime candidates. However, even loosely coupled workloads like grid search or Monte Carlo simulations can benefit from shared data downloads (see §5.4.1), faster initialization, and coordinated result aggregation. Burst computing also facilitates optimizations such as function fusion, where stateless functions within a dataflow are merged into a single, stateful burst to reduce communication overhead and improve locality [47]. Interactive workloads or those requiring synchronized responses across workers similarly gain from burst simultaneity. In contrast, those with minimal data sharing or high variance in execution time across workers are typically better served by conventional FaaS. While our current prototype reclaims resources per pack—leading to potential idle resources when worker durations differ—it targets short-lived, coordinated workloads (i.e., workers typically progress in lockstep) where imbalance is rare, as seen in PageRank and MilliSort/MilliQuery [31]. Addressing this limitation via dynamically sized packs is a path for future work. Ultimately, choosing between burst and FaaS depends on locality needs, parallelism requirements, and the importance of coordinated execution.

Coding limitations While burst computing introduces new capabilities for expressing parallel and stateful workloads, it also imposes certain programming constraints that must be acknowledged. The model is designed as a low-level primitive, offering developers greater flexibility rather than prescribing orchestration logic. Similar to MPI, it executes the same code across all workers while assigning each a unique identity, enabling divergence in behavior. However, unlike MPI, bursts are triggered with FaaS-like simplicity, abstracting away explicit resource management. The primary coding complexity introduced lies in inter-worker communication, yet this is mitigated by using programming primitives that are agnostic to worker count and locality, thereby simplifying development. Although multi-burst orchestration—e.g., for full DAG-based workflows—remains future work, it can leverage well-established techniques such as dependency persistence in object storage or worker reuse across flares. A higher-level framework could encapsulate this coordination, offering abstractions for DAG scheduling and state propagation across bursts [9, 36, 43]. Many real-world applications (e.g., clustering, gradient descent, aggregation queries, N-body simulations) can already be implemented within a single burst, avoiding orchestration overhead altogether. Furthermore, libraries could expose common distributed patterns (e.g., a distributed sort) via simple APIs, hiding burst-level details from end users. Thus, while the model lacks full orchestration capabilities out of the box, it provides foundational support for powerful abstractions and ensures parallel execution—a property often missing in existing FaaS-based solutions, where developers must manually coordinate parallelism without system-level guarantees [5, 9, 16].

Serverless clusters Burst computing poses a step towards redefining the boundary between serverless architectures and traditional cluster-based systems [17, 38]. Recent work has explored how to emulate cluster-like capabilities atop existing FaaS platforms [9, 57]. However, our approach takes a fundamentally different perspective: rather than adapting conventional cluster technologies to work around the limitations of current FaaS substrates, burst computing seeks to evolve serverless services themselves, introducing native abstractions and execution models that offer the parallelism, coordination, and locality benefits traditionally associated with clusters—yet within a fully serverless paradigm. This opens the door to realizing truly serverless versions of platforms like Spark, Dask, or Flink, moving beyond the managed, but still serverful, offerings available today.

7 Related work

The concept of resource burstability has already been discussed in the serverless literature. FaaS has been referred to as a "burstable supercomputer-on-demand" [16] for "burst-parallel serverless applications" [55], albeit with limitations. "Flash bursts" in HPC [31] explore the feasibility of 1 ms jobs spanning a large number of servers, which is attractive but unfeasible with existing technologies in public clouds. Granular computing [30, 42, 48] explores very similar ideas to improve cluster utilization. Müller et al. [38] pursue serverless burstability for batch jobs, highlighting that functions are too limited for the job.

Burst computing goes a step further in defining a new way of running burst-parallel jobs in the cloud. It evolves from FaaS to exploit serverless burstability, but it unlocks the limitations of working with functions to provide a compute environment tailored for massively parallel collaborative jobs. Recent works also promote the need to evolve the FaaS model to overcome important challenges [11, 13]. To our knowledge, we are the first ones to raise the unit of management from a function to the job level in a serverless service with a group invocation abstraction to pack workers together and enable locality within the job.

Several papers [12, 21, 59] tackle handling function invocations faster. They aim to respond to thousands of function calls with very low latency. Others relax function isolation to handle multiple invocations together and cut on start-up and data sharing overhead [2, 50]. Similarly, some works perform opportunistic function packing, placing multiple function invocations from the same user on the same container to improve invocation latency and resource usage [1, 8, 22, 53, 61]. This shows that burst jobs are becoming widely accepted in FaaS settings, and locality is a key enabler. All these works only consider individual function invocations and are thus limited. Burst computing further optimizes job execution thanks to group invocations and guaranteed worker parallelism. Still, the mentioned works may be combined with burst computing to accelerate resource allocation.

A different line of work handles complex computations atop FaaS through higher-level function orchestration, task schedulers, or workflow optimizers [9, 29, 32, 34, 36, 37, 62, 63]. Since they operate with individual function invocations, they cannot be compared with burst computing, but benefit from it to guarantee task parallelism and fast start-up time. Orchestration techniques may be applied atop burst computing, e.g., to coordinate multiple bursts, manage data dependencies between them, optimize task parallelism, and handle faults.

Other works explore a hybrid approach where traditional compute engines (e.g., Spark or Flink) offload computation spikes to serverless functions [14, 20, 51] to achieve faster adaptation to dynamic loads. This requires the deployment and management of the traditional compute engine, which is not ideal to achieve serverless execution of parallel jobs. Burst computing fully hides infrastructure: users only deploy their code and run jobs, while the cloud handles the rest.

Communication and state sharing in FaaS has been widely explored. Some works combine a data store and a FaaS platform into the same system to optimize data access by placing functions where the data they use is kept [52]. This is a fundamentally different kind of locality than in burst computing as it does not support any function grouping semantics and may present resource contention problems [45]. Other solutions [4, 6, 28, 40] target applications that require function coordination and communication, close to the objectives of burst computing. However, they employ a disaggregated storage solution to relay data between stateless functions and cannot exploit any form of locality. Boxer [58] explores direct communication, and FMI [10] builds a library of collectives between groups of FaaS functions (with NAT traversal). They only tackle current FaaS platforms and do not provide any locality optimizations (in contrast to burst computing). These solutions are orthogonal to our contributions. For instance, FMI may be used as a BCM remote backend to accelerate pack-to-pack transfers, but burst computing still contributes zero-copy communication between workers in the same pack.

Finally, we highlight a clear trend in FaaS that is aligned with burst computing: the emergence of "Big Lambdas" with multiple CPU cores (currently up to 6 vCPUs in AWS). Jobs may leverage function intra-parallelism and, similar to burst computing, worker locality becomes relevant and improves the execution of parallel applications. these high-capacity functions still lack group-aware mechanisms and parallel guarantees, requiring complex user-side handling.

8 Conclusion

In this paper, we have presented burst computing, a novel cloud computing model designed to address the growing demand to run sudden, variable, burst-parallel workloads without provisioning resources in advance. We have reviewed the challenges and shortcomings of current technologies (i.e., FaaS), and we have demonstrated the effectiveness and versatility of our proposed solution.

Burst computing offers several key advantages over existing FaaS technologies, primarily the addition of a group invocation primitive that allows the platform to manage jobs as a unit, instead of independent function invocations. This raises tenant isolation to the job level and allows to allocate resources en masse and apply worker packing, which in turn enables powerful locality between workers. Our experiments and performance evaluations have shown that our platform achieves significant improvements by exploiting this locality in job invocation latency, worker simultaneity, code and data loading, and worker-to-worker communication with group collectives. Further, we demonstrated speed-ups of 13×, and 2× in PageRank, and TeraSort, respectively, thereby validating efficacy in real-world scenarios.

In conclusion, burst computing represents a significant advancement for serverless data processing in the cloud, with potential to become a new paradigm of cloud services. It unlocks key limitations of FaaS, becoming the next step forward to support applications previously stymied by its restrictive model [18]. We believe that our contributions are just the substrate for further innovation such as new workflow definition tools and orchestration engines that leverage burst computing jobs and strive towards a simplified dynamic utilization of cloud resources.

Acknowledgments

We thank our shepherd Alex Merenstein, the anonymous reviewers, and early readers for their valuable feedback. This work is partially funded by the Horizon Europe programme under grant agreements No. 101092644 (Neardata) and No. 101093110 (Extract); and by the Spanish Ministry of Economic Affairs and Digital Transformation and the European Union-NextGenerationEU (PRTR and MRR framework) through the CLOUDLESS UNICO I+D CLOUD 2022.

Availability

The burst computing platform prototype and evaluation code is available online at https://github.com/Burst-Computing.

References

[1] Mania Abdi, Samuel Ginzburg, Xiayue Charles Lin, Jose Faleiro, Gohar Irfan Chaudhry, Inigo Goiri, Ricardo Bianchini, Daniel S Berger, and Rodrigo Fonseca. 2023. Palette Load Balancing: Locality Hints for Serverless Functions. In *Proceedings of the Eighteenth European Conference on Computer Systems (EuroSys '23)*. Association for Computing Machinery, New York, NY, USA, 365–380. https://doi.org/10.1145/3552326.3567496

[2] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. 2018. SAND: Towards High-Performance Serverless Computing. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 923–935. https://www.usenix.org/conference/atc18/presentation/akkus

[3] Lixiang Ao, Liz Izhikevich, Geoffrey M. Voelker, and George Porter. 2018. Sprocket: A Serverless Video Processing Framework. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC '18)*. Association for Computing Machinery, New York, NY, USA, 263–274. https://doi.org/10.1145/3267809.3267815

[4] Daniel Barcelona-Pons, Pedro García-López, and Bernard Metzler. 2023. Glider: Serverless Ephemeral Stateful Near-Data Computation. In *Proceedings of the 24th International Middleware Conference* (Bologna, Italy) *(Middleware '23)*. Association for Computing Machinery, New York, NY, USA, 247–260. https://doi.org/10.1145/3590140.3629119

[5] Daniel Barcelona-Pons and Pedro García-López. 2020. Benchmarking Parallelism in FaaS Platforms. *Future Generation Computer Systems* 124 (Oct. 2020), 268–284. https://doi.org/10.1016/j.future.2021.06.005

[6] Daniel Barcelona-Pons, Marc Sánchez-Artigas, Gerard París, Pierre Sutra, and Pedro García-López. 2019. On the FaaS Track: Building Stateful Distributed Applications with Serverless Architectures. In *Proceedings of the 20th International Middleware Conference* (Davis, CA, USA) *(Middleware '19)*. Association for Computing Machinery, New York, NY, USA, 41–54. https://doi.org/10.1145/3361525.3361535

[7] Daniel Barcelona-Pons, Pierre Sutra, Marc Sánchez-Artigas, Gerard París, and Pedro García-López. 2022. Stateful Serverless Computing with Crucial. *ACM Transactions on Software Engineering and Methodology* 31, 3 (March 2022), 39:1–39:38. https://doi.org/10.1145/3490386

[8] Rohan Basu Roy, Tirthak Patel, Richmond Liew, Yadu Nand Babuji, Ryan Chard, and Devesh Tiwari. 2023. ProPack: Executing Concurrent Serverless Functions Faster and Cheaper. In *Proceedings of the 32nd International Symposium on High-Performance Parallel and Distributed Computing* (Orlando, FL, USA) *(HPDC '23)*. Association for Computing Machinery, New York, NY, USA, 211–224. https://doi.org/10.1145/3588195.3592988

[9] Benjamin Carver, Jingyuan Zhang, Ao Wang, Ali Anwar, Panruo Wu, and Yue Cheng. 2020. Wukong: a scalable and locality-enhanced framework for serverless parallel computing. In *Proceedings of the 11th ACM Symposium on Cloud Computing* (Virtual Event, USA) *(SoCC '20)*. Association for Computing Machinery, New York, NY, USA, 1–15. https://doi.org/10.1145/3419111.3421286

[10] Marcin Copik, Roman Böhringer, Alexandru Calotoiu, and Torsten Hoefler. 2023. FMI: Fast and Cheap Message Passing for Serverless Functions. In *Proceedings of the 37th International Conference on Supercomputing* (Orlando, FL, USA) *(ICS '23)*. Association for Computing Machinery, New York, NY, USA, 373–385. https://doi.org/10.1145/3577193.3593718

[11] Marcin Copik, Alexandru Calotoiu, Gyorgy Rethy, Roman Böhringer, Rodrigo Bruno, and Torsten Hoefler. 2024. Process-as-a-Service: Unifying Elastic and Stateful Clouds with Serverless Processes. In *Proceedings of the 2024 ACM Symposium on Cloud Computing* (Redmond, WA, USA) *(SoCC '24)*. Association for Computing Machinery, New York, NY, USA, 223–242. https://doi.org/10.1145/3698038.3698567

[12] Marcin Copik, Konstantin Taranov, Alexandru Calotoiu, and Torsten Hoefler. 2023. rFaaS: Enabling High Performance Serverless with RDMA and Leases. In *2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE Computer Society, Los Alamitos, CA, USA, 897–907. https://doi.org/10.1109/IPDPS54959.2023.00094

[13] Germán T. Eizaguirre, Daniel Barcelona-Pons, Aitor Arjona, Gil Vernik, Pedro García-López, and Theodore Alexandrov. 2024. Serverful Functions: Leveraging Servers in Complex Serverless Workflows (industry track). In *Proceedings of the 25th International Middleware Conference Industrial Track* (Hong Kong, Hong Kong) *(Middleware Industrial Track '24)*. Association for Computing Machinery, New York, NY, USA, 15–21. https://doi.org/10.1145/3700824.3701095

[14] Apache Flink. 2023. Stateful Functions. Retrieved Oct. 8, 2024 from https://nightlies.apache.org/flink/flink-statefun-docs-master/

[15] Sadjad Fouladi, Francisco Romero, Dan Iter, Qian Li, Shuvo Chatterjee, Christos Kozyrakis, Matei Zaharia, and Keith Winstein. 2019. From Laptop to Lambda: Outsourcing Everyday Jobs to Thousands of Transient Functional Containers. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 475–488. http://www.usenix.org/conference/atc19/presentation/fouladi

[16] Sadjad Fouladi, Riad S. Wahby, Brennan Shacklett, Karthikeyan Vasuki Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter,

and Keith Winstein. 2017. Encoding, Fast and Slow: Low-Latency Video Processing Using Thousands of Tiny Threads. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Association, Boston, MA, 363–376. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/fouladi

[17] Pedro Garcia Lopez, Aleksander Slominski, Bernard Metzler, Michael Berhendt, and Simon Shillaker. 2024. Serverless End Game: Disaggregation enabling Transparency. In *Proceedings of the 2nd Workshop on SErverless Systems, Applications and MEthodologies* (Athens, Greece) *(SESAME '24)*. Association for Computing Machinery, New York, NY, USA, 9–14. https://doi.org/10.1145/3642977.3652094

[18] Joseph M. Hellerstein, Jose Faleiro, Joseph E. Gonzalez, Johann Schleier-Smith, Vikram Sreekanti, Alexey Tumanov, and Chenggang Wu. 2018. Serverless Computing: One Step Forward, Two Steps Back. https://doi.org/10.48550/ARXIV.1812.03651

[19] Bo Huang, Shengsheng Huang, Jinquan Dai, Jie Huang, and Tao Xie. 2010. The HiBench benchmark suite: Characterization of the MapReduce-based data analysis. In *2010 IEEE 26th International Conference on Data Engineering Workshops (ICDEW 2010)*. IEEE Computer Society, Los Alamitos, CA, USA, 41–51. https://doi.org/10.1109/ICDEW.2010.5452747

[20] Aman Jain, Ata F. Baarzi, George Kesidis, Bhuvan Urgaonkar, Nader Alfares, and Mahmut Kandemir. 2020. SplitServe: Efficiently Splitting Apache Spark Jobs Across FaaS and IaaS. In *Proceedings of the 21st International Middleware Conference* (Delft, Netherlands) *(Middleware '20)*. Association for Computing Machinery, New York, NY, USA, 236–250. https://doi.org/10.1145/3423211.3425695

[21] Zhipeng Jia and Emmett Witchel. 2021. Nightcore: efficient and scalable serverless computing for latency-sensitive, interactive microservices. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '21)*. Association for Computing Machinery, New York, NY, USA, 152–166. https://doi.org/10.1145/3445814.3446701

[22] Chao Jin, Zili Zhang, Xingyu Xiang, Songyun Zou, Gang Huang, Xuanzhe Liu, and Xin Jin. 2023. Ditto: Efficient Serverless Analytics with Elastic Parallelism. In *Proceedings of the ACM SIGCOMM 2023 Conference* (New York, NY, USA) *(ACM SIGCOMM '23)*. Association for Computing Machinery, New York, NY, USA, 406–419. https://doi.org/10.1145/3603269.3604816

[23] Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. 2017. Occupy the cloud: distributed computing for the 99%. In *Proceedings of the 2017 Symposium on Cloud Computing (SoCC '17)*. Association for Computing Machinery, New York, NY, USA, 445–451. https://doi.org/10.1145/3127479.3128601

[24] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Menezes Carreira, Karl Krauth, Neeraja Yadwadkar, Joseph Gonzalez, Raluca Ada Popa, Ion Stoica, and David A. Patterson. 2019. *Cloud Programming Simplified: A Berkeley View on Serverless Computing*. Technical Report UCB/EECS-2019-3. EECS Department, University of California, Berkeley. https://www2.eecs.berkeley.edu/Pubs/TechRpts/2019/EECS-2019-3.pdf

[25] Kostis Kaffes, Neeraja J. Yadwadkar, and Christos Kozyrakis. 2019. Centralized Core-granular Scheduling for Serverless Functions. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC '19)*. Association for Computing Machinery, New York, NY, USA, 158–164. https://doi.org/10.1145/3357223.3362709

[26] Nima Kaviani, Dmitriy Kalinin, and Michael Maximilien. 2019. Towards Serverless as Commodity: a case of Knative. In *Proceedings of the 5th International Workshop on Serverless Computing* (Davis, CA, USA) *(WOSC '19)*. Association for Computing Machinery, New York, NY, USA, 13–18. https://doi.org/10.1145/3366623.3368135

[27] Jeongchul Kim and Kyungyong Lee. 2019. FunctionBench: A Suite of Workloads for Serverless Cloud Function Service. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*. IEEE Computer Society, Los Alamitos, CA, USA, 502–504. https://doi.org/10.1109/CLOUD.2019.00091

[28] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. 2018. Pocket: Elastic Ephemeral Storage for Serverless Analytics. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 427–444. https://www.usenix.org/conference/osdi18/presentation/klimovic

[29] Swaroop Kotni, Ajay Nayak, Vinod Ganapathy, and Arkaprava Basu. 2021. Faastlane: Accelerating Function-as-a-Service Workflows. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, Boston, MA, 805–820. https://www.usenix.org/conference/atc21/presentation/kotni

[30] Collin Lee and John Ousterhout. 2019. Granular Computing. In *Proceedings of the Workshop on Hot Topics in Operating Systems* (Bertinoro, Italy) *(HotOS '19)*. Association for Computing Machinery, New York, NY, USA, 149–154. https://doi.org/10.1145/3317550.3321447

[31] Yilong Li, Seo Jin Park, and John Ousterhout. 2021. MilliSort and MilliQuery: Large-Scale Data-Intensive Computing in Milliseconds. In *18th USENIX Symposium on Networked Systems Design and Implementa-*

tion (NSDI 21). USENIX Association, Boston, MA, 593–611. https://www.usenix.org/conference/nsdi21/presentation/li-yilong

[32] Yiming Li, Laiping Zhao, Yanan Yang, and Wenyu Qu. 2023. Rethinking Deployment for Serverless Functions: A Performance-First Perspective. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Denver, CO, USA) *(SC '23)*. Association for Computing Machinery, New York, NY, USA, Article 67, 14 pages. https://doi.org/10.1145/3581784.3613211

[33] Bryan Liston. 2016. Ad Hoc Big Data Processing Made Simple with Serverless MapReduce. Retrieved May 02, 2024 from https://aws.amazon.com/blogs/compute/ad-hoc-big-data-processing-made-simple-with-serverless-mapreduce/

[34] David H. Liu, Amit Levy, Shadi Noghabi, and Sebastian Burckhardt. 2023. Doing More with Less: Orchestrating Serverless Applications without an Orchestrator. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. USENIX Association, Boston, MA, 1505–1519. https://www.usenix.org/conference/nsdi23/presentation/liu-david

[35] Fangming Lu, Xingda Wei, Zhuobin Huang, Rong Chen, Minyu Wu, and Haibo Chen. 2024. Serialization/Deserialization-free State Transfer in Serverless Workflows. In *Proceedings of the Nineteenth European Conference on Computer Systems* (Athens, Greece) *(EuroSys '24)*. Association for Computing Machinery, New York, NY, USA, 132–147. https://doi.org/10.1145/3627703.3629568

[36] Ashraf Mahgoub, Edgardo Barsallo Yi, Karthick Shankar, Sameh Elnikety, Somali Chaterji, and Saurabh Bagchi. 2022. ORION and the Three Rights: Sizing, Bundling, and Prewarming for Serverless DAGs. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, 303–320. https://www.usenix.org/conference/osdi22/presentation/mahgoub

[37] Shruti Mohanty, Vivek M. Bhasi, Myungjun Son, Mahmut Taylan Kandemir, and Chita Das. 2024. FAAStloop: Optimizing Loop-Based Applications for Serverless Computing. In *Proceedings of the 2024 ACM Symposium on Cloud Computing* (Redmond, WA, USA) *(SoCC '24)*. Association for Computing Machinery, New York, NY, USA, 943–960. https://doi.org/10.1145/3698038.3698560

[38] Ingo Müller, Rodrigo Bruno, Ana Klimovic, John Wilkes, Eric Sedlar, and Gustavo Alonso. 2020. Serverless Clusters: The Missing Piece for Interactive Batch Applications? (April 2020), 3 pages. https://doi.org/10.3929/ethz-b-000405616 Presented at 10th Workshop on Systems for Post-Moore Architectures (SPMA 2020).

[39] Gerard París, Pedro García-López, and Marc Sánchez-Artigas. 2020. Serverless Elastic Exploration of Unbalanced Algorithms. In *2020 IEEE 13th International Conference on Cloud Computing (CLOUD)*. IEEE Computer Society, Los Alamitos, CA, USA, 149–157. https://doi.org/10.1109/CLOUD49709.2020.00033 ISSN: 2159-6190.

[40] Qifan Pu, Shivaram Venkataraman, and Ion Stoica. 2019. Shuffling, Fast and Slow: Scalable Analytics on Serverless Infrastructure. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA, 193–206. https://www.usenix.org/conference/nsdi19/presentation/pu

[41] Wei Qiu, Marcin Copik, Yun Wang, Alexandru Calotoiu, and Torsten Hoefler. 2023. User-guided Page Merging for Memory Deduplication in Serverless Systems. In *2023 IEEE International Conference on Big Data (BigData)*. IEEE Computer Society, Los Alamitos, CA, USA, 159–169. https://doi.org/10.1109/BigData59044.2023.10386487

[42] Zhenyuan Ruan, Shihang Li, Kaiyan Fan, Seo Jin Park, Marcos K. Aguilera, Adam Belay, and Malte Schwarzkopf. 2025. Quicksand: Harnessing Stranded Datacenter Resources with Granular Computing. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*. USENIX Association, Philadelphia, PA, 147–165. https://www.usenix.org/conference/nsdi25/presentation/ruan

[43] Josep Sampe, Marc Sanchez-Artigas, Gil Vernik, Ido Yehekzel, and Pedro Garcia-Lopez. 2023. Outsourcing Data Processing Jobs With Lithops. *IEEE Transactions on Cloud Computing* 11, 01 (Jan. 2023), 1026–1037. https://doi.org/10.1109/TCC.2021.3129000

[44] Josep Sampé, Gil Vernik, Marc Sánchez-Artigas, and Pedro García-López. 2018. Serverless Data Analytics in the IBM Cloud. In *Proceedings of the 19th International Middleware Conference Industry* (Rennes, France) *(Middleware '18)*. Association for Computing Machinery, New York, NY, USA, 1–8. https://doi.org/10.1145/3284028.3284029

[45] Josep Sampé, Marc Sánchez-Artigas, Pedro García-López, and Gerard París. 2017. Data-driven serverless functions for object storage. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference (Middleware '17)*. Association for Computing Machinery, New York, NY, USA, 121–133. https://doi.org/10.1145/3135974.3135980

[46] Marc Sánchez-Artigas, Germán T. Eizaguirre, Gil Vernik, Lachlan Stuart, and Pedro García-López. 2020. Primula: a Practical Shuffle/Sort Operator for Serverless Computing. In *Proceedings of the 21st International Middleware Conference Industrial Track* (Delft, Netherlands) *(Middleware '20)*. Association for Computing Machinery, New York, NY, USA, 31–37. https://doi.org/10.1145/3429357.3430522

[47] Trever Schirmer, Joel Scheuner, Tobias Pfandzelter, and David Bermbach. 2022. Fusionize: Improving Serverless Application Performance through Feedback-Driven Function Fusion. In *2022 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE Computer Society, Los Alamitos, CA, USA, 85–95. https://doi.org/10.1109/IC2E55432.2022.00017

[48] Carlos Segarra, Simon Shillaker, Guo Li, Eleftheria Mappoura, Rodrigo Bruno, Lluís Vilanova, and Peter Pietzuch. 2025. GRANNY: Granular Management of Compute-Intensive Applications in the Cloud. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*. USENIX Association, Philadelphia, PA, 205–218. https://www.usenix.org/conference/nsdi25/presentation/segarra

[49] Vaishaal Shankar, Karl Krauth, Kailas Vodrahalli, Qifan Pu, Benjamin Recht, Ion Stoica, Jonathan Ragan-Kelley, Eric Jonas, and Shivaram Venkataraman. 2020. Serverless linear algebra. In *Proceedings of the 11th ACM Symposium on Cloud Computing* (Virtual Event, USA) *(SoCC '20)*. Association for Computing Machinery, New York, NY, USA, 281–295. https://doi.org/10.1145/3419111.3421287

[50] Simon Shillaker and Peter Pietzuch. 2020. Faasm: Lightweight Isolation for Efficient Stateful Serverless Computing. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, Boston, MA, 419–433. https://www.usenix.org/conference/atc20/presentation/shillaker

[51] Won Wook Song, Taegeon Um, Sameh Elnikety, Myeongjae Jeon, and Byung-Gon Chun. 2023. Sponge: Fast Reactive Scaling for Stream Processing with Serverless Frameworks. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. USENIX Association, Boston, MA, 301–314. https://www.usenix.org/conference/atc23/presentation/song

[52] Vikram Sreekanti, Chenggang Wu, Xiayue Charles Lin, Johann Schleier-Smith, Joseph E. Gonzalez, Joseph M. Hellerstein, and Alexey Tumanov. 2020. Cloudburst: stateful functions-as-a-service. *Proceedings of the VLDB Endowment* 13, 12 (July 2020), 2438–2452. https://doi.org/10.14778/3407790.3407836

[53] Jovan Stojkovic, Tianyin Xu, Hubertus Franke, and Josep Torrellas. 2023. MXFaaS: Resource Sharing in Serverless Environments for Parallelism and Efficiency. In *Proceedings of the 50th Annual International Symposium on Computer Architecture* (Orlando, FL, USA) *(ISCA '23)*. Association for Computing Machinery, New York, NY, USA, Article 34, 15 pages. https://doi.org/10.1145/3579371.3589069

[54] Ali Tariq, Austin Pahl, Sharat Nimmagadda, Eric Rozner, and Siddharth Lanka. 2020. Sequoia: enabling quality-of-service in serverless computing. In *Proceedings of the 11th ACM Symposium on Cloud Computing (SoCC '20)*. Association for Computing Machinery, New York, NY, USA, 311–327. https://doi.org/10.1145/3419111.3421306

[55] Shelby Thomas, Lixiang Ao, Geoffrey M. Voelker, and George Porter. 2020. Particle: ephemeral endpoints for serverless networking. In *Proceedings of the 11th ACM Symposium on Cloud Computing (SoCC '20)*. Association for Computing Machinery, New York, NY, USA, 16–29. https://doi.org/10.1145/3419111.3421275

[56] Bishoy Wadie, Lachlan Stuart, Christopher M. Rath, Bernhard Drotleff, Sergii Mamedov, and Theodore Alexandrov. 2024. METASPACE-ML: Context-specific metabolite annotation for imaging mass spectrometry using machine learning. *Nature Communications* 15, 9110 (2024), 16 pages. https://doi.org/10.1038/s41467-024-52213-9

[57] Michael Wawrzoniak, Gianluca Moro, Rodrigo Bruno, Ana Klimovic, and Gustavo Alonso. 2024. Off-the-shelf Data Analytics on Serverless. In *Proceedings of the 14th Conference on Innovative Data Systems Research, CIDR 2024*. CIDR, Chaminade, USA, 10 pages. https://doi.org/10.3929/ethz-b-000646171

[58] Michal Wawrzoniak, Ingo Müller, Gustavo Alonso, and Rodrigo Bruno. 2021. Boxer: Data Analytics on Network-enabled Serverless Platforms. In *11th Conference on Innovative Data Systems Research, CIDR 2021*. CIDR, Chaminade, USA, 8 pages. https://doi.org/10.3929/ethz-b-000456492

[59] Xingda Wei, Fangming Lu, Tianxia Wang, Jinyu Gu, Yuhan Yang, Rong Chen, and Haibo Chen. 2023. No Provisioned Concurrency: Fast RDMA-codesigned Remote Fork for Serverless Computing. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. USENIX Association, Boston, MA, 497–517. https://www.usenix.org/conference/osdi23/presentation/wei-rdma

[60] Sebastian Werner and Stefan Tai. 2024. A reference architecture for serverless big data processing. *Future Generation Computer Systems* 155 (2024), 179–192. https://doi.org/10.1016/j.future.2024.01.029

[61] Zhaorui Wu, Yuhui Deng, Yi Zhou, Jie Li, Shujie Pang, and Xiao Qin. 2024. FaaSBatch: Boosting Serverless Efficiency With In-Container Parallelism and Resource Multiplexing . *IEEE Trans. Comput.* 73, 04 (April 2024), 1071–1085. https://doi.org/10.1109/TC.2024.3352834

[62] Minchen Yu, Tingjia Cao, Wei Wang, and Ruichuan Chen. 2023. Following the Data, Not the Function: Rethinking Function Orchestration in Serverless Computing. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. USENIX Association, Boston, MA, 1489–1504. https://www.usenix.org/conference/nsdi23/presentation/yu

[63] Hong Zhang, Yupeng Tang, Anurag Khandelwal, Jingrong Chen, and Ion Stoica. 2021. Caerus: NIM-

BLE Task Scheduling for Serverless Analytics. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, Boston, MA, 653–669. https://www.usenix.org/conference/nsdi21/presentation/zhang-hong

DEEPSERVE: Serverless Large Language Model Serving at Scale

Junhao Hu[23*], Jiang Xu[1], Zhixia Liu[1], Yulong He[1], Yuetao Chen[1], Hao Xu[1], Jiang Liu[1], Jie Meng[1], Baoquan Zhang[1], Shining Wan[1], Gengyuan Dan[1], Zhiyu Dong[1], Zhihao Ren[1], Changhong Liu[1], Tao Xie[32], Dayun Lin[1], Qin Zhang[1], Yue Yu[1], Hao Feng[1], Xusheng Chen[1†], Yizhou Shan[1†]

[1]*Huawei Cloud* [2]*SCS, Peking University* [3]*Key Lab of HCST (PKU), MOE*

Abstract

In this paper, we propose DEEPSERVE, a scalable and serverless AI platform designed to efficiently serve large language models (LLMs) at scale in cloud environments. DEEPSERVE addresses key challenges such as resource allocation, serving efficiency, and cold start latencies through four main design components. First, DEEPSERVE uses a simple serverless abstraction called the request-job-task model, which helps manage diverse AI workloads across post-training and model-serving tasks. Second, DEEPSERVE integrates an in-house serving engine named FLOWSERVE using a microkernel-inspired design, NPU-centric execution, and SPMD-based parallelism to optimize LLM serving. Third, DEEPSERVE includes novel scheduling policies tailored for a configuration with both PD-disaggregated and PD-colocated instances. Fourth, DEEPSERVE includes optimizations such as pre-warmed pods, DRAM pre-loading, and NPU-fork, which allow DEEPSERVE to scale up to 64 instances in seconds. DEEPSERVE has been in production for over a year, operating on a large Ascend NPU cluster and providing industry-standard APIs for fine-tuning, agent serving, and model serving to our customers.

1 Introduction

The rapid rise of generative AI, exemplified by the success of platforms such as ChatGPT, has transformed the landscape of AI and increased the demand for scalable systems capable of serving large language models (LLMs). Model-as-a-Service (MaaS) platforms, such as OpenAI's offerings, enable millions of users to access powerful AI capabilities through a simple API interface. Consequently, LLM serving has become one of the most crucial workloads in modern data centers.

Ensuring such services' performance, efficiency, and cost-effectiveness is complex, and we identify the following three challenges to achieve optimal performance and resource utilization while ensuring Service Level Objectives (SLOs) for multi-tenant users. First, AI workloads vary significantly in duration, from fine-tuning that can last hours or even days, to agent serving and LLM serving, which typically range from seconds to minutes. This variation makes it difficult to dynamically allocate shared resources without either underutilizing or overloading the system. Second, as LLM serving becomes increasingly distributed and stateful, managing resource allocation, synchronization, and fault tolerance across multiple instances is more complex. A single inference request may span multiple distributed instances and involve cached states, creating additional challenges in ensuring efficient operation. Third, the variability in LLM serving demands leads to fluctuating resource needs, further complicating resource optimization and the handling of cold-start latencies [10].

To address the preceding challenges, in this paper, we propose DEEPSERVE, Huawei Cloud's fully-hosted and serverless platform, offering industry-standard fine-tuning, agent serving, and model-serving APIs [31]. DEEPSERVE has been running for over a year atop a large Ascend NPU cluster [24]. We report how DEEPSERVE addresses the preceding challenges using four major design components: serverless abstraction and infrastructure, serving engine, scheduling algorithms, and scaling optimizations.

Serverless Infrastructure. DEEPSERVE introduces a developer-facing serverless abstraction called the request-job-task model. In this model, a *request* is an external trigger (e.g., a user-sent HTTP call). A *job* (or multiple *jobs*) matching the request type (e.g., chat, fine-tuning) handles the request. A task is a fine-grained operation within a job (e.g., prefill task, decode task). Users send HTTP *requests* that trigger internal *jobs*, which are broken down into smaller *tasks*. This request-job-task abstraction allows for the dynamic scaling of workloads and efficient resource sharing across post-training and model-serving tasks. DEEPSERVE's architecture, as shown in Figure 1, is built around this abstraction and consists of three core components: Job Executors (JEs), Task Executors (TEs), and a cluster manager. JEs handle incoming requests

*This work was completed during his internship at Huawei Cloud.
†Co-corresponding authors.

and decompose them into manageable tasks that are then distributed to TEs for execution. The cluster manager ensures the health and scaling of the job and task executors. DEEPSERVE deploys JEs and TEs dedicated to post-training and serving. We focus on serving in this paper.

Serving Engine. In DEEPSERVE, we design an efficient serving engine called FLOWSERVE, built on three fundamental principles. The first is a microkernel-inspired design, which divides system functionality into modular components that can scale independently. This separation ensures that different system parts can evolve and operate asynchronously. The second principle is NPU-centric execution, which aims to keep the NPU busy to minimize delays caused by other resources such as CPUs, DRAM, or storage. Finally, FLOWSERVE adopts a Single-Program-Multiple-Data (SPMD)-based design, enabling efficient parallel processing and scaling across multiple NPUs.

The FLOWSERVE engine has six core functionalities: tokenization, model execution, scheduler, memory management, caching management, and networking management. The tokenizer operates independently. Adhering to the SPMD design principle, FLOWSERVE follows a master-executor architecture: the master oversees scheduling, caching, and networking decisions, while per-NPU executors carry out these decisions on their respective NPUs. FLOWSERVE's scheduler is centralized at the master module, using both asynchronous KV cache prefetch and asynchronous execution to keep an NPU busy. FLOWSERVE has a Relational Tensor Cache (RTC) module to manage the relationship between tensors (primarily on the KV cache) and a distributed flow (DistFlow) module to transfer tensors across tiered storage within a single engine or across engines in a peer-to-peer manner.

Scheduling Algorithm. DEEPSERVE presents three designs to tackle the challenges introduced by prefix caching and disaggregation. First, we implement a locality-aware algorithm to maximize KV cache reuse, a design shared by previous work [11, 39, 50]. Second, we extensively compare PD-disaggregated and PD-colocated TEs in a controlled environment with varying configurations. Based on the profiling results, we develop a PD-aware scheduling policy that accounts for the dynamics of online serving and decode-length uncertainty. Finally, we propose a combined scheduling algorithm that integrates load-aware, locality-aware, and PD-aware strategies.

Scaling Optimization. DEEPSERVE achieves fast scaling by quickly adjusting to fluctuating workloads. It employs multiple key techniques, including reserving pre-warmed Pods and TEs, pre-loading models into DRAM, and leveraging high-speed NPU-to-NPU links for efficient model loading. Combined with optimizations such as parallel initialization and predictive model pre-loading, these techniques significantly reduce initialization time. DEEPSERVE can scale up to 64 instances in parallel within seconds.

This paper makes the following main contributions:

- Design of DEEPSERVE, a large-scale serverless AI platform for LLM serving (§3).
- Design of FLOWSERVE, an efficient and modular serving engine architecture (§4).
- Study of scheduling techniques for PD-disaggregated and PD-colocated setups (§5).
- Detailed description of optimization techniques for fast scaling in LLM serving (§6).

2 Background

We describe LLM and Ascend NPU chips in this section.

LLM. LLM inference consists of two main stages: the *prefill stage* and the *decode stage*. In the prefill stage, the model processes the full input prompt $(x_1, x_2, \ldots, x_n)$, computes Key-Value (KV) vectors for each token, and stores them in the KV cache [14, 15, 22]. The model then generates the next token x_{n+1} to initiate the decode stage. This stage is compute-bound and benefits from high parallelism. In the decode stage, the model iteratively generates one token at a time. For each step, it computes and appends the corresponding KV vectors to the KV cache. Token generation continues until a stopping condition is met. This stage is memory-bound, dominated by cache lookup and memory access rather than computation.

Ascend NPU and SuperPod. Our system runs on Huawei's Ascend Neural Processing Unit (NPU) AI chips [24]. The Ascend 910B provides 400 TFLOPS of FP16 compute and 64 GB of HBM. The newer Ascend 910C consists of two dies, each offering 400 TFLOPS and 64 GB of HBM. We currently support two generations of NPU clusters (see Figure 1). The first generation uses a scaled-out architecture. Each server contains 8 Ascend 910B chips, and servers are connected via a 200 Gbps RoCE network. The second generation adopts a scaled-up architecture, recently released as the CLOUDMATRIX384 SuperPod [37]. It comprises 48 servers and 384 Ascend 910C chips, all interconnected via a high-bandwidth network around 200 GB/s (unidirectional). All CPUs and NPUs in the SuperPod share a unified memory address space, enabling access to any chip's DRAM or HBM.

3 DEEPSERVE: A Serverless AI Platform

DEEPSERVE is Huawei Cloud's fully-hosted and serverless platform for running emerging generative AI workloads, offering industry-standard fine-tuning, agent serving, and model-serving APIs [31]. DEEPSERVE has been running for over a year atop a large Ascend NPU cluster. As a cloud platform, our goal is to maximize AI cluster *performance* and *utilization* while ensuring SLO guarantees for multi-tenant users. However, achieving this goal with emerging AI workloads faces the following three key challenges.

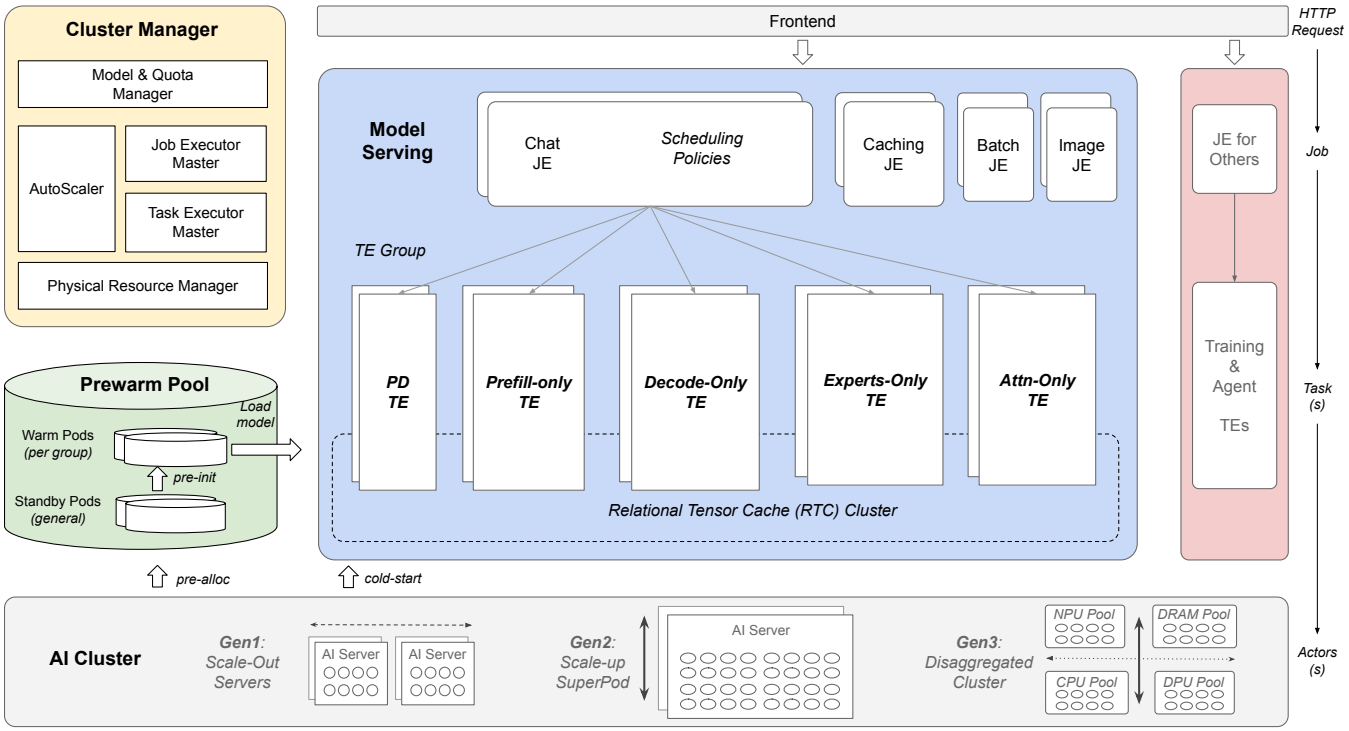


Figure 1: **DEEPSERVE Overall Architecture**. (a) User requests are routed through the frontend and dispatched to the appropriate JEs. (b) Model serving includes various types of JEs, such as those for chat completion and batch inference. (c) Each model-serving JE independently runs distributed scheduling algorithms. (d) Model-serving TEs in an RTC group can exchange tensors using DistFlow APIs (see §4.4). (e) The cluster manager is a centralized, high-availability module. (f) We omit post-training and agent serving for brevity. (g) We plan to run three generations of the NPU cluster. We currently support Gen1 and Gen2.

- Challenge 1: AI workloads have varying durations, posing challenges for efficient resource sharing. For example, fine-tuning can take hours to days, while agent serving and LLM serving typically take seconds to minutes. The challenge lies in dynamically allocating shared resources among these workloads without underutilizing or overloading the system.

- Challenge 2: LLM serving is becoming more distributed and stateful. A single inference request may span multiple distributed instances [12, 33, 51] and use cached states [11, 14, 50], making it challenging to efficiently manage resource allocation, synchronization, and fault tolerance across the system.

- Challenge 3: LLM serving is highly variable, leading to fluctuating resource demands [10, 49]. This fluctuation introduces challenges in optimizing resource utilization and handling cold-start latencies.

To address these challenges, DEEPSERVE (Figure 1) integrates the following four key designs.

Serverless Abstraction. DEEPSERVE introduces a developer-facing serverless abstraction called the request-job-task model. In this model, a *request* is an external trigger (e.g., a user-sent HTTP call). A *job* (or multiple *jobs*) matching the request type (e.g., chat, fine-tuning) handles the request. A task is a fine-grained operation within a job (e.g., prefill task, decode task). Users interact with DEEPSERVE by sending HTTP requests, each triggering one or more internal jobs. A job, in turn, may spawn multiple tasks. For example, a fine-tuning request triggers multiple jobs, such as preprocessing, training, and evaluation. A chat request triggers a single serving job. In the chat case, if executed on a PD-colocated engine [22], the job generates one task. If executed in a prefill-decode-disaggregated setup [33], the job generates two tasks: one for the prefill instance and the other for the decode instance. This abstraction allows DEEPSERVE to (i) scale AI workloads over shared infrastructure, (ii) consolidate post-training and serving on the same cluster, and (iii) simplify distributed LLM serving.

More specifically, DEEPSERVE's serverless abstraction consists of three core components: job executors, task executors, and a cluster manager. The Job Executor (JE) processes incoming requests, decomposes them into tasks, and assigns

tasks to available task executors for execution. The task executor (TE) is responsible for executing the tasks. DEEPSERVE deploys JEs and TEs dedicated to post-training, agent serving, and model serving. The cluster manager is a highly available system that oversees and scales all JEs and TEs. It includes centralized master modules for both JEs and TEs, each monitoring their health. Due to space limits, this paper focuses on the model-serving aspect.

Efficient LLM Serving. We design FLOWSERVE (§4) as an efficient serving engine used by each model-serving TE. To minimize interference between the prefill and decode stages of LLMs, FLOWSERVE adopts a PD-disaggregated serving paradigm for both dense and sparse models [11, 51].

Scheduling. To support increasingly stateful and disaggregated serving, we propose distributed scheduling algorithms (§5) that run on model-serving JEs. Model-serving TEs running the same model and serving mode (e.g., prefill-only, decode-only) are organized into a TE group. JEs dispatch requests to appropriate TEs within these groups based on our scheduling algorithms.

Fast Scaling. DEEPSERVE uses fast scaling (§6) to quickly adjust to changing online workloads. Key techniques include reserving pre-warmed Pods and TEs, pre-loading models into DRAM, and using high-speed NPU-to-NPU links for faster model loading. These techniques greatly reduce code-start time and allow DEEPSERVE to scale up to 64 instances in parallel within seconds.

In the rest of the paper, we discuss **serving** in §4, **scheduling** in §5, and **scaling** in §6.

4 FLOWSERVE: An Efficient Serving Engine

This section presents FLOWSERVE, our in-house serving engine for LLMs, Large Multimodal Models (LMM), and embedding models.

4.1 Overview

We build FLOWSERVE with the following three main goals[1]. First, we want to create a high-throughput serving engine that works as a single TE. Second, we aim for fast iteration, given that serving is a fast-moving field. Third, we want FLOWSERVE to be Ascend-NPU-native, able to work with different generations of the Ascend cluster—from regular scaled-out servers to scaled-up SuperPod, and eventually fully disaggregated data centers [38].

To achieve these goals, we build FLOWSERVE based on three guiding principles:

- **Microkernel-Inspired Design.** We apply the core principles of microkernel architecture by decoupling serving-engine functionalities into modular components. This modularity enables independent scaling, independent evolution, and asynchronous operations, thereby enhancing throughput.
- **NPU-Centric Execution.** We aim to keep NPUs busy all the time, reducing delays caused by waiting for other resources, such as CPU, DRAM, storage, and network.
- **SPMD-based Design.** We use the Single-Program-Multiple-Data (SPMD [5]) design to enable efficient parallel processing and scaling across multiple NPUs, a design shared by vLLM [22].

Figure 2 presents FLOWSERVE's overall architecture. Each model-serving TE has a *TE-shell* and a FLOWSERVE engine. The *TE-shell* is an infrastructure module consisting of predefined modules such as scaling, health reporting, and a few customized modules that serve as context caching handlers.

The FLOWSERVE engine has six core functionalities: tokenization (with a tokenizer), model execution, scheduler, memory management, caching management, and networking management. The tokenizer is an independent module that can scale on its own. Adhering to the SPMD design principle, FLOWSERVE follows a master-executor architecture: the master oversees scheduling, caching, networking decisions, while per-NPU executors carry out these decisions on their respective NPUs. We map the remaining functionalities, excluding the tokenization, onto this master-executor design as follows:

- **Scheduling and Model Execution (§4.2).** The master scheduler assigns the next batch, and each executor runs the model's forward pass on this batch. The master broadcasts requests to all executors when initiating a batch.
- **Caching and Memory Management (§4.3).** We build a Relational Tensor Cache (RTC) to unify the management of both caching and memory allocation. The master maintains indexing structures such as a prefix tree [50], while each executor handles memory allocation.
- **Networking Management (§4.4).** We build Distributed Flow (DistFlow) to transfer tensors between model-serving TEs. DistFlow runs on each executor, providing memory-transfer APIs for both the model generator and scheduler, supporting multiple transfer backends and Ascend cluster generations.

We implement FLOWSERVE primarily in Python, with RTC and DistFlow implemented in C++. Although having design principles is important, translating them into an efficient implementation is a significant challenge. Figure 3 shows the decoding performance of FLOWSERVE in three versions over a period of three months. From v1 to v2, we introduced asynchronous scheduling and IPC optimization, which resulted in more than 2x improvements when the TPOT SLO was set to 50ms. From v2 to v3, our optimizations focused on data structures, sampling, and so forth; these optimizations resulted in roughly 20% improvement.

[1] These goals were established in late 2023 and have guided the design ever since.

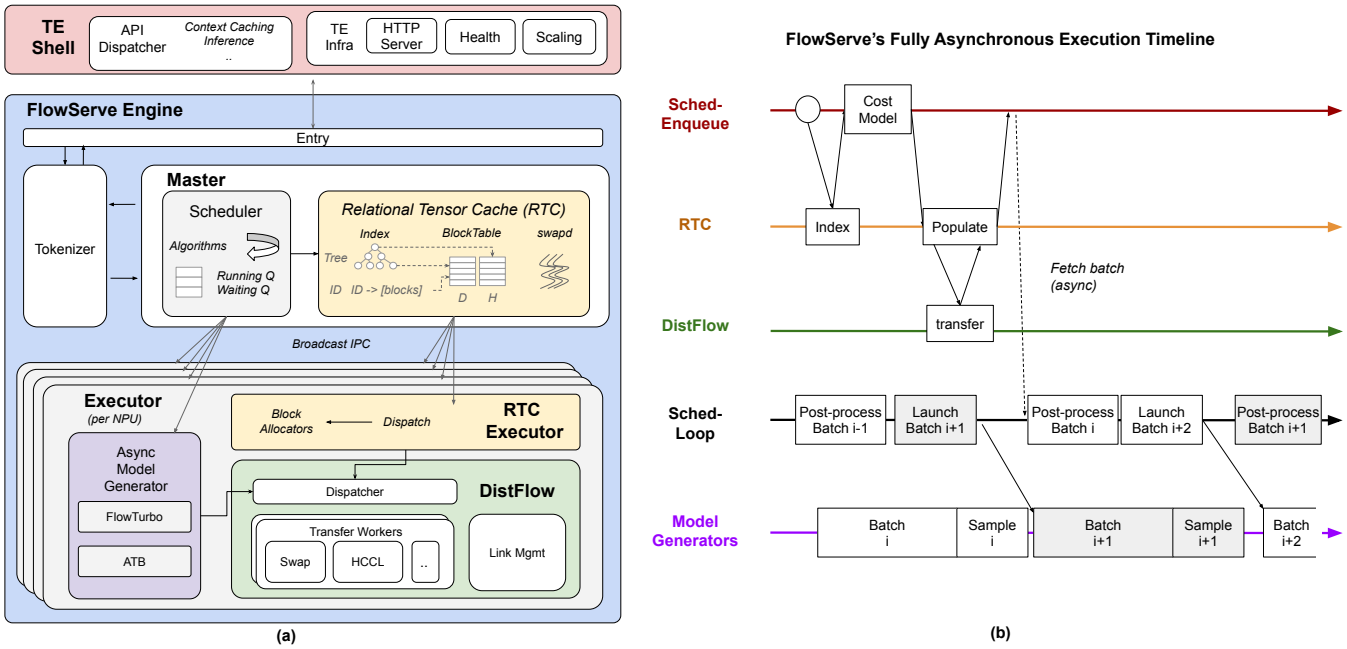


Figure 2: **FLOWSERVE Architecture and Asynchronous Execution Timeline.** (a) The FLOWSERVE engine can be deployed without the TE-shell. (b) The FLOWSERVE's model generator supports multiple backends. FlowTurbo is a torch-based dynamic graph. Ascend-Transformer-Boost (ATB) is a C++-based static graph [16].

4.2 Scheduling

The scheduler is centralized in the master module. We choose this approach over the distributed schedulers (used by SGLang [50]) due to its simplicity, albeit at the cost of frequent Inter-Process Communications (IPCs). The scheduler supports asynchronous KV cache prefetching enabled by RTC and DistFlow (described below) and asynchronous execution, a design shared by vLLM [22], SGLang [50], and NanoFlow [52]. We now describe the execution timeline in the right part of Figure 2.

Asynchronous KV Cache Prefetch. A sched-enqueue thread handles tokenized requests from the tokenizer with the following three steps. First, the thread calls the RTC `match` API to check for any preserved KV cache. The returned info tells whether a prompt prefix or ID has cached KV and the cached KV's location (e.g., in the NPU or swapped to tiered storage). Second, if there is cached KV but a portion of it is not in the NPU, the scheduler runs a cost model to decide whether reusing the cache is beneficial. If the cost model suggests that reusing the cache can improve performance, the scheduler calls the `populate` API to request RTC to fetch the cache into the NPU. This step is done asynchronously. RTC will call DistFlow to read KV from tiered storage or other TEs. Once RTC completes the cache population, it notifies the sched-enqueue thread, which marks the request as ready. The sched-loop will pick it up during the next scheduling cycle.

Asynchronous Execution. The asynchronous execution design aligns with our NPU-centric principle of fully utilizing the NPU. It is similar to the new zero-overhead scheduler introduced in SGLang-v0.4.0 [50]. The key observation behind this design is that scheduling decisions do not depend on the actual token IDs generated by the model, but rather on the number of tokens to be processed in each run. In typical decoding scenarios, where one token is generated per sequence per decoding step, the scheduler can predict the required resources for the next run before the current one completes. Doing so allows the scheduler to operate in a separate thread, preparing the necessary inputs for the model generator beforehand. By running the scheduler in parallel with the model execution, we eliminate unnecessary CPU wait time, ensuring that the NPU remains busy.

Pipeline Parallelism (PP). We optimize our scheduler for PP by running a centralized scheduler at the first stage of PP; other stages accept only requests from previous stages. This design enables FLOWSERVE to manage all micro-batches in a unified way. This approach has two main benefits. (1) Memory resources are managed in one place, making it easy to preempt sequences across micro-batches. (2) With chunked prefill enabled, the scheduler distributes chunks across consecutive micro-batches, rather than sticking to just one micro-batch [1]. Doing so helps reduce Time-To-First-Token (TTFT) by at least 20%.

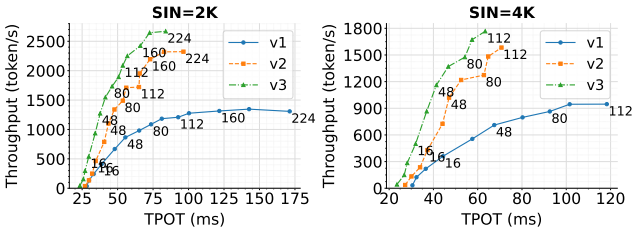


Figure 3: **FLOWSERVE Offline Serving Perf.** We run a 34B model with TP=4. The left has a prefill length of 2K, and the right is 4K. We run 256 decoding steps and report the average TPOT and decoding throughput.

Table 1: The Core APIs of Relational Tensor Cache.

API	Description
`MatchByPrefixToken`	Find preserved KV cache by tokens
`MatchByID`	Find preserved KV cache by ID
`Populate`	Fetch preserved KV cache into NPU
`QueryPopulate`	Check populate status
`AllocBlocks`	Alloc blocks for prefill
`AppendBlock`	Alloc block for decode
`Copy`	Copy blocks from NPU to DRAM
`Free`	Free blocks

4.3 Caching

We build a Relational Tensor Cache (RTC) to integrate prefix caching [50] and position-independent caching [14, 45] into FLOWSERVE. The core function of RTC is to manage the *relationship* between tensors, primarily the KV cache.

Abstraction. Table 1 lists the main APIs of RTC. RTC provides two main sets of APIs. The first set is for managing basic blocks, such as `AllocBlocks` and `AppendBlock`, which are used by prefill and decode requests, respectively. The second set is for managing KV caching. Notably, RTC includes various match APIs, each supporting different indexing mechanisms, such as prefix-token-based or explicit ID-based matching. FLOWSERVE's implicit caching uses the prefix-token-based mechanism, while the explicit ID-based mechanism is used by DEEPSERVE's explicit context caching endpoint. RTC also offers a novel `populate` API. When invoked, RTC fetches the specified KV cache in the given address range into local NPUs.

Design. RTC follows the master-executor architecture of FLOWSERVE. The master module manages the RTC module, making decisions about allocation and data movement, while each executor runs an RTC executor to perform these actions. Internally, RTC includes a traditional block table, originally proposed by vLLM [22], for managing data blocks. Additionally, RTC employs a hybrid indexing layer that combines radix-tree indexing [50] with ID-based indexing. In our current design, each index node can point to data stored either in the NPU or in local DRAM. We are also adapting RTC for SuperPod to leverage its global shared memory. Finally, the RTC master runs multiple background threads to handle tasks such as background swapping and prefetching.

4.4 Networking

Emerging distributed serving techniques—such as disaggregated prefill-and-decode [12, 33, 51] and disaggregated attention-and-experts [7, 32]—require new communication primitives that go beyond traditional collectives. `DistFlow` is designed to meet this demand. `DistFlow` specializes in peer-to-peer and many-to-many tensor transfers across heterogeneous memory. It supports transfers both within a single TE across tiered storage and between distributed TEs.

Abstraction. `DistFlow` provides a simple but expressive interface. Its control-plane APIs include `LinkCluster` for establishing peer groups, and its primary data-plane API is `transfer(srcInfo, dstInfo)`. Users are required to specify exact source and destination memory buffers; `DistFlow` operates on raw memory addresses rather than higher-level block abstractions. In FLOWSERVE, this API is used by the centralized scheduler, each executor's model generator, and the RTC executor.

Design. `DistFlow` follows key principles from high-performance RPC systems [19, 20, 29, 42]. It employs scalable threading models to avoid synchronization bottlenecks and supports multiple backends to adapt to heterogeneous network fabrics. In scaled-out Ascend clusters, `DistFlow` uses the Huawei Collective Communication Library (HCCL) peer-to-peer API as its transfer backend. In scaled-up Ascend SuperPods, where CPUs and NPUs share a unified memory space, it leverages NPU memory copy primitives for direct, high-bandwidth tensor transfers.

4.5 Disaggregated Serving

Serving is increasingly disaggregated. In DEEPSERVE, we define two levels of disaggregation:

- **Task-level**. This level refers to disaggregating prefill and decode into separate TEs. Our implementation is similar to previous work [12, 33, 51]: FLOWSERVE in the prefill TE sends prefilled KV cache to decoding TE either by-req or by-layer using DistFlow's transfer API. Figure 4 shows an online serving test comparing PD-disaggregated and PD-colocated using an internal trace. Disaggregation greatly improves throughput under certain SLOs and lowers TPOT with the same throughput.

- **Operator-level**. This level refers to disaggregating attention and experts into separate TEs. We take inspiration from pioneering work in this space [7, 32]. We are actively working toward deploying it over SuperPod.

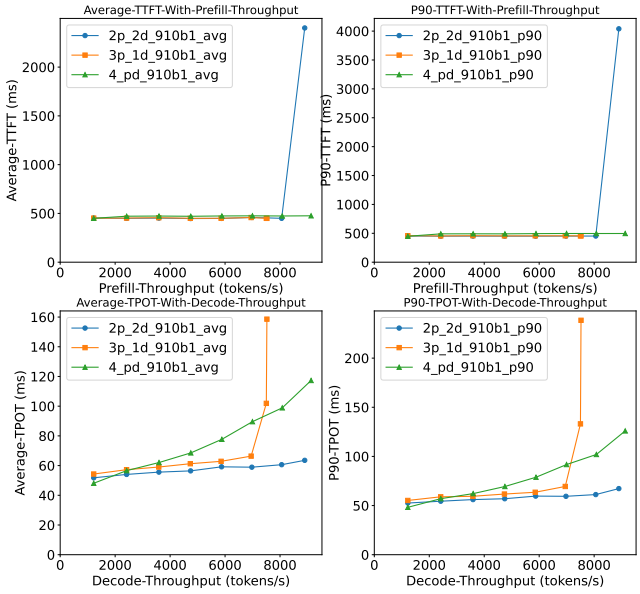


Figure 4: **FLOWSERVE Online Serving Perf.** We run a 34B model with TP=4 using an internal trace (roughly 2K input with 200 output). We test three setups: (1) PD-disaggregated with two prefill and two decode, (2) PD-disaggregated with two prefill and one decode, and (3) four PD-colocated. We vary RPS from 0.2 to 1.2 in a step of 0.2.

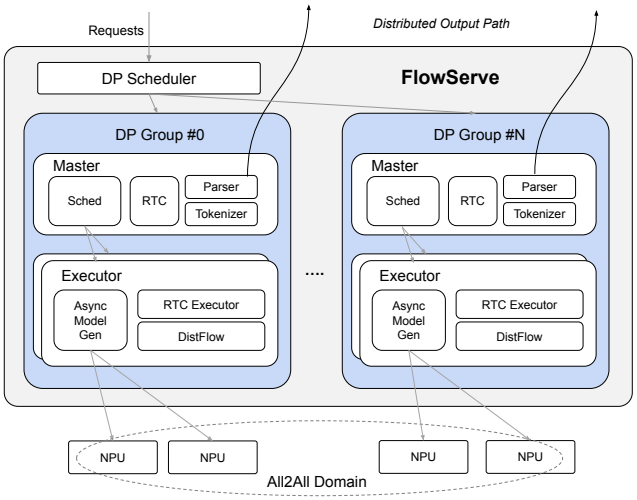


Figure 5: **Adapting FLOWSERVE for SuperPod.** A single FLOWSERVE TE can manage up to tens of servers and hundreds of NPU chips.

4.6 Serving at SuperPod-Scale

This section outlines the key changes (to FLOWSERVE) that enable it to run at SuperPod scale. We begin by examining the workload shift that motivates these changes, followed by a description of the corresponding architectural changes.

From a workload perspective, large Mixture-of-Experts (MoE) models—such as DeepSeek-V3/R1 [25] and Pangu-Ultra-MoE [41]—are a natural fit for SuperPod-scale infrastructure. First, MoE models require efficient all-to-all communication before and after the expert modules, benefiting significantly from a large-scaled-up domain with high-bandwidth interconnects. Second, MoE experts achieve high compute efficiency at only large batch sizes. Enabling such batches necessitates data parallelism in the attention layers (DP Attention) instead of traditional tensor parallelism. Under DP Attention, colocating the prefill and decode stages leads to severe interference [12,25]. As a result, effective deployment requires disaggregating prefill and decode (PD disaggregation), which further depends on fast, low-latency interconnects. The CLOUDMATRIX384 SuperPod provides a significantly larger scaled-up domain with a global shared memory address space and enables low-latency, high-bandwidth data transfers between any CPU or NPU in the system, making it ideally suited for serving large-scale MoE models.

Motivated by these requirements, we introduce two major architectural changes to scale FLOWSERVE to hundreds of NPU chips, as summarized in Figure 5. First, to avoid scheduling bottlenecks, we partition the system into multiple parallel DP groups. Each group includes a full set of components—its own scheduler, RTC, DistFlow, and executors. A centralized DP scheduler dispatches requests across DP groups using either a round-robin or greedy strategy. To eliminate bottlenecks on the return path, each DP group independently returns tokens to the frontends. Second, to support large-scale PD disaggregation, we extend `DistFlow` to support M:N connections. Specifically, we establish transfer channels between all prefill and decode DP groups.

5 Distributed Scheduling

This section discusses DEEPSERVE's distributed LLM request scheduling policies.

5.1 Overview

In DEEPSERVE, distributed scheduling for LLM requests faces three new challenges. The first challenge involves locality and states. Before prefix caching was introduced into FLOWSERVE, distributed scheduling was straightforward, as all TEs could be treated as stateless, and scheduling was based solely on load. However, with the introduction of prefix caching, TEs have become stateful, and the goal is to schedule requests where KV cache reuse is possible, leading to the need for locality-aware scheduling. Achieving locality-aware scheduling is non-trivial. Selecting the best TE for

a given request requires balancing KV cache reuse while avoiding overloading the TE that holds the most cache. The second challenge comes from disaggregated serving. Distributed scheduling was easier before PD-disaggregation was added to FLOWSERVE because all TEs were the same. However, with PD disaggregation, choosing the best TE for a request becomes more complex, as it is unclear whether PD-disaggregated TEs or a PD-colocated TE will perform better. The last challenge arises from the coexistence of both prefix caching and disaggregation in DEEPSERVE, requiring any practical algorithm to consider both factors.

DEEPSERVE uses the following designs in response to these challenges: a locality-aware scheduling algorithm (§5.2), a PD-aware scheduling algorithm (§5.3), and a combined scheduling algorithm (§5.4) as listed in Algorithm 1.

5.2 Locality-aware Scheduling

In this section, we aim to answer one question:

> Given a request and a set of TEs with associated KV cache residency information, how can we select the TE that maximizes KV cache reuse to efficiently serve the request?

To address this question, we implement a prompt-tree-based, locality-aware scheduling policy in the `select_tes_prefix_match` function, as shown in Algorithm 1. Similar to Preble [39], SGLang [50], and MemServe [11], the distributed scheduler in JE maintains a global prompt tree for each type of TE, while each TE also maintains a local prompt tree that shares an index with its corresponding global tree. When a request arrives at the JE, it matches the prompt tokens against the global prompt trees and selects the TE that has the longest common prefix, indicating the largest preserved historical KV cache.

5.3 PD-aware Scheduling

In this section, we aim to answer one question:

> Given a request and a set of TEs with both PD-disaggregated and PD-colocated instances, how can we select the most appropriate TE type that best aligns with the characteristics of the request?

We first study the performance comparison between PD-disaggregated and PD-colocated across various dimensions in a heatmap. We then discuss converting such a heatmap into a practical scheduling algorithm.

5.3.1 Study

We first run tests to compare PD-disaggregated and PD-colocated TEs (with chunked prefill). Figure 6 presents the results (see caption for detailed setup).

The heat map shows that PD-disaggregated and PD-colocated TEs divide the request space into distinct regions. We find three key observations. First, the PD-disaggregated setup performs better for requests with longer prompts and shorter decode. Additionally, as prefill length increases, its advantage becomes more pronounced for requests with longer decode lengths. Second, PD-disaggregated TEs provide a larger performance advantage over PD-colocated TEs (dark read) than the reverse (light blue), suggesting that a correct choice of PD-disaggregated TE leads to significant performance gains, while an incorrect choice results in minimal loss. Finally, the advantage of PD-disaggregated and PD-colocated TEs remains consistent across different RPS values.

5.3.2 Algorithm

The preceding study is conducted in a controlled environment, and applying it to a practical scheduling algorithm is challenging due to the dynamic nature of RPS and the uncertainty of decode length at the time of scheduling. There are multiple potential approaches to convert the study into an algorithm.

We propose a simple policy called `select-tes-PD-heatmap`, described in Algorithm 1. The policy works as follows. First, we combine the heat maps across all RPS values through element-wise addition. Given the stability of the heat map, over 80% of the squares consistently show either positive or negative values across all RPSs, while the remaining 20% are uncertain. Second, we predict the decode length for an incoming request using a predict model with 84.9% accuracy to balance prediction precision and overhead. For more details, please refer to the following discussion section. Third, we identify the corresponding square on the heat map based on the combined heat map, the prefill length, and the predicted decode length. If the value is positive, we select PD-disaggregated TEs; if negative, we select PD-colocated TEs.

We perform a microbenchmark study shown in Figure 7. We make three key observations. First, under certain RPS levels (e.g., 10 reqs/s), the PD-aware scheduling policy outperforms RR. Second, at low RPS levels, the performance of PD-aware matches that of RR. The reason is that under low RPS, interference between prefill and decode operations within PD-colocated TEs remains negligible, and PD-disaggregated TEs do not offer significant advantages. Third, at very high RPS levels, PD-aware performs worse than RR. The reason is that PD-disaggregated TEs, with the same resources (e.g., two cards), are more prone to overloading. However, even when overloaded, PD-aware scheduling does not exhibit significant performance degradation compared to RR.

5.3.3 Discussion of the Predict Model

Accurately predicting decode lengths with minimal overhead remains a challenging research problem. We adopt

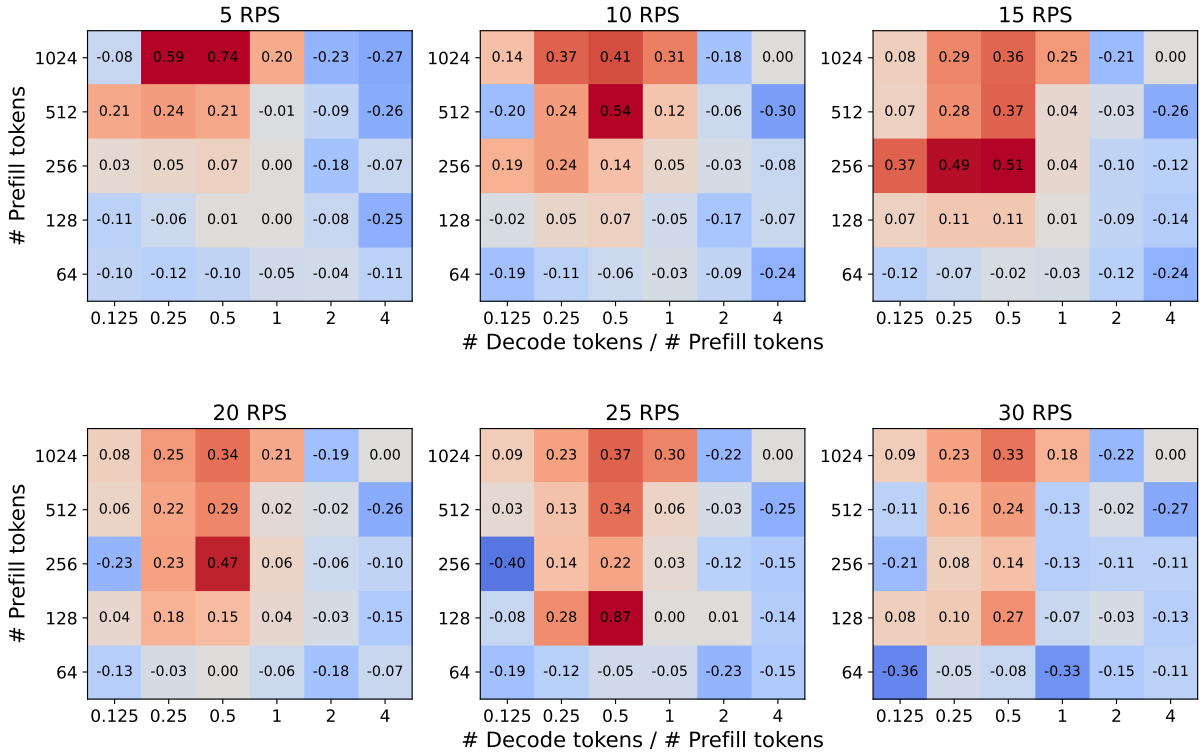


Figure 6: **Comparing the Performance of PD-disaggregated and PD-colocated (with chunked prefill) using Heatmap.** The y-axis represents the prefill length, and the x-axis shows the ratio of decode length to prefill length. For each combination of prefill and decode lengths, we execute a batch of identical requests at a fixed RPS on both PD-disaggregated and PD-colocated TEs. The heatmap cells display the difference in JCT between the two setups, computed as the ratio of JCT for the PD-colocated TE to the PD-disaggregated TE, minus one. A positive value indicates better performance of the PD-disaggregated TE, while a negative value suggests that the PD-colocated TE is more efficient. This figure uses a 34B model with TP=4.

the approach proposed in TetriServe [12], which employs a lightweight LLM-based classification model—referred to as the predict model—to classify decode lengths into fixed-size buckets, assuming execution by a specified target LLM.

We predict length ranges rather than exact token counts for two reasons. First, inference parameters such as temperature and top-p [3] introduce substantial variability in outputs, making accurate prediction difficult. Second, our goal is to support scheduling decisions, for which approximate length ranges are sufficient (Figure 6).

The training process for the predict model consists of the following three steps. First, we construct a prompt-only training dataset using public sources, a target model (e.g., LLaMA 70B), and a predict model (e.g., OPT 125M [17]). Second, we query the target model with the prompts to generate responses, which we discretize into fixed-size length buckets to serve as ground truth. Third, we split the dataset into training and evaluation sets, and train the predict model accordingly.

In our experiments, we use a bucket granularity of 128 tokens and achieve 84.9% accuracy. Since our focus lies in using prediction to inform scheduling, we leave further accuracy improvements to future work.

5.4 Combined Algorithm

We propose a combined algorithm that integrates load-aware, locality-aware, and PD-aware scheduling. The scheduling algorithm, outlined in Algorithm 1, relies on three core function calls: `locality_aware`, `PD_aware`, and `load_aware`. These functions progressively refine the TE group, narrowing it down from the entire set to a single TE based on the request's characteristics and the underlying TEs.

The scheduling process proceeds as follows. First, a subgroup of TEs is selected by choosing a specific type, such as PD-colocated or PD-disaggregated TEs, based on the request's length and the heatmap of the TE group (Section 5.3). Second, once a subgroup of TEs is identified, the selection is further refined. If the load is balanced across the remaining TEs, the algorithm prioritizes a TE with the most reusable KV cache, using tree-based prefix matching (Section 5.2). If

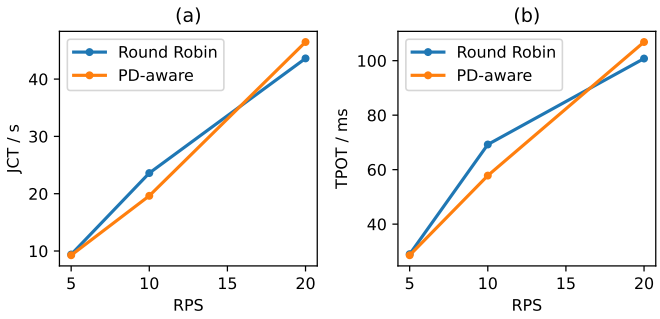


Figure 7: **Study of Distributed Scheduling Algorithm.** We run a 34B model with TP=4, and report JCT / TPOT. We run an internal trace sampled from a code-generation service. The cluster consists of four servers with two PD-colocated TEs and a pair of PD-disaggregated TEs (1P1D).

the load is unbalanced, the algorithm instead prioritizes a TE with the least load to achieve better load distribution.

Algorithm 1: Distributed Scheduling Policy

Input: A new request: req, a TE group: tes
Output: A TE to forward the request to

```
Function dist_sched(req, tes):
    tes ← PD_aware (req, tes);
    if tes.is_load_balanced() then
        tes ← locality_aware (req, tes);
    else
        tes ← load_aware (req, tes);
    return tes;
Function PD_aware(req, tes):
    p_l ← req.get_prefill_length ();
    d_l ← req.get_decode_length ();
    tes ← tes.select_tes_PD_heatmap (p_l, d_l);
    return tes;
Function locality_aware(req, tes):
    tes ← tes.select_tes_prefix_match (req);
    return tes;
```

6 Fast Scaling

In DEEPSERVE, the cluster manager's AUTOSCALER determines when to scale TEs and JEs based on metrics such as load or SLO-violation rates. Fast scaling is critical but particularly challenging for LLM-serving TEs, as it requires loading large model weights onto NPUs. This section describes our optimizations for fast scaling.

Figure 8 shows the autoscaling workflow, which involves five steps, with challenges and solutions summarized in Table 2 and performance breakdown in Figure 9. First, in the *Scaler-Pre* step, DEEPSERVE prepares resources and pod environments for the new TEs. For large models, a TE may span multiple pods on multiple machines. Second, the *TE-Pre-Load* step initializes the FLOWSERVE instance but does not load the model onto the NPU. Third, the *TE-Load* step covers loading model weights onto the NPU. We decouple the second and third steps to allow for pre-warmed TEs, facilitating faster scaling (see §6.1). Fourth, the *TE-Post-Load* step involves engine warm-up and CPU/NPU block allocation. Open-source inference engines such as vLLM [22] rely on warm-up to profile the allocable HBM size for KV caches. We find this step unnecessary in production, because we can offline profile this data and store it in configurations. To address the slowdown of the first request after removing warm-up, we add a dummy message post-startup. Finally, in the *Scaler-Post* step, the new TEs are announced to the cluster, and JEs direct requests to them.

6.1 Pre-warming Pods and TEs

DEEPSERVE maintains two levels of pre-warmed resources. First, at the pod level, DEEPSERVE reserves a small number of pre-warmed pods with basic environment setup. These workload-independent pods are usually managed by the infrastructure layer, such as Kubernetes, and can be shared across services to reduce overhead. Second, on these pre-warmed pods, DEEPSERVE maintains a small pool of pre-warmed TEs to minimize *TE-Pre-Load* time. The *TE-Pre-Load* step includes FLOWSERVE's startup time, which involves loading Python libraries, initializing NPU states, and setting up HCCL cross-NPU interconnections. We optimize this step by approximately 35% for most models using techniques such as late importing and parallel initialization. However, as shown in Figure 9, this step still accounts for the majority of startup time. To further reduce the startup time, we move the initialization out of the critical path by incorporating a TE pre-warming mechanism.

Our TE pre-warming design is carried out in two stages. First, we make the pre-warmed TEs model-agnostic. For example, a TP-8 pre-warmed TE can be adapted to run either a Llama3-70B or a Qwen2-72B model. This stage requires carefully distinguishing between model-specific and model-agnostic parameters. Second, we make the pre-warmed TEs agnostic to parallelism strategies by recognizing that, regardless of TP/PP/SP configurations, all TEs follow a master-SPMD architecture. This stage allows independent pools of pre-warmed SPMD-masters and SPMD-executors, and these pools can be packed on demand.

Table 2: **A Summary of DEEPSERVE's End-to-End Scaling Steps, Challenges, and Solutions.**

ID	Step	Definition	Major Issues	Our Solutions
1	Scaler-Pre	Creating pods to hold the TE.	1. Resource allocation is slow	1. Pre-warmed Pods
2	TE-Pre-Load	Launching the TE w/o model loading	1. Python startup is slow 2. NPU init is slow	1. Pre-warmed TEs
3	TE-Load	Loading the model onto the NPU	1. Model weight is large	1. DRAM pre-loading 2. NPU-fork
4	TE-Post-Load	Preparing TE to serve requests	1. Engine warmup is slow 2. Block alloc is slow	1. Offline profiling 2. Async allocation 3. Dummy req warmup
5	Scaler-Post	From TE ready to serve first request	1. The update of the global TE list is slow	1. Proactive pushing

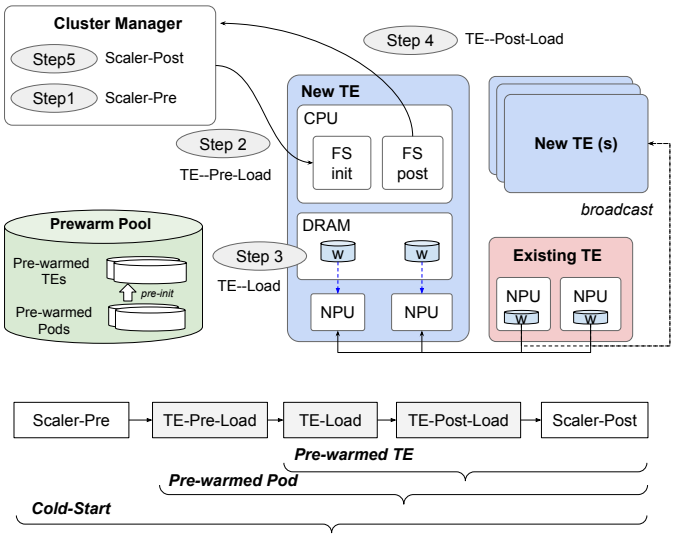


Figure 8: **DEEPSERVE's Scaling Design.** (a) We show TE-Load's two cases: loading from local DRAM (DRAM-hit) and loading from another TE's NPU (NPU-fork). NPU-fork can be either via scale-up or scale-out network link. (c) The bottom shows the timeline for three scaling cases.

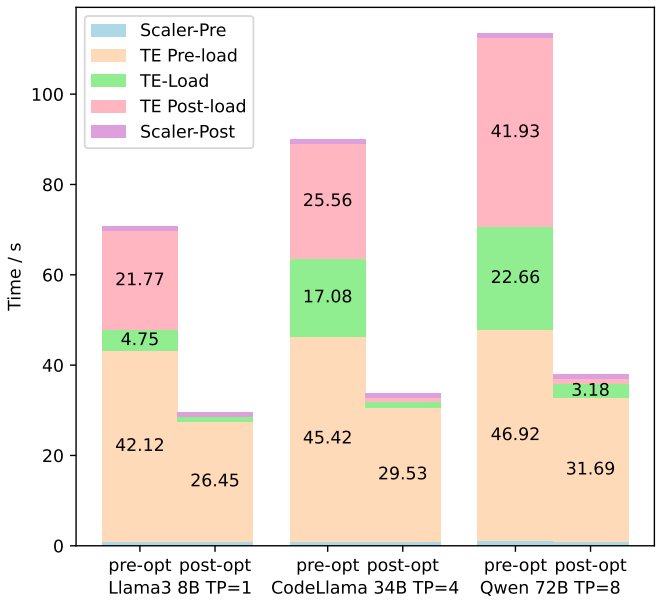


Figure 9: **Scaling E2E Breakdown**. We present the scaling latency both before and after optimizations. Even after optimization, the *TE-Pre-load* step still remains the dominant factor in scaling time, although this latency can be further reduced through pre-warming (§6.1).

6.2 Optimizing Model Loading

DEEPSERVE has two model-loading paths: *local loading* via PCIe from local DRAM or SSD, and *NPU-fork* using high-speed NPU-to-NPU links from a running TE. While NPU-fork is generally faster, it has higher hardware requirements and cannot be used during code start (scaling from 0 TE).

Local loading with pre-warming. We use the safetensors format [36] for model storage. Compared to the native binary format, safetensors reduce serialization costs by storing tensors in contiguous blocks that can be directly mapped into memory and trigger data reads only on page faults. Safetensors also simplify pre-loading into the page cache, reducing read amplification. Instead of loading the entire model file, each TP process loads only the required partition on demand.

To further optimize TE-load performance, we co-design it with our TE pre-warming mechanism. The cluster manager predicts models that are likely to scale and pre-loads them into DRAM page cache using pre-warmed TEs. When scaling is triggered, the master prioritizes selecting pre-warmed TEs with the required model already loaded. On our hardware, each machine has 1.5TB of DRAM, sufficient for pre-loading 10 70B models or 100 7B models.

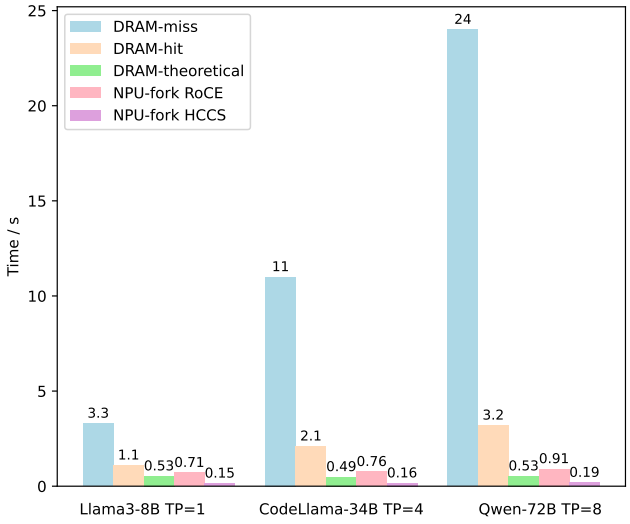


Figure 10: **TE-Load Study.** DRAM-hit means loading from pre-loaded weights in DRAM; DRAM-miss means pre-load miss, thus loading from SSD; DRAM-theoretical is calculated by model weights dividing PCIe bandwidth. NPU-fork has two different links—HCCS and RoCE (see the main text for the differences).

Figure 10 shows TE-load's performance under different cases (NPU-fork, DRAM-hit, and DRAM-miss). DRAM-hit occurs when the cluster manager correctly predicts the models to scale, while DRAM-miss reflects the opposite scenario. We also compute the theoretical model-loading time by dividing the model weights by the PCIe bandwidth. For DRAM-hit, the difference between measured and theoretical time arises from two factors: PyTorch-model tensor initialization (about 0.3s) and PCIe-bandwidth contention. The latter becomes more significant with larger TP ranks due to shared PCIe links. For the three models in Figure 10, the weights loaded by each NPU are roughly the same, but local loading time increases with larger TP ranks due to PCIe-link sharing among NPUs.

NPU-fork. NPU clusters typically feature high-speed NPU-to-NPU links designed for distributed training. In §4.3, we demonstrate how these links are used for efficient inference. We further exploit them in our NPU-fork technique to transfer model weights during fast scaling. When NPU-fork is triggered, the master notifies a running FLOWSERVE TE to connect to a pre-warmed TE, with model weights transferred using the DistFlow module (§4.4).

Figure 10 shows the performance of NPU-fork. On our Ascend hardware, there are two types of links: HCCS (higher bandwidth, smaller scale) and RoCE (lower bandwidth, scalable to thousands of nodes). All evaluations are conducted on cross-node setups. Overall, loading with HCCS is significantly faster than with RoCE, indicating that NPU-fork will benefit from the SuperPod architecture. The model-loading time is similar across the three models, as the weights to be loaded per NPU are roughly the same. NPU-fork experiences less bandwidth contention compared to local loading, as NPU-fork uses different physical links.

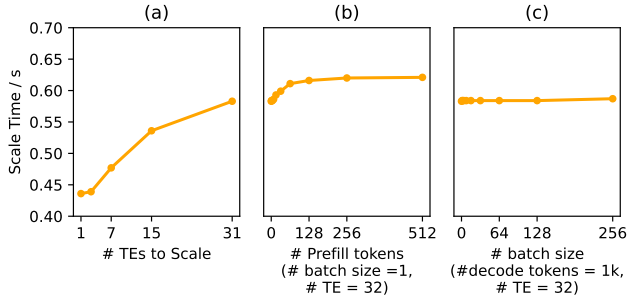


Figure 11: **Scalability and Sensitivity Study of NPU-fork**. We run NPU-fork over a scaled-up network (HCCS) using Llama3-8B-TP1. (a) Scaling multiple TEs in parallel from one running TE. (b) Time for scaling to 32 TEs when the source TE is prefilling sequences of different lengths. (c) Scaling time when the source TE is decoding different batches of sequences, each with a length of 1k tokens.

Figure 11 illustrates potential degradation when multiple expansions (up to 32) occur concurrently or when prefilling and decoding overlap the expansions. NPU-fork can scale to a large number of TEs by transmitting model weights simultaneously to multiple TEs using the `broadcast` API in the HCCL collective communication library. We also measure resource-contention sensitivity when the source TE handles prefill and decode requests. Since the NPU has dedicated AICPUs for data transfer, contention is limited.

7 Discussion

We now discuss the broader applicability of DEEPSERVE across hardware platforms and workloads.

NPU-agnostic vs. NPU-specific features. Although DEEPSERVE is built on Ascend NPU chips at Huawei Cloud, its high-level architecture and core design are largely hardware-agnostic. For example, the "NPU-fork" mechanism used for auto-scaling leverages Ascend's HCCS interconnect for rapid scaling. However, similar techniques can be applied to other hardware platforms with fast inter-chip links, such as NVIDIA GPUs with NVLink. Most NPU-specific features involve low-level computation and networking primitives. For instance, in the CloudMatrix384 SuperPod, memory across all NPUs is accessible, allowing data transfers between any two memory locations—a capability not available in regular scaled-out AI servers.

Generalization to non-LLMs. While the paper primarily focuses on LLMs, most components generalize to other model types, such as embedding models and multimodal-understanding models. In fact, all these models run within the FLOWSERVE engine similar to vLLM.

Fault-Recovery Mechanism. In the event of a TE or JE failure, DEEPSERVE reboots the affected component and redirects requests to redundant counterparts. The reboot process ensures recovery within 5 minutes. RTC maintains soft states, which can be recomputed if lost and are append-only. Therefore, we do not implement complex consistency protocols to avoid unnecessary performance overhead for RTC. However, in production, we have not observed a significant impact on service quality, primarily due to (1) system-level redundancy and (2) the SuperPod architecture, which ensures fast and uniform NPU-to-NPU communication regardless of physical location, enabling seamless substitution of failed NPUs.

8 Related Work

In this section, we review related work in four key areas: serverless infrastructure, serving engines, scheduling, and scaling for large language model (LLM) workloads.

Serverless Infrastructure. Extensive research has focused on optimizing LLM serving in serverless architectures [2, 8, 23, 35, 44, 47, 48]. Industry solutions, such as AWS SageMaker and Azure ML [28], deliver cloud-native solutions optimized for deploying, managing, and scaling model inference, tailored specifically to meet enterprise needs. Recent open-source systems, including KServe [21], AIBrix [6], NVIDIA Dynamo [30], and LeaderWorkerSet (LWS) [26], offer practical implementations with varying degrees of support for cloud deployment, auto-scaling, multi-model, multi-node execution, and so forth. DEEPSERVE offers similar functionality but differs in two key aspects. First, while existing systems primarily target GPU clusters, DEEPSERVE is designed for NPU-based clusters. Second, to the best of our knowledge, DEEPSERVE is the first publicly described platform that introduces request-job-task abstraction, integrating diverse AI workloads from post-training to serving.

Serving. Serving is a rapidly evolving field, with many open-source engines such as FLOWSERVE emerging. vLLM [22] pioneers PagedAttention for higher throughput. SGLang [50] uses RadixAttention for reusing cache. LightLLM [13] adopts asynchronous execution for improved throughput. We believe that all serving engines will eventually adopt similar architectures to optimize AI chip efficiency, with differences mainly in hardware support, features, and programming languages. Networking-wise, the Mooncake Transfer Engine (MTE) [34] is closest to FLOWSERVE's DistFlow. Both systems support multiple backends and efficient threading models, following seminal lines of work [19, 20, 42]. The key difference lies in backend implementation: MTE uses RDMA by default, while FLOWSERVE uses HCCL's peer-to-peer APIs on regular scaled-out servers and memory-copy primitives on the CLOUDMATRIX384 SuperPod.

Scheduling. Efficient scheduling is critical for improving the performance of serving systems [1, 9, 11, 22, 33, 34, 43, 46, 50, 51]. For example, at the local layer, Orca [46] proposes iterative-level scheduling to reduce bubbles. Sarathi [1] proposes chunked-prefill to overcome suboptimal prefill processing. FastServe [43] utilizes a multi-level priority feedback queue to minimize JCT. At the global layer, MuxServe [9] formulates a multiplexing problem and proposes a placement algorithm and adaptive batch scheduling strategy to identify optimal colocations in LLM serving. MemServe [11] prioritizes locality by directing requests to instances with the highest cache-hit rate. Our work introduces a novel PD-aware scheduling policy to determine whether a request should be processed by PD-disaggregated or PD-colocated TEs. Additionally, our approach integrates PD awareness with locality- and load-aware scheduling, providing a comprehensive solution for optimizing resource utilization and throughput across heterogeneous TE configurations.

Scaling Optimizations. Scaling serving instances dynamically is a major challenge, mainly because these instances are large and continue growing as modern LLMs' size increases. Recent work on model autoscaling [4, 18, 27, 49] has focused on improving resource use and reducing scaling delays. For example, SpotServe [27] and Llumnix [40] speed up scaling by making task migration between instances cheaper. BlitzScale [49] improves throughput when loading parameters, reduces delays in handling burst requests, and improves the overall service. Our NPU-fork mechanism is similar to BlitzScale [49] but differs in the underlying network fabrics.

9 Conclusion

We have presented DEEPSERVE, a serverless AI platform developed at Huawei Cloud. We have described its serverless abstraction and infrastructure, which enables efficient management of AI workloads. We have also outlined the architecture of FLOWSERVE, our in-house serving engine, detailing its design principles and core components. Additionally, we have explored distributed scheduling policies across a heterogeneous pool of serving instances. Finally, we have provided an end-to-end analysis of fast scaling, highlighting the techniques that allow DEEPSERVE to quickly adjust to fluctuating workloads.

Acknowledgment

This work was partially supported by National Natural Science Foundation of China under Grant No. 92464301. We would also like to thank our shepherd Yue Cheng and other anonymous reviewers for their insightful comments and suggestions, which greatly help improve the quality of this paper.

References

[1] Amey Agrawal, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav S Gulavani, and Ramachandran Ramjee. SARATHI: Efficient LLM inference by piggybacking decodes with chunked prefills. *CoRR*, 2023.

[2] Ahsan Ali, Riccardo Pinciroli, Feng Yan, and Evgenia Smirni. Batch: machine learning inference serving on serverless platforms with adaptive batching. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15, 2020.

[3] AWS Bedrock. AWS Bedrock.

https://docs.aws.amazon.com/bedrock/latest/userguide/inference-parameters.html.

[4] Zhihao Bai, Zhen Zhang, Yibo Zhu, and Xin Jin. PipeSwitch: Fast pipelined context switching for deep learning applications. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation*, pages 499–514, 2020.

[5] Paul Barham, Aakanksha Chowdhery, Jeff Dean, Sanjay Ghemawat, Steven Hand, Daniel Hurt, Michael Isard, Hyeontaek Lim, Ruoming Pang, Sudip Roy, Brennan Saeta, Parker Schuh, Ryan Sepassi, Laurent Shafey, Chandu Thekkath, and Yonghui Wu. Pathways: Asynchronous distributed dataflow for ML. In *Proceedings of the Machine Learning and Systems*, pages 430–449, 2022.

[6] ByteDance. AIBrix.

https://github.com/vllm-project/aibrix.

[7] Shaoyuan Chen, Yutong Lin, Mingxing Zhang, and Yongwei Wu. Efficient and economic large language model inference with attention offloading. *CoRR*, 2024.

[8] Seungbeom Choi, Sunho Lee, Yeonjae Kim, Jongse Park, Youngjin Kwon, and Jaehyuk Huh. Serving heterogeneous machine learning models on multi-GPU servers with spatio-temporal sharing. In *Proceedings of the 2022 USENIX Annual Technical Conference*, pages 199–216, 2022.

[9] Jiangfei Duan, Runyu Lu, Haojie Duanmu, Xiuhong Li, Xingcheng Zhang, Dahua Lin, Ion Stoica, and Hao Zhang. MuxServe: Flexible spatial-temporal multiplexing for multiple LLM serving. In *Proceedings of the 41st International Conference on Machine Learning*, pages 11905–11917, 2024.

[10] Yao Fu, Leyang Xue, Yeqi Huang, Andrei-Octavian Brabete, Dmitrii Ustiugov, Yuvraj Patel, and Luo Mai. ServerlessLLM: Low-latency serverless inference for large language models. In *Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation*, pages 135–153, 2024.

[11] Cunchen Hu, Heyang Huang, Junhao Hu, Jiang Xu, Xusheng Chen, Tao Xie, Chenxi Wang, Sa Wang, Yungang Bao, Ninghui Sun, et al. Memserve: Context caching for disaggregated LLM serving with elastic memory pool. *CoRR*, 2024.

[12] Cunchen Hu, Heyang Huang, Liangliang Xu, Xusheng Chen, Jiang Xu, Shuang Chen, Hao Feng, Chenxi Wang, Sa Wang, Yungang Bao, et al. Inference without interference: Disaggregate LLM inference for mixed downstream workloads. *CoRR*, 2024.

[13] Jiawei Hu, Hong Jia, Mahbub Hassan, Lina Yao, Brano Kusy, and Wen Hu. LightLLM: A versatile large language model for predictive light sensing. In *Proceedings of the 23rd ACM Conference on Embedded Networked Sensor Systems*, pages 158–171, 2025.

[14] Junhao Hu, Wenrui Huang, Haoyi Wang, Weidong Wang, Tiancheng Hu, Qin Zhang, Hao Feng, Xusheng Chen, Yizhou Shan, and Tao Xie. EPIC: efficient position-independent caching for serving large language models. In *Proceedings of the 42nd International Conference on Machine Learning*, 2025.

[15] Junhao Hu, Wenrui Huang, Weidong Wang, Zhenwen Li, Tiancheng Hu, Zhixia Liu, Xusheng Chen, Tao Xie, and Yizhou Shan. RaaS: Reasoning-aware attention sparsity for efficient llm reasoning. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics*, 2024.

[16] Huawei. Mindie atb models.

https://www.hiascend.com/document/detail/zh/mindie/10RC2/mindiellm/llmdev/mindie_llm0004.html, 2025. Accessed: 2025-01-14.

[17] HugginFace. HugginFace.

https://huggingface.co/docs/transformers/model_doc/opt#transformers.OPTForSequenceClassification.

[18] Jinwoo Jeong, Seungsu Baek, and Jeongseob Ahn. Fast and efficient model serving using multi-GPUs with direct-host-access. In *Proceedings of the 18th European Conference on Computer Systems*, pages 249–265, 2023.

[19] Anuj Kalia, Michael Kaminsky, and David G. Andersen. FaSST: Fast, scalable and simple distributed transactions with two-sided (RDMA) datagram RPCs. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation*, pages 185–201, 2016.

[20] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Datacenter RPCs can be general and fast. In *Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation*, pages 1–16, 2019.

[21] KServe. KServe.

https://github.com/kserve/kserve.

[22] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with PagedAttention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 611–626, 2023.

[23] Jie Li, Laiping Zhao, Yanan Yang, Kunlin Zhan, and Keqiu Li. Tetris: Memory-efficient serverless inference through tensor sharing. In *Proceedings of the 2022 USENIX Annual Technical Conference*, pages 572–488, 2022.

[24] Heng Liao, Jiajin Tu, Jing Xia, and Xiping Zhou. DaVinci: A scalable architecture for neural network computing. In *Proceedings of the 2019 IEEE Hot Chips Symposium*, pages 1–44, 2019.

[25] Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. DeepSeek-v3 technical report. *CoRR*, 2024.

[26] LWS. LWS.

https://github.com/kubernetes-sigs/lws.

[27] Xupeng Miao, Chunan Shi, Jiangfei Duan, Xiaoli Xi, Dahua Lin, Bin Cui, and Zhihao Jia. SpotServe: Serving generative large language models on preemptible instances. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1112–1127, 2024.

[28] MinIO. MinIO.

https://min.io.

[29] Sumit Kumar Monga, Sanidhya Kashyap, and Changwoo Min. Birds of a feather flock together: Scaling RDMA RPCs with flock. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 212–227, 2021.

[30] NVIDIA. Dynamo.

https://github.com/ai-dynamo/dynamo.

[31] OpenAI. Openai, 2025. Accessed: 2025-01-14.

[32] Xiurui Pan, Endian Li, Qiao Li, Shengwen Liang, Yizhou Shan, Ke Zhou, Yingwei Luo, Xiaolin Wang, and Jie Zhang. Instinfer: In-storage attention offloading for cost-effective long-context llm inference. *CoRR*, 2024.

[33] Pratyush Patel, Esha Choukse, Chaojie Zhang, Aashaka Shah, Íñigo Goiri, Saeed Maleki, and Ricardo Bianchini. Splitwise: Efficient generative LLM inference using phase splitting. In *Proceedings of the 51st ACM/IEEE Annual International Symposium on Computer Architecture*, pages 118–132, 2024.

[34] Ruoyu Qin, Zheming Li, Weiran He, Jialei Cui, Feng Ren, Mingxing Zhang, Yongwei Wu, Weimin Zheng, and Xinran Xu. Mooncake: Trading more storage for less computation - A KVCache-centric architecture for serving LLM chatbot. In *Proceedings of the 23rd USENIX Conference on File and Storage Technologies*, pages 155–170, 2025.

[35] Francisco Romero, Qian Li, Neeraja J. Yadwadkar, and Christos Kozyrakis. INFaaS: Automated model-less inference serving. In *Proceedings of the 2021 USENIX Annual Technical Conference*, pages 397–411, 2021.

[36] The safetensors contributors. Safetensors: A safe and efficient format for tensor serialization, 2023. Accessed: 2025-01-14.

[37] SemiAnalysis. Huawei AI CloudMatrix384.

https://semianalysis.com/2025/04/16/huawei-ai-cloudmatrix-384-chinas-answer-to-nvidia-gb200-nvl72/.

[38] Yizhou Shan, Will Lin, Zhiyuan Guo, and Yiying Zhang. Towards a fully disaggregated and programmable data center. In *Proceedings of the 13th ACM SIGOPS Asia-Pacific Workshop on Systems*, 2022.

[39] Vikranth Srivatsa, Zijian He, Reyna Abhyankar, Dongming Li, and Yiying Zhang. Preble: Efficient distributed prompt scheduling for LLM serving. In *Proceedings of the 13th International Conference on Learning Representations*, 2025.

[40] Biao Sun, Ziming Huang, Hanyu Zhao, Wencong Xiao, Xinyi Zhang, Yong Li, and Wei Lin. Llumnix: Dynamic scheduling for large language model serving. In *Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation*, pages 173–191, 2024.

[41] Yehui Tang, Yichun Yin, Yaoyuan Wang, Hang Zhou, Yu Pan, Wei Guo, Ziyang Zhang, Miao Rang, Fangcheng Liu, Naifu Zhang, Binghan Li, Yonghan Dong, Xiaojun Meng, Yasheng Wang, Dong Li, Yin Li, Dandan Tu, Can Chen, Youliang Yan, Fisher Yu, Ruiming Tang, Yunhe Wang, Botian Huang, Bo Wang, Boxiao Liu, Changzheng Zhang, Da Kuang, Fei Liu, Gang Huang, Jiansheng Wei, Jiarui Qin, Jie Ran, Jinpeng Li, Jun Zhao, Liang Dai, Lin Li, Liqun Deng, Peifeng Qin, Pengyuan Zeng, Qiang Gu, Shaohua Tang, Shengjun Cheng, Tao Gao, Tao Yu, Tianshu Li, Tianyu Bi, Wei He, Weikai Mao, Wenyong Huang, Wulong Liu, Xiabing Li, Xianzhi Yu, Xueyu Wu, Xu He, Yangkai Du, Yan Xu, Ye Tian, Yimeng Wu, Yongbing Huang, Yong Tian, Yong Zhu, Yue Li, Yufei Wang, Yuhang Gai, Yujun Li, Yu Luo, Yunsheng Ni, Yusen Sun, Zelin Chen, Zhe Liu, Zhicheng Liu, Zhipeng Tu, Zilin Ding, and Zongyuan Zhan. Pangu ultra MoE: How to train your big MoE on Ascend NPUs. *CoRR*, 2025.

[42] Shin-Yeh Tsai and Yiying Zhang. LITE kernel RDMA support for datacenter applications. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 306–324, 2017.

[43] Bingyang Wu, Yinmin Zhong, Zili Zhang, Gang Huang, Xuanzhe Liu, and Xin Jin. Fast distributed inference serving for large language models. *CoRR*, 2023.

[44] Yanan Yang, Laiping Zhao, Yiming Li, Huanyu Zhang, Jie Li, Mingyang Zhao, Xingzhen Chen, and Keqiu Li. INFless: a native serverless system for low-latency, high-throughput inference. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 768–781, 2022.

[45] Jiayi Yao, Hanchen Li, Yuhan Liu, Siddhant Ray, Yihua Cheng, Qizheng Zhang, Kuntai Du, Shan Lu, and Junchen Jiang. CacheBlend: Fast large language model serving for RAG with cached knowledge fusion. In *Proceedings of the 20th European Conference on Computer Systems*, pages 94–109, 2025.

[46] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. Orca: A distributed serving system for transformer-based generative models. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation*, pages 521–538, 2022.

[47] Minchen Yu, Zhifeng Jiang, Hok Chun Ng, Wei Wang, Ruichuan Chen, and Bo Li. Gillis: Serving large neural networks in serverless functions with automatic model partitioning. In *Proceedings of the 41st IEEE International Conference on Distributed Computing Systems*, pages 138–148, 2021.

[48] Chengliang Zhang, Minchen Yu, Wei Wang, and Feng Yan. MArk: Exploiting cloud services for cost-effective, SLO-aware machine learning inference serving. In *Proceedings of the 2019 USENIX Annual Technical Conference*, pages 1049–1062, 2019.

[49] Dingyan Zhang, Haotian Wang, Yang Liu, Xingda Wei, Yizhou Shan, Rong Chen, and Haibo Chen. Fast and live model auto scaling with O(1) host caching. *CoRR*, 2024.

[50] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Jeff Huang, Chuyue Sun, Cody_Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E Gonzalez, et al. Efficiently programming large language models using SGLang. *CoRR*, 2023.

[51] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. DistServe: Disaggregating prefill and decoding for goodput-optimized large language model serving. In *Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation*, pages 193–210, 2024.

[52] Kan Zhu, Yilong Zhao, Liangyu Zhao, Gefei Zuo, Yile Gu, Dedong Xie, Yufei Gao, Qinyu Xu, Tian Tang, Zihao Ye, et al. Nanoflow: Towards optimal large language model serving throughput. *CoRR*, 2024.

Cosmic: Cost-Effective Support for Cloud-Assisted 3D Printing

Yuan Yao[1], Chuan He[2], Chinedum Okwudire[2], and Harsha V. Madhyastha[1]

[1]*University of Southern California* [2]*University of Michigan*

Abstract

In this paper, we consider a new workload for which serverless platforms are well-suited: the execution of a 3D printer controller in the cloud. This workload is qualitatively different from those considered in prior work due to the stringent timing requirements. Our measurements on popular serverless platforms reveal millisecond-level overheads that impair the timely execution of the example control algorithm we consider. To mitigate the impact of these overheads, we judiciously partition the execution of the algorithm across a set of serverless functions and exploit timely speculation. Our evaluations on AWS Lambda show that, for 30 diverse print jobs, *Cosmic* is able to ensure the timely execution of the controller while reducing cost by 2.8×–3.5× compared to other approaches.

1 Introduction

Serverless computing platforms like AWS Lambda [2] and Azure Functions [3] have gained in popularity over the last few years. In these services, the customer registers their code – typically in the form of a container – with the cloud provider, who runs an instance of the code for every request received. Thus, the customer is freed from the onus of having to monitor load and correspondingly scale up or scale down their service deployment, and they do not pay for idle time.

The scalability and cost-effectiveness of serverless computing make it ideal for workloads with intermittent execution patterns, such as periodic data processing or event-driven workflows. However, when these workloads also demand high parallelization and millisecond-level latency guarantees, significant challenges arise. First, the overhead of cold starts and function invocations, while acceptable for many applications, becomes prohibitive for latency-critical tasks. Second, the management of large datasets across serverless instances introduces significant delays and coordination overhead, as network bandwidth and transfer times dominate.

In this paper, we study these challenges in serverless computing with a non-traditional workload: 3D printing. In particular, our interest is in the control algorithm that guides the operation of a printer by determining the sequence in which it should print the different portions of a product. What distinguishes the controller is the stringent constraint on the timeliness of its execution: before the printer is ready to execute the i^{th} step of a print job, the controller must have identified which portion of the product to print in that step.

Using a state-of-the-art control algorithm for a specific form of 3D printing as an example, we demonstrate that commodity 3D printers lack the hardware necessary to run complex control algorithms on-device. On the one hand, the CPU on the printer cannot execute the sequential control algorithm fast enough to keep up with the speed of printing. On the other hand, to ensure that I/O delays do not slow down the controller, the state needed by all steps of the algorithm must be preloaded into memory. But, the amount of on-board memory does not suffice to accommodate all the necessary state. Therefore, we consider the execution of the controller in a cloud region close to the 3D printer.

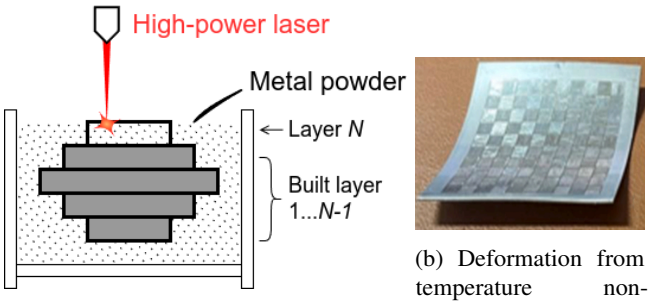
In contrast to running the control algorithm across a fleet of virtual machines, serverless offers two benefits.

- *No cost during idle time:* First, with serverless, the user incurs no cost during idle periods, such as when the printer pauses after each layer to spread metallic powder. These short idle periods of a few seconds make suspending and resuming a VM impractical.
- *No cost to keep state in memory:* Second, to handle requests at low overhead, cloud providers attempt to reuse serverless workers across requests [1]. These *warm starts* retain the state read into the memory of a serverless worker while printing one layer for subsequent layers, at no cost to the user.

Nevertheless, our measurements on AWS Lambda and Azure Functions show that exploiting these benefits while meeting the controller's timing constraints is nontrivial. Function invocation overheads, even on pre-warmed workers, remain too high to meet the stringent requirements of each step of the controller. While parallelizing each step across multiple workers reduces computation time, it introduces substantial delays due to the network overhead of transferring the relevant inputs to all workers.

To address these challenges, we have designed and implemented *Cosmic* (Cost-effective Serverless for Millisecond Computations), a system for cost-effective and timely execution of the low latency compute pipeline necessary to support high quality 3D printing. *Cosmic* distributes sequential algorithms across serverless workers, with three key strategies: intelligent workload partitioning to minimize coordination overhead and optimize function reuse, speculative execution to mask invocation delays, and an adaptive search for the configuration which balances performance and cost.

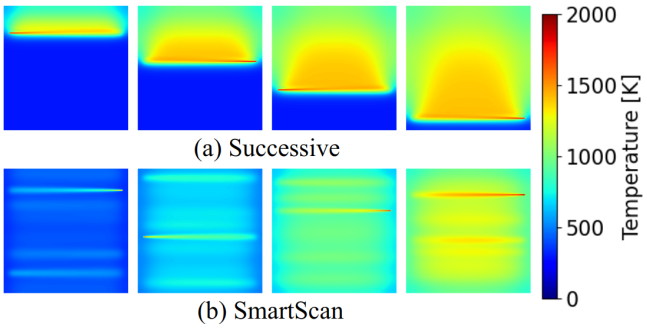
We use *Cosmic* to run the controller on Lambda for 30 print jobs: 10 diverse parts, each in 3 different sizes. We show that *Cosmic* satisfies the timing constraints on the controller's execution for all jobs; other approaches from prior work that have used serverless either fail in meeting these constraints for 23 of the 30 jobs, or do so at 3.2×–3.5× higher cost.



(a) Illustration of LPBF printing.

(b) Deformation from temperature non-uniformity.

Figure 1: **Powder bed fusion.**



(a) Successive

(b) SmartScan

Figure 2: **Comparison of temperature distribution when scanning a layer's cells (a) in order versus (b) with SmartScan.**

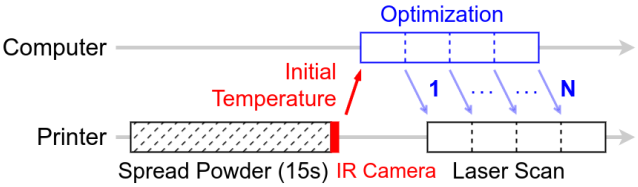


Figure 3: **Timeline of SmartScan execution.**

In summary, we make the following contributions:

- We consider a workload that has not been of much interest in the systems and networking research community – real-time control of 3D printers – and showcase the relevance of serverless computing for this workload.
- We present measurements of popular serverless platforms to highlight previously ignored overheads that make it challenging to satisfy the millisecond-level timing requirements of the control algorithm we consider.
- We introduce a new strategy for executing a sequential algorithm on serverless and demonstrate how to use it to cost-effectively satisfy the algorithm's timing constraints.
- Lastly, we demonstrate the utility of our system, *Cosmic*, by using it to run the controller for a variety of print jobs.

Though we focus in this paper on supporting the execution of a 3D printer's controller, many aspects of *Cosmic*'s design extend to other latency-sensitive compute pipelines. We discuss the broader applicability of our work in Section 7.

2 Background and Motivation

Powder bed fusion. There exist a variety of 3D printing technologies [12, 20, 26, 28]. In this paper, we focus specifically on laser powder bed fusion (LPBF), which is used to build three-dimensional parts by selectively fusing or melting metal powder using a laser, as shown in Figure 1(a).

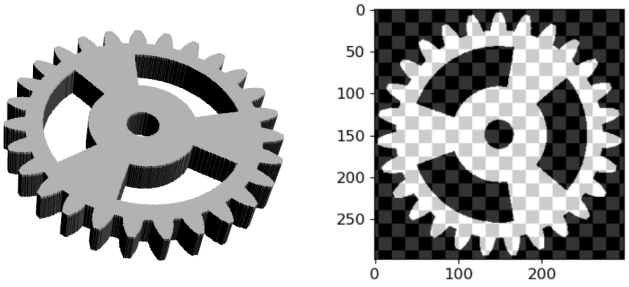
In LPBF, three-dimensional parts are constructed layer by layer. Prior to scanning a layer, a thin layer of metal powder is spread across the platform. The high-power laser then scans this layer, selectively melting the powder to create a solid cross-section of the part. This procedure is repeated for each layer until the entire part is constructed.

One of the primary challenges in LPBF printing is in managing the distribution of temperature during the scanning process. For example, consider Figure 2(a), which shows the print of a square metal plate with the laser scanning each layer from the top edge to the bottom edge. By the time the laser starts scanning the end of the layer, the top half cools down, resulting in a non-uniform distribution of temperature across the layer. As shown in Figure 1(b), such non-uniformity in temperatures can lead to residual stresses and deformities in the final product.

SmartScan. To maximize temperature uniformity, the solution is to optimize the order in which an LPBF printer scans each layer. For this, we consider *SmartScan* [29], a feedforward control algorithm. SmartScan relies on a logical division of each layer into *cells*, e.g., a square metal plate can be partitioned either into a set of horizontal or vertical stripes, or a set of square islands. To determine the optimal sequence in which the cells in a layer must be scanned, SmartScan uses a physics-based thermal model to simulate temperature distribution and heat transfer.

As depicted in Figure 3, the print of a layer starts with the distribution of metal powder across the print platform, which takes a fixed duration of 15 seconds. The printer then employs an infrared camera to capture an initial map of how temperature varies across the layer. The SmartScan algorithm then iteratively computes the index of the next cell to print. Each iteration involves multiplying the current temperature map with a precomputed heat transfer matrix to produce an updated temperature map, which the controller analyzes to pick the next cell index. The process continues until the entire layer has been scanned. Figure 2(b) illustrates the improved temperature uniformity achieved with SmartScan.

Note that, though the cell-specific heat transfer matrices for a part can be reused across multiple prints of that part, the results of SmartScan's computations cannot be reused. The optimal sequence for printing a layer does not depend only on the layer's geometry. It also depends on the initial temperature distribution captured by the infrared camera before the scanning of a layer begins. The temperature distribution at the start of each layer is influenced by multiple dynamic factors which vary across layers, such as residual heat from



(a) A gear (b) One of the layers

Figure 4: **An example part printed using LPBF, and an example partitioning of one of its layers into cells.**

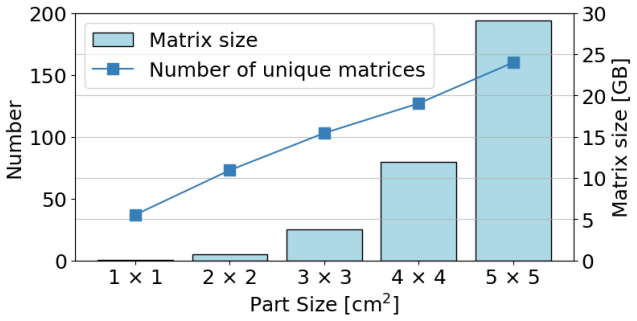


Figure 5: **For the gear in Figure 4, size of each heat transfer matrix, and number of unique matrices in relation to layer size.**

previously melted layers and the spreading of powder across the platform. Unlike the modeling of heat transfer across cells within a layer, these other heat transfer processes cannot be modeled efficiently; hence, real-time computation of the optimal scanning sequence is essential.

Challenges in local execution. Unfortunately, running SmartScan on an LPBF printer's on-board computer is infeasible for most print jobs. This is because, as shown in Figure 3, in order to maintain the accuracy of SmartScan's simulation of heat transfer, it is crucial that, by the time the printer is done scanning a cell, the cell that it should print next has already been computed and delivered to it. Given the typical speed of printing, the length of the time within which SmartScan's computation of the next cell to print must complete – which we refer to as a *time window* – is quite short. For example, at a speed of 600mm/s [29], printing a length of 3cm takes only 50ms. If SmartScan fails to finish computation within a time window, the printer will fall back to printing a random cell, thus degrading printing quality.

Due to this stringent timing constraint, the heat transfer matrices for *all* cells in a layer must be prefetched into memory; else, within any window, delays when reading the heat transfer matrix from disk for the cell currently being printed will leave little to no time to perform the computation necessary to select the next cell. This is a problem because, as shown in Figure 5 for the example gear shape in Figure 4, as the size of the printed part grows, so does the number of cells and the size of the matrix for each cell. For example, to print the gear with each layer of size 16cm^2, 1,514 GB of data – 127 matrices, each of size 11.92 GB – will need to be in memory, which greatly exceeds the amount of memory on-board an LPBF printer.

Moreover, the processor on a typical LPBF printer is incapable of executing SmartScan to keep up with the speed of printing. For example, it takes around 450ms to complete the matrix multiplication necessary to model the distribution of heat during the print of a single cell in the above-mentioned gear. This is significantly longer than the roughly 50ms time window within which this computation must complete.

3 SmartScan in the Cloud

One could potentially try to address the limitations of the computer on-board an LPBF printer by offloading control of the printer to a fleet of edge servers. But, given that SmartScan's memory and compute requirements increase with the size of the part being printed (Figure 5), an edge deployment sufficient to support the print of large parts would be significantly under-utilized when printing small parts.

To instead enable elasticity, we seek to run SmartScan at a cloud data center close to the printer. If the printer loses connectivity with its controller in the midst of a print job, it can revert to how it would operate without SmartScan, i.e., print all remaining cells in a random order. We focus on parts that are symmetric along the z-axis; for such parts, since the heat transfer matrices are identical for every layer, in-memory data can be reused across layers.

When running SmartScan in the cloud, precomputed matrices can be generated on a cloud VM and uploaded to cloud storage. Within a datacenter, cloud providers do not charge for data transmission between VMs and cloud storage, as well as between the storage and serverless workers. Moreover, since the heat distribution matrices need to be precomputed once per part, their generation has no impact on the delays involved in every job which prints that part.

3.1 Why serverless?

In every step of the SmartScan algorithm, the controller computes the temperature distribution expected after the printer scans and melts the cell chosen in the previous step. It does so by multiplying the temperature distribution vector computed in the previous step with the precomputed heat transfer matrix for the chosen cell. To ensure that the next cell to print is determined before the printer completes printing a cell, the multiplication can be parallelized across a set of virtual machines (VMs) or serverless workers; each VM/worker can multiply the vector with a partition of the matrix, and these results can be aggregated by a coordinator.

We argue that serverless workers are more well-suited for this workload than VMs for two reasons.

3.1.1 Cost overhead of VMs

Paying for in-memory data when not in use. First, SmartScan's timing constraints make it infeasible for a VM to dynamically read in the heat transfer matrix for the chosen cell on-demand from cloud storage. For instance, if we execute SmartScan for a 16cm^2 gear on a x2gd.xlarge VM on Amazon EC2, the computation can keep up with a printing speed of 600mm/s. But, the length of each time window at this speed is roughly 50ms, which is significantly shorter than the 350ms it takes for the VM to read in the 11.92 GB matrix for a cell from an EBS volume which uses a gp3 SSD configured with the maximum permitted limits on IOPS and throughput. Retrieving this data from Amazon S3 takes even longer: 8 seconds.

Therefore, to adhere to SmartScan's timing constraints, the heat transfer matrices for all cells must be readily available in memory throughout the algorithm's execution. This calls for the use of several expensive high-memory VMs. For example, the 249 cells in a 16cm^2 gear map to 127 unique heat transfer matrices, whose total size is 1,514 GB.

Paying for idle VMs. Second, as mentioned earlier, there is a 15-second period between layers when metallic powder is spread onto the printer bed. Though no computation is performed during this period, 15 seconds is too short to suspend and resume all the VMs that support SmartScan's execution. Thus, the user will need to pay to keep all data in-memory during this period.

Note that it is challenging to reduce idle time on VMs by multiplexing each VM across multiple print jobs. On the one hand, the 15-second idle period for one job is not long enough to read the data for another job into memory. On the other hand, if multiple jobs printing the same part share a set of VMs preloaded with the data for that part, the execution of SmartScan for these jobs will need to be carefully coordinated so that they do not interfere with each other's ability to use the CPUs on the VMs.

3.1.2 Advantages of serverless

In addition to enabling their customers to rent VMs, many cloud providers today also have function-as-a-service offerings; AWS Lambda and Azure Functions are popular examples. In these services, the user registers with the cloud provider a container which encapsulates the user's code. In response to a new request, the cloud provider spins up an instance of that container – which we refer to as a *worker* – and executes the request in that instance.

Two characteristics of serverless platforms help address the aforementioned cost overheads associated with using VMs to execute SmartScan.

Pay per use pricing. First, the pricing model for services such as AWS Lambda is structured such that users have to pay only for the duration when a worker is running the user's code. As a result, the user will not incur any cost during the 15-second idle period between layers when no computation is needed. Moreover, in each time window, the user need only pay for the workers that are computing the expected impact on temperature distribution due to the scanning of the cell chosen in the previous window.

Warm starts. Second, once a worker is initiated, the cloud provider keeps that worker's state in memory for a few minutes even after the worker's execution ends. As long as the gap between consecutive invocations of a function is less than a few minutes, the worker initiated for the first invocation is reused for the second invocation. Thus, the workers that are used to execute SmartScan when scanning one layer of a part will retain the heat transfer matrices in memory even when the scanning of the next layer is underway. Importantly, despite the cloud provider keeping the worker's state in memory between successive invocations of a function, the user incurs no cost during this period.

Though cloud platforms do not provide any guarantees on reusing worker state in this manner, our measurements on both AWS Lambda and Azure Functions reveal that warm starts are highly reliable. Over a 24-hour period, we conducted hourly trials where we issued sequences of function calls at fixed intervals. For shorter invocation intervals, ranging from 50 milliseconds to 1 second, cold starts were extremely rare, occurring in less than 0.1% of trials in a day. For longer idle periods, ranging from 5 seconds to 1 minute, memory eviction becomes slightly more likely, with cold starts increasing from 0.6% to 3.2% as the interval lengthened. To prevent cold starts across layers, which typically involve idle times of tens of seconds, users can periodically issue dummy requests to keep workers alive. The cost of sending such dummy requests is negligible, as they require only a few milliseconds of execution per second.

3.2 Challenges with serverless

Taking advantage of the characteristics of serverless platforms to support the cost-effective execution of SmartScan is, unfortunately, not easy. To see why, we now consider some of the approaches considered in prior work for other workloads and demonstrate their limitations via measurements. For simplicity, in this section and the next, we assume that each cell maps to a unique heat transfer matrix.

3.2.1 Even warm starts incur invocation overhead

The simplest approach to implement SmartScan on serverless, which we refer to as the *Invoke-on-demand* strategy, involves registering a separate function for each cell in the part being printed. At the start of the print, we concurrently invoke all functions once so that they can read in the matrix assigned to them from cloud storage into memory and exit; because of warm starts, this data remains in memory. Thereafter, to print each layer, a coordinator VM collects the temperature map from the printer, and then iteratively repeats the following steps (as shown in Figure 6): it invokes the function associated

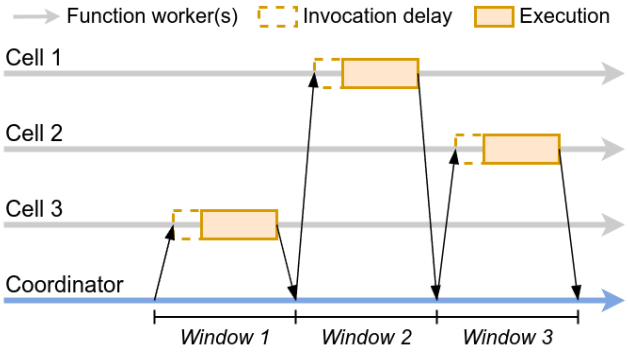


Figure 6: **Execution of SmartScan wherein a new function is invoked every time the algorithm picks a new cell to scan.**

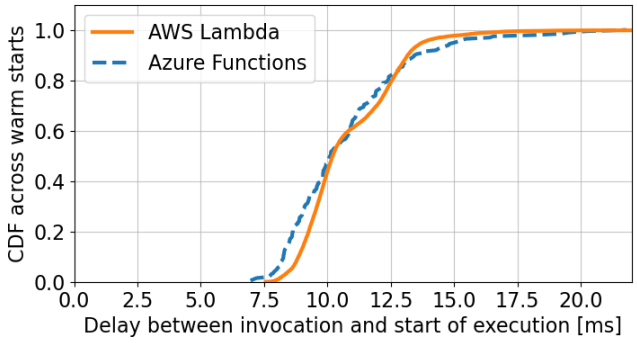


Figure 7: **Distribution of the overhead of invoking a serverless function when executed on a pre-warmed worker.**

with the cell chosen in the previous step, collects the result and identifies the next cell to scan, and relays this choice to the printer. In each step, the executed function multiplies the input temperature map with its in-memory heat transfer matrix to compute the estimated temperature map after the printer scans the chosen cell.

Like prior work which has aimed to take advantage of serverless computing's pay-for-use pricing model [34, 35], this simple implementation ensures that, at every step, the user incurs cost only for the computation associated with one cell. The user also does not incur any cost during the 15-second idle period between the scanning of successive layers. Thus, as long as the number of unique heat transfer matrices associated with a part is large, the cost for a SmartScan-optimized print of the part will be significantly smaller with this implementation than one which runs SmartScan on EC2 VMs; see Appendix A for a detailed analysis.

Invocation delays result in violation of timing constraint. However, we find that the *Invoke-on-demand* strategy for executing SmartScan on Lambda fails to keep up with the printing speed. For example, in our execution of SmartScan on Lambda for any one layer of a 4cm^2 gear, Lambda workers are actively executing code for only 1.30 seconds, which is less than the 3.33 seconds needed to print the layer. But, SmartScan's per-layer runtime includes an additional 2.98 seconds. When combined, these delays add up to 4.28 seconds per layer, exceeding the 3.33-second print time constraint. This additional delay averages 24.8ms for each of the 120 cells in a layer, encompassing not only the *invocation delay* (the time between when we invoke a function and when a worker begins executing the function's code), but also the time spent sending temperature maps to the workers and the cleanup overhead incurred by repeatedly invoking functions.

Invocation delays occur despite the fact that all of our function invocations are served by pre-warmed workers that already have the function state in memory. The underlying cause is overheads internal to Lambda associated with its routing of requests [7, 10] and context preparation.

To measure this overhead, we invoke a pre-warmed function instance from an EC2 VM. At the start of its execution, the function issues a HTTP request back to the VM. We measure the invocation delay by comparing the times at which the VM invokes the function and when it receives the HTTP request from the function. Note that the measured invocation delay includes a network round-trip between the VM and the worker, which is sub-millisecond within a datacenter.

Based on our measurements on AWS Lambda, Figure 7 shows that the invocation delay is 10.2ms at the median, and 13.2ms at the 90th percentile. Figure 7 also shows that these delays are similar with Azure Functions. For workloads that have previously been considered a good match for serverless, such as video processing [8, 16] and model training [18, 34], function invocation delays have been insignificant since the execution time of individual functions has been in the order of seconds or higher. In contrast, in our workload, overheads even in the order of 10ms end up resulting in violations of SmartScan's timing constraints.

3.2.2 High parallelism leads to coordination overheads

Eliminating invocation overhead inflates cost. To avoid the overheads associated with routing each function invocation to a worker, we can invoke the functions for all cells at the start of a new layer. Every function then waits for input from the coordinator to perform the appropriate matrix multiplication, akin to the strategy used in some prior systems [8, 16].

Executing SmartScan with this *All-persistent* strategy, however, significantly inflates costs. The user would not need to pay for any computation during the idle time between layers when metallic powder is being spread for the next layer. But, since the functions for all cells will be active throughout the duration that a layer is being scanned, we lose the benefit of warm starts; the user will have to pay for a function's state to be kept in memory even when the worker assigned to run that function is idle. Since serverless workers cost more per GB of memory per second than VMs, the net effect is that executing SmartScan with the *All-persistent* approach eliminates the benefits of using serverless.

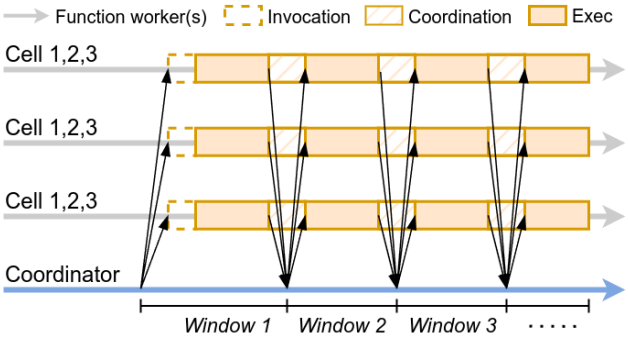


Figure 8: **Execution of SmartScan wherein the computation for every cell is parallelized across all workers.**

Keeping all workers active introduces coordination overheads. To lower cost, one might think: if the workers for all functions are active throughout the print of a layer, we might as well utilize the CPUs on all of them during that period? We can accomplish this by rethinking the assignment of cells to workers. Thus far, we have considered that each function is assigned to store and process the matrix for a unique cell. Instead, we can assign $\frac{1}{N}^{th}$ of every matrix to each function, where N is the number of workers needed to keep all the matrices in memory. As shown in Figure 8, in every step of SmartScan's execution, the coordinator VM sends the current temperature map to all workers, each of whom multiplies their slice of the matrix for the chosen cell. By aggregating results from all workers, the coordinator assembles the updated temperature map.

The high degree of parallelism with this *All-active* strategy, however, ends up inflating SmartScan's runtime. For example, we need 159 Lambda workers to accommodate all the matrices for the 16cm^2 gear. When we execute SmartScan across these many workers using the *All-active* strategy, it takes 16 seconds per layer, which greatly exceeds the constraint of 13.3 seconds imposed by the printing speed.

The primary source of delay here is the limited network bandwidth at the coordinator. For example, of the 16-second runtime per layer, only 270 ms involve any worker actively executing matrix multiplications. The remaining *coordination delays* stem from the overheads incurred in transferring the temperature map in each step from the coordinator to all workers. In our example job, each of the 249 steps involves the coordinator pushing a 312 KB map to 159 workers – up to 50 MB of data per step. Note that, though each worker computes the output temperature for only a subset of cells, every worker receives the entire input temperature map. This is because the heat transfer matrix models the distribution of heat from every cell to every other cell. This data is transmitted at a rate of roughly 5 Gbps, resulting in a delay – the *coordination delay* – which takes up almost all of the runtime.

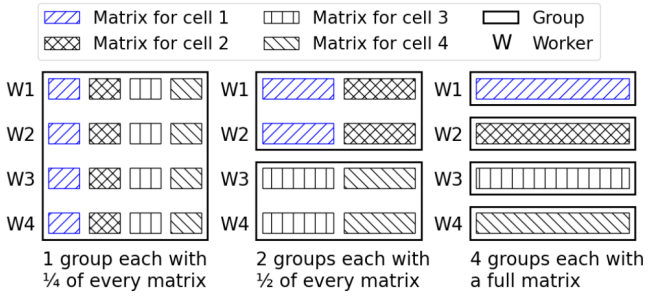


Figure 9: **Three ways of dividing the matrices for 4 cells into groups across 4 workers.**

4 Design

The takeaway from our measurements in the previous section is that simple approaches for executing SmartScan using serverless either fail to meet the algorithm's timing constraints or significantly inflate cost compared to using VMs. To address this problem, we present *Cosmic*, a new system for executing low-latency compute pipelines like SmartScan on serverless platforms.

Cosmic's design has three key components. First, in terms of how serverless is used to run SmartScan, we expand the configuration space beyond the point solutions discussed in the previous section. Second, we develop a model that can accurately estimate for each configuration, the execution time of SmartScan in that configuration and the cost the user would incur. Using this model, *Cosmic* is able to identify the lowest cost configuration that is expected to satisfy SmartScan's timing constraints. Third, we implement a runtime which can ensure that SmartScan is executed in accordance with the chosen configuration.

Though *Cosmic* is tailored to SmartScan's 3D-printing control loop, we believe its modular components are workload-agnostic and reusable across latency-sensitive cloud applications. We elaborate on this in Section 7.

4.1 Expanding configuration space

Any particular serverless configuration for running SmartScan specifies two properties: a) *which* set of functions are used to run the computation for any particular cell, and b) *when* those functions are invoked. In comparison to the approaches discussed in the previous section, we expand the space of configurations along two dimensions.

Grouping. First, we rethink the assignment of cells to functions. Existing approaches either assign each cell to a separate set of functions (*Invoke-on-demand* or *All-persistent*) or have a common set of functions execute the computations for all cells (*All-active*). In contrast, *Cosmic* allows for many more intermediate configurations as follows.

If N functions are registered in total across all cells – where N must at least be equal to the number of workers needed to store in memory the matrices for all cells – we divide these functions into G groups of $\frac{N}{G}$ workers each. Each group

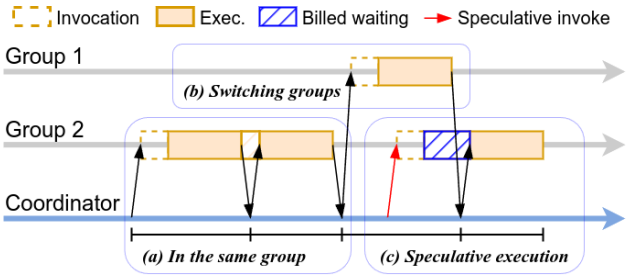


Figure 10: **Illustration of SmartScan execution with *Cosmic*.**

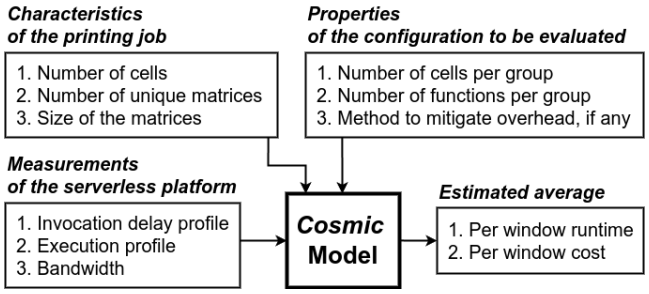


Figure 11: **Inputs and outputs of model used in *Cosmic*.**

of workers is responsible for performing the computations corresponding to $\frac{1}{G}^{th}$ of the part's cells.

For example, Figure 9 shows three groupings for a part which has four cells, each with a unique heat transfer matrix. The right and left show configurations realized by existing approaches: 4 groups with a unique function assigned to each cell (right), or the state for all cells is equally distributed across one group of 4 functions (left). In the middle is a new configuration enabled by our approach: 2 groups of 2 functions each, with 2 cells assigned to either group.

Speculation. Second, we observe that it is not necessary to incur the cost overhead of keeping the workers for all cells active throughout the printing of a layer. Instead, it suffices if every function is invoked prior to when it needs to start running. Thus, while any step of SmartScan is executing, we can *predict* the cell likely to be chosen in that step and preemptively invoke the functions assigned to that cell.

Benefits. Partitioning workers into groups and assigning multiple cells to each group combined with speculative execution offers multiple advantages.

- *New mechanisms for mitigating invocation overhead.* When the cells chosen in two successive windows belong to the same group, no function invocation overhead is incurred at the start of the second window. For example, in the first two windows in Figure 10, the workers that will execute the computation for the second cell will already be active after the first window. In addition, speculative invocation of functions enables *Cosmic* to reduce the impact of function invocation overheads without incurring the cost overhead of keeping all workers alive.
- *New tradeoffs between invocation and coordination overhead.* At one extreme, assigning all cells to a single group eliminates invocation overheads, but will result in high coordination overhead. At the other extreme, assigning each cell to its own small subset of functions ensures that the network on the coordinator VM does not prove to be a bottleneck; but, without incurring the cost overheads associated with speculation or keeping all workers alive, the start of every window will be delayed by function invocation overheads. By varying the number of cells per group, we can realize various intermediate points in the tradeoff between the two overheads.
- *Easier speculation.* The assignment of multiple cells to the same group of workers allows for easier use of speculation. As long as the predicted cell belongs to the same group as the cell chosen by SmartScan, invocation delays will not affect our ability to satisfy timing constraints.

4.2 Choosing most cost-effective configuration

Enumerating all configurations. Given a product to print, we use the combination of three attributes to represent every serverless configuration that *Cosmic* can use to execute SmartScan while the product is printed:

- The number of cells per group: varies from 1 to the total number of cells that the product has been partitioned into
- The number of functions per group: at least the number of workers needed to accommodate in memory the heat transfer matrices for all cells in a group
- Which, if any, method is employed to address function invocation overheads: workers for all functions are kept active throughout the printing of each layer, functions for the next window are invoked speculatively, or neither

The question at hand then is: among all configurations which will enable SmartScan's execution to keep up with the printing speed, which one minimizes cost? In our current implementation, *Cosmic* exhaustively enumerates all configurations; for the parts we have considered, this takes around 10 seconds at the start of a print job. Among those configurations where the estimated runtime is less than the time needed to print a layer, *Cosmic* picks the one with the lowest estimated cost. In extreme cases where the time constraint is too stringent (e.g., because of a high printing speed), *Cosmic* can determine when no feasible solution exists. In such cases, SmartScan must be run across a fleet of VMs.

Model to estimate runtime and cost. For every configuration that *Cosmic* considers during its search for the lowest cost configuration, it needs to estimate SmartScan's runtime per layer with this configuration and the corresponding dollar cost. Given the large number of configurations for even a simple part, it is impractical to determine these estimates by executing SmartScan in every configuration.

Instead, we need to estimate the runtime and cost associated with any configuration *without* running SmartScan in that

configuration. For this, as shown in Figure 11, we develop a model which takes three kinds of inputs: 1) characteristics of the product to be printed, 2) properties of the configuration being evaluated, and 3) measurements of the serverless platform. Since the SmartScan algorithm involves a linear sequence of steps, with each step lasting a window, the model's task is equivalent to computing the average runtime and cost per window. The duration of each window has two components:

- First, the coordinator VM requests the serverless platform to invoke the functions for the cell being scanned in the window. The workers which execute these functions request the function input from the coordinator. The delays involved depend on invocation overheads and the effectiveness of the speculation strategy.
- Then, the coordinator sends the input temperature map to all workers and receives the output from them. Delays include coordination and computation time.

We model each component of window duration separately.

In configurations where workers are kept alive throughout the printing of a layer, the first delay component is zero as no function invocation overhead is invoked. In configurations which employ speculation, this delay is zero for a fraction of the windows equal to the speculation accuracy; the expected speculation accuracy can be estimated from previously completed prints of the same product. The delay is also zero for windows in which the group is reused; we estimate the expected chance of reuse by simulating random selection of cells. For the other windows, or for all windows in the remaining configurations, we estimate the magnitude of the first component as equal to the overhead of invoking functions on warm workers. We rely on a one-time profiling of function invocation delays, using measurements akin to those used to generate Figure 7. Due to the heavy tailed distribution of these delays, instead of simply using the average or median value for all windows, we sample from the measured distribution independently for every window.

The second component of a window's duration, which is also the period for which the user is billed, includes 1) the time for the coordinator to disseminate the temperature map obtained at the end of the previous window as input to all the workers, and 2) the time taken by the workers to execute the matrix multiplication which models the distribution of heat during the printing. For the former, we observe that the bandwidth between the coordinator VM on EC2 and the Lambda workers is the bottleneck. We measure this bandwidth limit once using `iperf`. For every print job, we can then use the size of the temperature map for each layer and the number of functions per group to estimate the time it will take for the coordinator to finish sending the inputs to all workers. For the latter, we rely on a profile of the time to execute matrix multiplication on Lambda. We record the distribution of execution time for a range of matrix sizes, and interpolate from this data for any particular part.

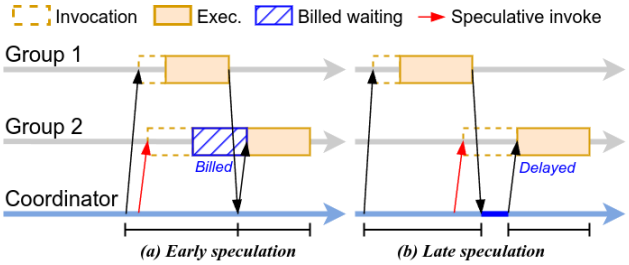


Figure 12: **Illustration of early and late speculation.**

4.3 Executing chosen configuration

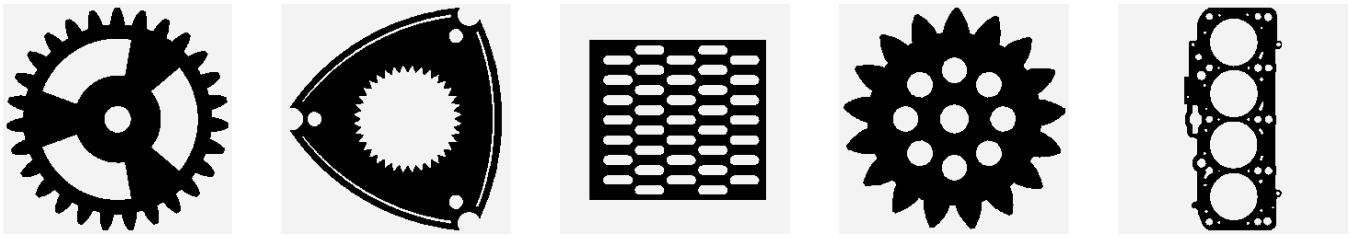
Once *Cosmic* uses its model to select the most cost-effective configuration, the coordinator VM oversees the execution. It invokes functions when necessary: all functions once at the start of the print job so that all workers can read their state into memory; then, the relevant functions at the start of each window. The primary complexity in the coordinator's role is in determining how and when it should speculate.

Speculation algorithm. While the SmartScan algorithm is running its computation in any particular window, we need a lightweight method to predict the cell that will be chosen for printing next. Our speculation strategy is inspired by approximation techniques commonly used in 3D printing algorithms to reduce computation at the expense of lower printing quality [29]. Instead of to avoid compute bottlenecks, we leverage approximation to quickly generate guesses with minimal additional execution time and cost.

In SmartScan, recall that the primary purpose of the computation is to model the diffusion of heat. SmartScan maintains a temperature map for a layer and simulates the scanning of each cell in two steps: 1) applying the input heat from the laser to the temperature map, and 2) updating the heat transfer across the layer during scanning. The compute bottleneck is the second step as it involves multiplying a matrix, which models the expected heat transfer, with the temperature map. In contrast, the first step simply involves adding a precomputed vector to the temperature map.

The approximation of this computation is rooted in the observation that scanning each cell with a laser typically takes only tens of milliseconds. Therefore, a good approximation of the updated temperature map can be obtained by considering only the laser input and ignoring the heat transfer over such a short period. This approximate temperature map, which can be computed by the coordinator using much less computation than the full algorithm executed concurrently by the serverless workers, can then be used to guess the next cell that will be selected.

Note that this approximation cannot replace the full algorithm. If we rely on the approximated temperature map in every window, the simulation gradually deviates from the real state of heat transfer, thus degrading SmartScan's ability to reduce the variance in temperature across the layer.



(1) Segmented gear (2) Triangular rotor (3) Slotted grid (4) Holed gear (5) Cylinder gasket

(6) Connecting rod (7) Mounting bracket (8) Ribbed bar (9) Holed L-bracket (10) Simple L-bracket

Figure 13: **Shapes of the printed parts.**

Speculation timing. In addition to identifying *which* functions to invoke speculatively, we need to determine *when* to invoke them. Launching the group of functions early can ensure that they are ready to begin executing when SmartScan's next window starts, but doing so can inflate cost. Deferring the speculative execution for as late as possible can reduce cost but also reduce the utility in overcoming function invocation overheads. Figure 12 illustrates these two scenarios.

To balance the two concerns, we rely on our profiling of invocation overheads (Figure 7). During any window in the execution of SmartScan, *Cosmic*'s coordinator estimates how long it expects the window to last. It speculatively invokes the functions for the predicted cell's group D ms prior to when it expects the current window to end, where D is the median of the measured distribution of function invocation delay.

5 Evaluation

We evaluate *Cosmic* from three perspectives: Is it able to satisfy the timing constraints associated with SmartScan's execution? What cost benefits does it offer? What contributions do each of *Cosmic*'s components make in enabling these benefits? We compare *Cosmic* to the three baselines from §3.2, which mimic the approaches used in prior work.

1) ***Invoke-on-demand***: The computation for every cell is distributed across a separate set of N functions, which are invoked on-demand.
2) ***All-persistent***: The computation for each cell is again spread across N functions. But, all functions are invoked at the start of a layer and return only once the computation for the entire layer ends. To minimize cost, we assign a group of cells to the same set of N functions if the workers for those functions can collectively store the state for all of these cells in memory.
3) ***All-active***: The data and computation for all cells are distributed across a common set of N functions. As a result, the workers for all functions are kept busy for the entire duration of the print job.

In addition, we also compare *Cosmic* to the traditional approach of using N VMs, with the data and computation for all cells equally split across the fleet. In all cases, we vary the value of N to identify the lowest cost solution which ensures that SmartScan keeps up with the printing speed.

5.1 Evaluation setup

We evaluate *Cosmic* and the baseline approaches by executing a real SmartScan controller in the cloud. We mimic a print job by having the controller relay its commands to a local desktop computer, which substitutes for an LPBF machine. This mirrors a realistic deployment because any industrial grade LPBF machine is itself driven by desktop software. On the desktop, we simulate the mechanical printing process using the printer's precise specifications, e.g., uploading randomly generated temperature maps and timing "laser movements" with corresponding delays. We host the controller in the AWS region closest to the desktop.

This setup enables us to examine the feasibility of real-time control, even though the current printer hardware does not yet support this capability; the current proprietary driver on our LPBF printer only accepts precomputed sequences. Importantly, it also provides two other major benefits: 1) enhanced data instrumentation, allowing fine-grained timing measurements, and 2) the ability to conduct large-scale, long-term evaluations without monopolizing physical resources.

Cloud hardware. We configure each AWS Lambda function to its maximum capacity: 10 GB of memory and 6 vCPUs. Using the largest worker size allows for fewer parallel workers or fewer groups for a given workload, reducing coordination delays and improving speculation accuracy. We choose Lambda functions equipped with ARM-based processors over x86 due to their lower cost, without sacrificing performance.

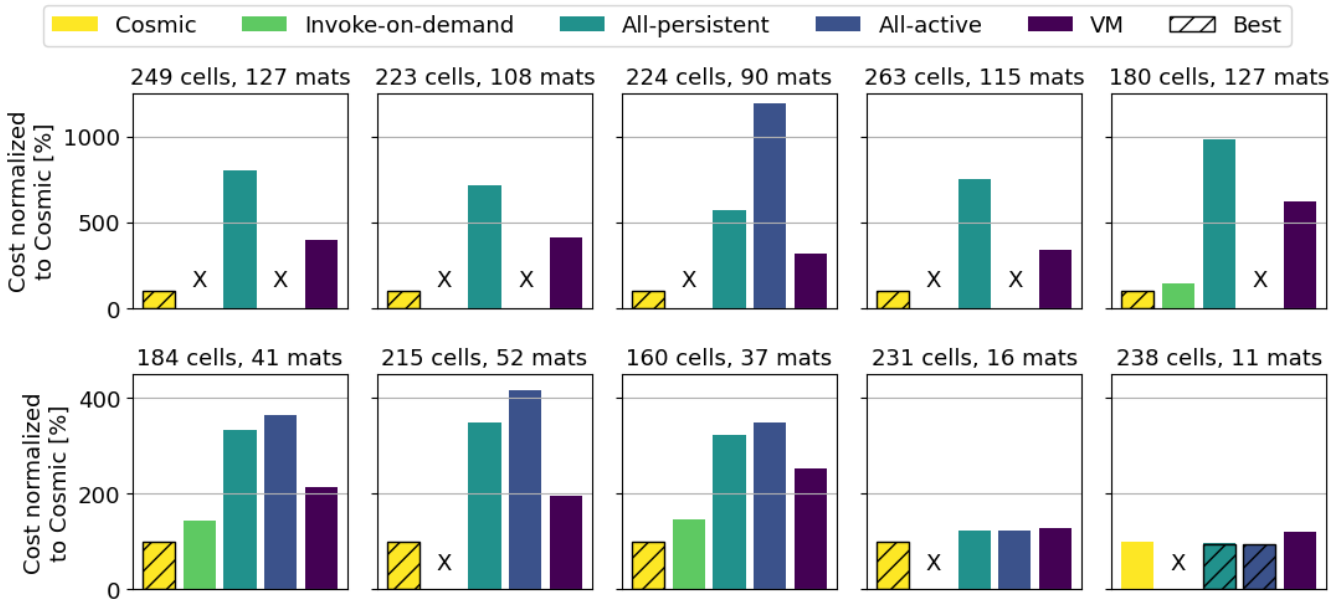


Figure 14: **Comparison of *Cosmic* with baseline approaches across 10 print jobs with layer size 16cm^2. The cost is normalized to that obtained with *Cosmic*. X indicates the timing constraints were not met. The bars filled with hatched marks show the cheapest solution for each job. The subplots are in the same order as in Figure 13.**

We set up the coordinator on a m6g.8xlarge instance with 32 vCPUs to efficiently handle concurrent requests. To minimize latency of communications between the coordinator and workers, we place all of them in the same VPC and disable TCP slow start on the coordinator.

For the VM-based baseline, we select instances from EC2's x2gd series, which are among the instance types with the highest memory-to-vCPU ratios, offering 16GB of memory per vCPU; a large memory-to-vCPU ratio helps avoid the cost inefficiency of over-provisioned cores.

Printing parameters. We consider a diverse set of 10 parts, shown in Figure 13, which are either derived from benchmarks used in PBF optimization studies [29,33] or designed to model real industrial parts. We consider each part in three layer sizes – 9cm^2, 16cm^2, and 25cm^2 – giving a total of 30 print jobs which vary widely with respect to 1) number of cells per layer, 2) number of unique heat transfer matrices, and 3) size of the matrices. For example, among the parts with a 16cm^2 size, the time constraint per window ranges from 38ms to 62ms, and the total memory requirement ranges from 131 GB to 1,514 GB.

We assume a printing speed of 800 mm/s, which falls within the range for optimal LPBF results on AISI 316L stainless steel, the material used in prior experiments [19, 22]. We selected this slightly higher speed than the more conservative 600 mm/s used in the original SmartScan paper [29], to evaluate *Cosmic* under more stringent conditions.

We assume the thickness of each layer to be 50 μm [29]. For example, a 1 cm thick part would comprise 200 layers.

We start each print job assuming temperature across the entire layer is uniformly at 293 Kelvin. Though the initial temperature map can vary in practice due to residual heat from previous prints, the computation performed when executing the SmartScan algorithm will be unchanged. Moreover, we show later that the accuracy of our speculation is largely the same as we vary the initial temperature map.

5.2 Satisfaction of timing constraints

In every print job, we measure the timing of the controller's commands received at the desktop. In all 30 print jobs, we find that *Cosmic*'s execution of SmartScan keeps up with the specified printing speed, i.e., prior to the start of every window, the desktop is expected to have received the next cell to be scanned. Though the *All-persistent* and VM-based approaches also manage to satisfy the timing constraints in all print jobs, they do so at the expense of over-provisioned compute resources and user-paid idle time. *All-active* satisfies the constraints in all 10 jobs with layer size 9cm^2, as few workers are needed to accommodate all the matrices in memory. However, *All-active* fails to finish running SmartScan in time for 11 of the remaining 20 jobs, as coordination overheads increase. *Invoke-on-demand* fares the worst, as it failed to meet the time constraints in 23 of the 30 jobs, irrespective of the amount of parallelism per cell. This is largely due to invocation delays associated with warm starts (§3.2.1).

5.3 Comparison of dollar costs

For printing the parts with layer size 16cm^2, Figure 14 shows the dollar cost for resources in the cloud with each of the approaches considered, normalized to the cost with *Cosmic*. Results are similar for prints of the other two sizes. For every approach, we consider the cost associated with the cheapest

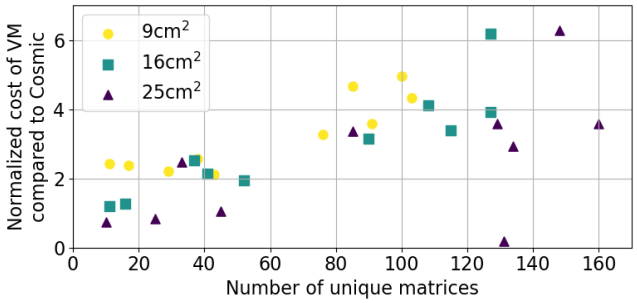


Figure 15: **Cost savings with *Cosmic* compared to VM-based approach as a function of number of heat transfer matrices.**

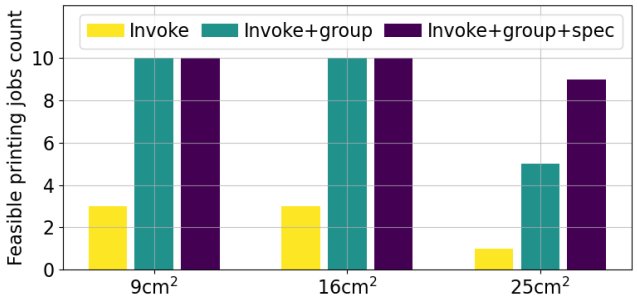


Figure 16: **Number of print jobs for which timing constraints are met first with *Invoke-on-demand*, and then by adding grouping and speculative invocation incrementally.**

configuration in which SmartScan's execution keeps up with the printing speed. We mark an 'X' if, for that job, that approach cannot meet the time constraints in any configuration.

We see that *Cosmic* outperforms all four baseline approaches in 26 of the 30 print jobs; it results in the lowest cost in all 10 prints of 9cm^2 layer size, in 9 of the 10 with layer size 16cm^2, and in 7 of the 10 with 25cm^2 layer size. In the 4 print jobs where *Cosmic* does not minimize cost, the cheapest solutions cost 0.94×, 0.83×, 0.75×, 0.17× lower, with the first achieved by *All-active* and the rest using VMs.

Cost-savings compared to VMs. Across all 30 print jobs, the cost overhead of VM-based solutions relative to *Cosmic* is 2.8× at the median, with a maximum of 6.3×. If the thickness of every part was 1cm, for example, these numbers translate to an average cost savings of $1.2, $4.5, and $18.7 per part for 9cm^2, 16cm^2, and 25cm^2 layer sizes, respectively. *Cosmic*'s cost savings relative to VMs increases with layer size due to the decreasing fraction of idle time in SmartScan's workflow (see Appendix A). Powder spreading between layers (§2) takes 15 seconds irrespective of layer size. Hence, this downtime takes up progressively less of SmartScan's total runtime as layer size increases: 73% for 9cm^2, 60% for 16cm^2, and 49% for 25cm^2.

As demonstrated in Figure 15, we also observe a correlation between *Cosmic*'s cost benefits relative to VMs and the number of unique heat transfer matrices needed for a specific print job. VMs are cost-effective for running SmartScan when the total size of precomputed data is relatively small (left end of the graph), as the data can be stored in a few instances which can all be kept busy without much coordination overhead. When the total data size is large, the number of VMs needed is dictated by the total memory size, resulting in most of the CPUs idling during runtime. In such cases, *Cosmic* benefits from Lambda's pricing policy, which charges customers only when workers are executing code, but not during idle periods in-between when worker state remains in memory.

Cost savings compared to other serverless approaches. In all but one of the 27 print jobs where using VMs is not the most cost-effective solution, *Cosmic* outperforms all other baseline approaches that use serverless. Again, if we consider all parts to be 1cm thick, the average cost savings enabled by *Cosmic* compared to the second-cheapest solution on serverless are $0.71, $8.3, and $78.4 per part for 9cm^2, 16cm^2, and 25cm^2 layer sizes, respectively.

Ideally, *Cosmic* is expected to outperform *Invoke-on-demand*, *All-persistent*, and *All-active* in all cases because *Cosmic*'s configuration space subsumes these approaches. However, in one of the print jobs – part 10 in layer size 16cm^2 – *All-active* provides a solution that costs 0.94× as much as *Cosmic*. Although this solution is within *Cosmic*'s search space, it is not chosen because *Cosmic*'s performance model incorrectly estimates it to be slightly more expensive than the selected alternative.

5.4 Utility of individual techniques

The fact that *Cosmic* combines many techniques to deliver cost savings is evident in the wide diversity of configurations that it chooses for the 30 print jobs we consider. The number of groups into which it partitions the serverless functions that it uses varies across jobs from 1 to 50. Whereas, the degree of parallelism per group ranges from 4 to 34. To mitigate the impact of function invocation delays, *Cosmic* chooses to keep all workers active throughout the print only in 3 jobs. It employs speculation in 21 of the 30 jobs. In the remaining cases – primarily when the number of groups is less than 10 – *Cosmic* relies on the property that no invocation overhead is incurred when the cells chosen in consecutive windows belong to the same group.

5.4.1 Ablation study

To further evaluate the utility of the two key techniques that *Cosmic* combines, we start with *Invoke-on-demand* and then incrementally add grouping of functions and speculative invocation. For each of the three approaches, Figure 16 shows the number of print jobs (grouped by layer size) for which they satisfied SmartScan's timing constraints. As seen earlier, *Invoke-on-demand* is capable of satisfying these constraints in only 7 of the 30 jobs. Adding grouping addresses this problem for all printing jobs with layer sizes 9cm^2 and 16cm^2, but falls short for larger parts. Adding speculative invocation

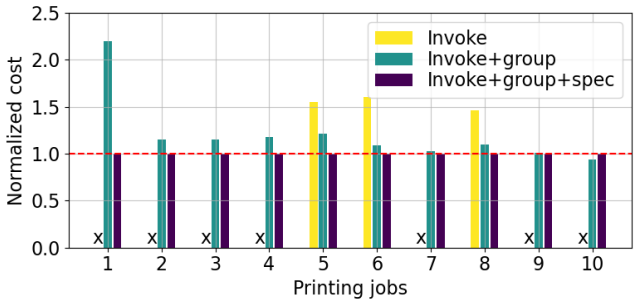


Figure 17: **For print jobs with layer size 16cm^2, cost with *Invoke-on-demand*, and with incremental addition of grouping and speculation; normalized to the cost with *Invoke-on-demand* plus grouping and speculation.**

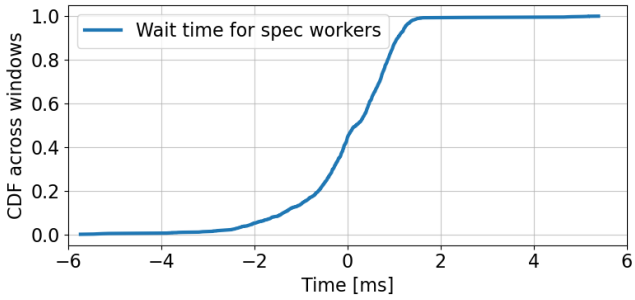


Figure 18: **For a specific print job, distribution across windows of the time from when the coordinator completes the current window until when the speculatively invoked workers are ready to start executing the next window.**

further improves the coverage to all but one job, for which all workers need to be kept persistently alive in order to ensure timely execution.

Beyond the ability to ensure timeliness, the techniques used in *Cosmic* also help reduce cost. Figure 17 demonstrates this for print jobs with 16cm^2 layer size; the results are similar for the other two layer sizes. In all 3 print jobs where *Invoke-on-demand* enables SmartScan to keep up with the printing speed, the addition of grouping lowers cost. We observe that grouping lowers runtime without increasing cost, as it increases the likelihood of reusing Lambda workers in consecutive windows, thereby avoiding invocation latency. In all but one of the printing jobs, adding speculation further reduces cost as it reduces the amount of parallelism needed to overcome the impact of invocation delays.

5.4.2 Speculative invocation

Next, we dive deeper into the efficacy of *Cosmic*'s speculative invocation of functions.

Speculation accuracy. First, across all 30 print jobs, we find that *Cosmic*'s algorithm for predicting the cell group whose functions need to be executed in the next window is 85% accurate on average. To examine the robustness of this accuracy to variance in the initial temperature map, we rerun *Cosmic*'s speculation for each part with 30 different initial temperature maps. We observed that the speculation accuracy is typically 85% $\pm$ 5%. This suggests that it is reasonable for our runtime/cost prediction model to take as input the speculation accuracy from previously completed prints of the same part.

Speculation timing. Next, we illustrate *Cosmic*'s effectiveness in using speculation to hide invocation delays at low cost. Figure 18 shows the distribution of speculative timing during the execution of SmartScan for a 16cm^2 gear shape. Specifically, we measure the gap from a) when the current window finishes, i.e., the coordinator has received outputs for the current window from all workers, until b) when the Lambda functions needed for the next window are ready to start execution, i.e., the coordinator has received a request for input from all the speculatively invoked functions. Negative values for this difference indicate early speculation, while positive values indicate late speculation. The data reveals a median value of 0.19ms, with the 5th and 95th percentiles at -2.00ms and 1.24ms, respectively. This implies that 95% of the speculation instances incur a billable idle time of no more than 2ms, and 95% of the speculation instances incur a latency overhead less than 1.24ms; both account for less than 5% of the duration of the average window in this case study. This efficiency demonstrates the balance achieved between cost and performance via accurate modeling.

6 Related Work

Applications on serverless. Prior work [8, 16] has utilized the massive parallelism enabled by serverless to significantly reduce the time needed to process videos. Other work [34] has taken advantage of serverless pricing policy to achieve time and cost savings in processing sparse graphs workloads. However, all of this work targets workloads in which each task takes tens of seconds, even minutes, to complete. To the best of our knowledge, we are the first to demonstrate the viability of serverless for workloads with latency targets in the order of tens of milliseconds.

Mitigating invocation latency. A significant body of prior work targets cold start latency in serverless computing. Some propose new optimizations that require changes to the serverless platform [14, 17, 32]. There has also been work [31] on reducing the platform's burden for keeping instances warm. Others try to avoid cold starts in the user space [8, 16, 23, 24, 30]. However, none of these prior efforts target eliminating warm start latency. We found that the latency associated with warm starts is a bottleneck in meeting millisecond-level time constraints, and we use a combination of strategies to mitigate this latency.

Performance modeling. Runtime optimization on serverless often requires accurate performance prediction. Previous efforts have modeled the combination of parallel and sequential function executions [24], used Monte Carlo simulations to predict the distribution of function costs [15], and modeled

cold start latency and execution time under different degrees of parallelism [9]. Our model accounts for a variety of delays, including function invocations and coordination overheads, and models the impact of techniques such as grouping of functions and speculative invocation.

Speculation. Speculative execution has been used to hide latency in a wide variety of networked systems, e.g., distributed file systems [27], distributed data processing [11], web browsing [25], and online gaming [21]. Speculative execution has also been employed to improve the performance of serverless applications. For example, SpecFaaS [32] pre-executes potential function calls before their control and data dependencies are resolved, and Xanadu [13] pre-invokes functions to mitigate cascading cold starts in function DAGs.

Our work developed a method for speculation specific to a 3D printing optimization algorithm, SmartScan [29]. Unlike prior work that uses speculation as a best-effort strategy to hide latency, our strategy for speculation is both time- and cost-sensitive. In addition, we use grouping of functions to improve the accuracy of speculation.

7 Discussion

Impact of changes in cloud pricing policies. *Cosmic* takes advantage of serverless computing's pay-as-you-go pricing model, which only charges for active execution time. In particular, *Cosmic* avoids the cost of retaining data in memory during the idle time between successive invocations of the same function. Presumably, serverless platforms already bake in the cost for supporting warm starts into their pricing policy, as they charge more per GB of memory than traditional VMs. These characteristics are not unique to AWS, but also exist in Azure [3], Google Cloud [5], and emerging serverless GPU inference services [4, 6], reflecting a deliberate design choice to enhance the utility of serverless, and one that is likely to persist in the near-term.

That said, if cloud providers alter their infrastructure – e.g., shortening warm start windows, or begin explicitly charging for idle memory – *Cosmic*'s design remains adaptable. To account for reduced persistence of warm workers, *Cosmic* can issue dummy requests to maintain warm containers at a negligible cost. *Cosmic* can also re-profile platform delays and pricing, adjusting invocation timing or the degree of parallelism. Though savings might decrease, many of our approach's benefits, such as minimizing the impact of invocation delays and reducing over-provisioning of resources, would still apply.

Printing larger parts. While our evaluation demonstrates *Cosmic*'s ability to meet timing constraints and reduce costs across a diverse set of print jobs, significantly larger printing jobs present additional challenges. With much greater parallelization, the overhead of communication between serverless functions and ensuring scalability without compromising timeliness are heavily constrained by the platform's specifications, and remains an area for future exploration.

Reusability of abstractions. Though we designed and evaluated *Cosmic* in the context of optimizing a specific form of 3D printing, its core abstractions and modular components are reusable in other domains: 1) *Cosmic*'s strategies for trading off invocation and coordination overheads – e.g., grouping parallelizable functions and optimizing invocation timing – apply to any latency-critical processing pipeline where each step is parallelizable. 2) *Cosmic*'s use of speculation to combat function invocation overheads is applicable in settings amenable to predictions. 3) *Cosmic*'s models for runtime and cost guided by measurement profiles and pricing policies can inform data partitioning and the degree of parallelization.

Generalizability to other workloads. *Cosmic*'s configuration framework and profiling tool, when supplied with other workload characteristics by users, can be adapted to a broad range of domains. For example, we believe *Cosmic* can be adapted for using serverless to support computations in scenarios where real-time spatial data is being analyzed, e.g., in AR/VR and online gaming. In these scenarios, models of the real/virtual world will need to be preloaded and data from user inputs or sensors must be processed quickly in order to render or make decisions in real-time. Processing of each input can be treated as similar to each window in SmartScan. Speculative execution could be used to predict inputs based on the user's trajectory, allowing for faster responses.

Another potential application of *Cosmic* could be autonomous vehicle fleet coordination, which requires real-time route planning and vehicle-to-vehicle coordination. Such systems rely on large preloaded datasets, including high-definition maps, traffic prediction models, and vehicle dynamics, to make decisions under tight timing constraints. At each step, the coordination algorithm processes these datasets to update vehicle paths dynamically, involving computation similar to SmartScan. *Cosmic* could enhance this process through speculative execution, precomputing likely scenarios – such as potential lane changes or congestion – to minimize delays.

8 Conclusion

In this paper, we demonstrated the utility of serverless computing for a new application workload: control of 3D printing from the cloud. While the pricing policy of serverless platforms make them a good match for this workload, our measurements highlighted the delays that make it challenging to meet millisecond-level time constraints. Our design of *Cosmic* combines a range of techniques with an accurate model that estimates runtime and costs. Our results showcased *Cosmic*'s utility across a diverse set of print jobs.

Acknowledgments

We thank the reviewers for their valuable feedback. This work was supported in part by NSF grant CNS-2106184 and a grant from Cisco.

References

[1] Lambda execution environments. https://docs.aws.amazon.com/lambda/latest/operatorguide/execution-environments.html, 2024.

[2] AWS Lambda. https://aws.amazon.com/lambda, 2025.

[3] Azure Functions. https://azure.microsoft.com/en-us/products/functions, 2025.

[4] Beam. https://www.beam.cloud/, 2025.

[5] Google Cloud Run. https://cloud.google.com/run, 2025.

[6] Modal. https://modal.com/, 2025.

[7] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. Firecracker: Lightweight virtualization for serverless applications. In *17th USENIX symposium on networked systems design and implementation (NSDI 20)*, pages 419–434, 2020.

[8] Lixiang Ao, Liz Izhikevich, Geoffrey M Voelker, and George Porter. Sprocket: A serverless video processing framework. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 263–274, 2018.

[9] Rohan Basu Roy, Tirthak Patel, Richmond Liew, Yadu Nand Babuji, Ryan Chard, and Devesh Tiwari. ProPack: Executing concurrent serverless functions faster and cheaper. In *Proceedings of the 32nd International Symposium on High-Performance Parallel and Distributed Computing*, pages 211–224, 2023.

[10] Marc Brooker, Adrian Costin Catangiu, Mike Danilov, Alexander Graf, Colm MacCárthaigh, and Andrei Sandu. Restoring uniqueness in microvm snapshots. *arXiv preprint arXiv:2102.12892*, 2021.

[11] Qi Chen, Cheng Liu, and Zhen Xiao. Improving MapReduce performance using smart speculative execution strategy. *IEEE Transactions on Computers*, 63(4):954–967, 2013.

[12] Sohini Chowdhury, N Yadaiah, Chander Prakash, Seeram Ramakrishna, Saurav Dixit, Lovi Raj Gupta, and Dharam Buddhi. Laser powder bed fusion: a state-of-the-art review of the technology, materials, properties & defects, and numerical modelling. *Journal of Materials Research and Technology*, 20:2109–2172, 2022.

[13] Nilanjan Daw, Umesh Bellur, and Purushottam Kulkarni. Xanadu: Mitigating cascading cold starts in serverless function chain deployments. In *Proceedings of the 21st International Middleware Conference*, pages 356–370, 2020.

[14] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 467–481, 2020.

[15] Simon Eismann, Johannes Grohmann, Erwin Van Eyk, Nikolas Herbst, and Samuel Kounev. Predicting the costs of serverless workflows. In *Proceedings of the ACM/SPEC international conference on performance engineering*, pages 265–276, 2020.

[16] Sadjad Fouladi, Riad S Wahby, Brennan Shacklett, Karthikeyan Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and Keith Winstein. Encoding, fast and slow: Low-latency video processing using thousands of tiny threads. In *NSDI*, volume 17, pages 363–376, 2017.

[17] Alexander Fuerst and Prateek Sharma. FaasCache: keeping serverless computing alive with greedy-dual caching. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 386–400, 2021.

[18] Runsheng Guo, Victor Guo, Antonio Kim, Josh Hildred, and Khuzaima Daudjee. Hydrozoa: Dynamic hybrid-parallel dnn training on serverless containers. *Proceedings of Machine Learning and Systems*, 4:779–794, 2022.

[19] Liang Hao, Wanlin Wang, Jie Zeng, Min Song, Shuai Chang, and Chenyang Zhu. Effect of scanning speed and laser power on formability, microstructure, and quality of 316L stainless steel prepared by selective laser melting. *Journal of Materials Research and Technology*, 25:3189–3199, 2023.

[20] Jigang Huang, Qin Qin, and Jie Wang. A review of stereolithography: Processes and systems. *Processes*, 8(9):1138, 2020.

[21] Kyungmin Lee, David Chu, Eduardo Cuervo, Johannes Kopf, Yury Degtyarev, Sergey Grizan, Alec Wolman, and Jason Flinn. Outatime: Using speculation to enable low-latency continuous interaction for mobile cloud gaming. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*, pages 151–165, 2015.

[22] Jiangwei Liu, Yanan Song, Chaoyue Chen, Xiebin Wang, Hu Li, Jiang Wang, Kai Guo, Jie Sun, et al. Effect of

scanning speed on the microstructure and mechanical behavior of 316L stainless steel fabricated by selective laser melting. *Materials & Design*, 186:108355, 2020.

[23] Xuanzhe Liu, Jinfeng Wen, Zhenpeng Chen, Ding Li, Junkai Chen, Yi Liu, Haoyu Wang, and Xin Jin. Faaslight: General application-level cold-start latency optimization for function-as-a-service in serverless computing. *ACM Transactions on Software Engineering and Methodology*, 32(5):1–29, 2023.

[24] Ashraf Mahgoub, Edgardo Barsallo Yi, Karthick Shankar, Sameh Elnikety, Somali Chaterji, and Saurabh Bagchi. ORION and the three rights: Sizing, bundling, and prewarming for serverless DAGs. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 303–320, 2022.

[25] James W Mickens, Jeremy Elson, Jon Howell, and Jay R Lorch. Crom: Faster web browsing using speculative execution. In *NSDI*, volume 10, pages 9–9, 2010.

[26] Amir Mostafaei, Amy M Elliott, John E Barnes, Fangzhou Li, Wenda Tan, Corson L Cramer, Peeyush Nandwana, and Markus Chmielus. Binder jet 3d printing—process parameters, materials, properties, modeling, and challenges. *Progress in Materials Science*, 119:100707, 2021.

[27] Edmund B Nightingale, Peter M Chen, and Jason Flinn. Speculative execution in a distributed file system. *ACM SIGOPS operating systems review*, 39(5):191–205, 2005.

[28] Kumaresan Rajan, Mahendran Samykano, Kumaran Kadirgama, Wan Sharuzi Wan Harun, and Md Mustafizur Rahman. Fused deposition modeling: process, materials, parameters, properties, and applications. *The International Journal of Advanced Manufacturing Technology*, 120(3):1531–1570, 2022.

[29] Keval S Ramani, Chuan He, Yueh-Lin Tsai, and Chinedum E Okwudire. Smartscan: An intelligent scanning approach for uniform thermal distribution, reduced residual stresses and deformations in pbf additive manufacturing. *Additive Manufacturing*, 52:102643, 2022.

[30] Rohan Basu Roy, Tirthak Patel, and Devesh Tiwari. DayDream: executing dynamic scientific workflows on serverless platforms with hot starts. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–18. IEEE, 2022.

[31] Rohan Basu Roy, Tirthak Patel, and Devesh Tiwari. Icebreaker: Warming serverless functions better with heterogeneity. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 753–767, 2022.

[32] Jovan Stojkovic, Tianyin Xu, Hubertus Franke, and Josep Torrellas. Specfaas: Accelerating serverless applications with speculative function execution. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 814–827. IEEE, 2023.

[33] Akihiro Takezawa, Honghu Guo, Ryotaro Kobayashi, Qian Chen, and Albert C To. Simultaneous optimization of hatching orientations and lattice density distribution for residual warpage reduction in laser powder bed fusion considering layerwise residual stress stacking. *Additive Manufacturing*, 60:103194, 2022.

[34] John Thorpe, Yifan Qiao, Jonathan Eyolfson, Shen Teng, Guanzhou Hu, Zhihao Jia, Jinliang Wei, Keval Vora, Ravi Netravali, Miryung Kim, et al. Dorylus: Affordable, scalable, and accurate GNN training with distributed CPU servers and serverless threads. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 495–514, 2021.

[35] Ao Wang, Jingyuan Zhang, Xiaolong Ma, Ali Anwar, Lukas Rupprecht, Dimitrios Skourtis, Vasily Tarasov, Feng Yan, and Yue Cheng. InfiniCache: exploiting ephemeral serverless functions to build a cost-effective memory cache. In *18th USENIX conference on file and storage technologies (FAST 20)*, pages 267–281, 2020.

A Appendix

The cost for a VM per window is formulated as:

$$F_{vm} = (n \cdot m) \cdot (t + 15\ s) \cdot c_{vm}\ ,$$

where

- n = number of unique matrices,
- m = size of in-memory precomputed data per cell in GB,
- t = time constraint in seconds,
- c_{vm} = cost per GB of memory per unit of time for the VM.

It's important to note that VMs must stay operational between layers, hence the additional 15 seconds of billed time. This formula also assumes full utilization of the VM's memory capacity.

In contrast, the cost for Lambda per window is:

$$F_{\lambda} = M \cdot (t) \cdot c_{\lambda}\ ,$$

where M is the total memory capacity of Lambda workers running in parallel, and assuming Lambda functions are configured to complete computations precisely within the time constraint t. Unlike VMs, where the scale of operation is dictated by the total data size, the number of Lambda functions

running in parallel during each window is determined primarily by the need to meet stringent time constraints. Consequently, this total capacity M often exceeds what is necessary for storing the precomputed data per cell m. Given that we use the largest Lambda instances, we have $M = N_P \cdot 10\ GB$, where N_P is the number of parallelism in each window.

Solving $F_{vm} > F_{\lambda}$ simplifies to:

$$n \cdot m > \frac{t}{t+15} \cdot N_P \cdot 92\ GB\ . \tag{1}$$

Here, the left side represents the total size of precomputed data. Taking the example of the 16cm^2 gear shaped print job, this inequality holds true as $1514\ GB > 404.8\ GB$, with $n = 127$, $m = 11.9\ GB$, $t = 10\ s$, and $N_P = 11$. Qualitative interpretation of this inequality shows that Lambda is potentially more cost-effective than VM under certain conditions: 1) a higher total size of precomputed data, 2) a lower time constraint relative to powder spreading time, and 3) fewer parallel workers required in each window.

ARTIFACT EVALUATED usenix ASSOCIATION AVAILABLE
ARTIFACT EVALUATED usenix ASSOCIATION FUNCTIONAL

GMI-DRL: Empowering Multi-GPU DRL with Adaptive-Grained Parallelism

Yuke Wang, Boyuan Feng[‡], Zheng Wang[◇], Guyue Huang[‡], Tong Geng[†], Ang Li[*], and Yufei Ding[◇]
[†]*University of Rochester,* [*]*Pacific Northwest National Laboratory,*
[‡]*UC at Santa Barbara,* [◇]*UC at San Diego,*
Rice University

Abstract

With the increasing popularity of robotics in industrial control and autonomous driving, deep reinforcement learning (DRL) raises the attention of various fields. However, DRL computation on the modern powerful multi-GPU platform is still inefficient due to its heterogeneous tasks and complicated inter-task interactions. To this end, we propose **GMI-DRL**, the first systematic design for scaling multi-GPU DRL via adaptive-grained parallelism. To facilitate such a new parallelism scheme, GMI-DRL introduces a new concept – *GPU Multiplexing Instance* (GMI), a unified resource-adjustable sub-GPU design for heterogeneous tasks in DRL scaling. Besides, GMI-DRL introduces an adaptive *Coordinator* to effectively manage workloads and resources for better system performance. GMI-DRL also incorporates a specialized *Communicator* with highly efficient inter-GMI communication support to meet diverse communication demands. Extensive experiments demonstrate that GMI-DRL outperforms state-of-the-art DRL accelerating solution in training throughput (up to 2.34×) and GPU utilization (up to 40.8% improvement) on the DGX-A100 platform.

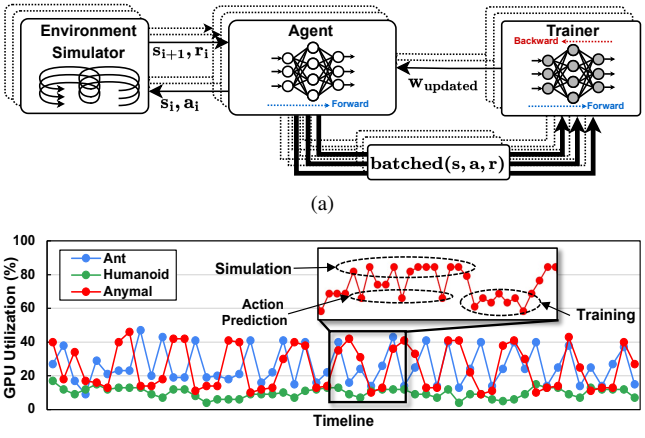
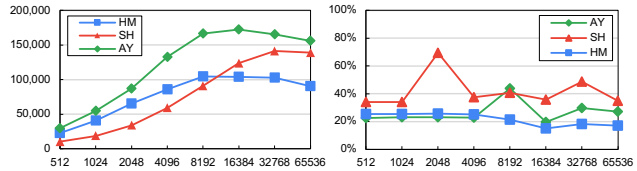


Figure 1: (a) Basic DRL computation flow in PPO [39] with three major components: *simulator*, *agent*, and *trainer*. Note that the dotted-outline boxes/arrows indicate DRL training copies. $\underline{s}$: the environment **state**; $\underline{a}$: the agent **action**; $\underline{r}$: the action **reward**; $\underline{w}$: the policy **weight**; $\underline{batched(s,a,r)}$: the **experience** data.; (b) averaged GPU utilization of Isaac Gym for PPO training on a DGX-A100 platform.

1 Introduction

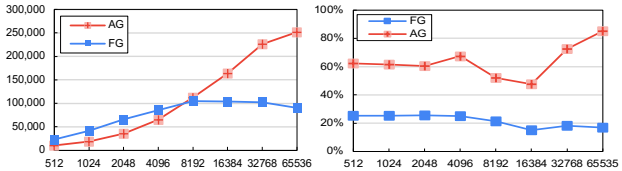
Recently, deep reinforcement learning (DRL) has raised significant interest in research and industry fields. DRL combines the conventional RL algorithms [25, 39] with deep neural networks (DNNs) and has demonstrated superhuman performance in decision-making. DRL has been applied in many real-world applications across various domains, like robotics [12,18,33,44,50], industrial control [4,8,41,51], and autonomous driving [5,20,38,48]. The latest DRL application highlighted most is OpenAI ChatGPT [33], which is trained with DRL from human feedback. Due to the cost, efficiency, and safety concerns in the real world, DRL training is commonly conducted in simulated environments on computers. Then the pretrained DRL model (policy) is fine-tuned and deployed to the real-world environment [15, 24].

Unlike the conventional DL methods (e.g., DNNs) that learn from offline collected datasets, DRL learns from online collected data generated through model-environment interactions. DRL computation features heterogeneous workloads and multi-granularity communication patterns (Figure 1(a)). It interleaves the execution of three components: *Environment Simulator*, *Agent*, and *Trainer*. Agents will interact with environment simulators to collect enough experience (consisting of state, action, and reward). After a number of (e.g., 32) agent-environment interactions, the agent will communicate with a trainer to update the policy model. The above three components feature distinct computation patterns. The environment simulator relies on physics simulation (e.g., local motion and robotics control) while the agent and trainer depend on GEMM-based NN operations (e.g., policy model inference/training).



(a) (b)

Figure 2: (a) Throughput (samples/seconds) and (b) GPU utilization (via `nvidia-smi`) for DRL training scaling (increasing batch size) across three different DRL benchmarks.



(a) (b)

Figure 3: Training Comparison between fine- (FG) and adaptive-grained (AG) parallelism on (a) throughput and (b) GPU utilization on the Humanoid [46] (HM) benchmark.

To catch up with the increasingly intricate and dynamic real-world scenarios with high fidelity, modern DRL applications (e.g., power-grid control [4] and robotics [24]) tend to be more complex (enormous action and state space) and require millions/billions of interaction steps for the final convergence. This motivates the urgent demands for scaling up DRL training on more powerful multi-GPU platforms (e.g., NVIDIA DGX [29]), typically equipped with many (8 or 16) large-size high-end GPUs (e.g., A100 and H100). Such platforms are also the key AI computing building blocks (for more than 60% of machines on the latest Top500 list [42]) and the future trend of DL scaling.

However, existing DRL designs are still yet to maximize the computing benefits of those powerful platforms. For illustration, we profile the popular PPO [39] algorithm from the state-of-the-art Isaac Gym [24] on NVIDIA DGX-A100 (8 GPUs). Figure 1(b) gives the average utilization of GPUs on three representative DRL benchmarks over 10 training epochs. We can see that the multi-GPU training of DRL exhibits both *temporally low GPU utilization* (where a DRL training flow would only perform intermittent DNN model training/inference per epoch) and *spatially low GPU utilization* (where the computation/memory demands of DRL training flow could hardly occupy the entire GPU). This indicates that there is a large portion of the computing power of GPU platforms being throttled by the existing designs. DRL performance can potentially be improved by turning those "spare" GPU resources into full utilization.

The common practice to overcome such issues is to adapt computation to fit the underlying resources (i.e., scaling through fine-grained parallelism) [7, 24]. For example, we can increase the simulation batch size to promote DRL training throughput. Figure 2 shows that when first increasing the simulation/training batch size, the performance would increase proportionally w.r.t. the batch size. The overall throughput performance degrades when further increasing the batch size beyond a certain threshold (e.g., 16,384 for Anymal [16] (AY)). Such an increase-followed-by-decrease performance trend is similar across benchmarks and is also observed in GPU utilization. This is mainly because when increasing the batch size, individual DRL components will scale up their resource demands (e.g, SMs, memory/cache) for more task-level concurrency (e.g., DRL simulation). Different DRL components running in parallel would compete for a fixed amount of GPU resources. GPU utilization would suffer since different DRL components are busy waiting for some common resources to move forward. For instance, the GEMM-based DRL policy training/inference would wait for SMs occupied by DRL simulation. The performance penalty of such contention would offset the parallelism benefits after reaching a certain threshold.

Such an observation motivates us to think about whether it is a better option to tailor resources for computation demands, so that we can seek for optimal compute-resource efficiency instead of blindly increasing the batch size. Our key insight is to explore adaptive-grained parallelism (AGP) on GPUs. AGP opens a new way for flexible utilization of GPU resources, where a large GPU can be used as a set of size-adjustable sub-GPUs for different tasks. AGP also enlarges the existing design and optimization space for using individual GPUs. For instance, we can treat one GPU as a multi-sub-GPU system for computation and communication at the sub-GPU level. When scaling up to multiple GPUs, AGP is also an effective way for better resource management of different workloads. It can facilitate the creation of the best-fit GPU resource plans, given the specified number of GPUs and the GPU affinity demand (e.g., NVLinks).

However, AGP is not a low-hanging fruit. It is naturally hindered by the memory isolation among different sub-GPUs due to GPU spatial multiplexing. This would be the performance killer for DRL with heterogeneous tasks and substantial inter-task communication. Efficient communication at the sub-GPU level is still missing from existing high-performance GPU communication libraries (e.g., NCCL [28]). Such communication hurdles may offset computation gains of GPU spatial multiplexing. Besides, finding the right-fit parallelism granularity (sub-GPU size) for individual tasks is non-trivial. The mismatch between sub-GPUs and their assigned DRL workloads would easily degrade the overall performance while wasting valuable GPU resources (e.g., SM). Furthermore, identifying the optimal layout of sub-GPUs requires careful consideration of the specialties of workloads (e.g., communication) and hardware (e.g., GPU interconnections).

To this end, we introduce **GMI-DRL**, the first end-to-end system for scaling multi-GPU DRL via adaptive-grained parallelism (AGP), with significantly improved throughput and

GPU utilization compared to fine-grained parallelism (Figure 3). To facilitate AGP, GMI-DRL introduces a new concept – *GPU Multiplexing Instance* (GMI), a unified resource-adjustable sub-GPU design tailored for heterogeneous tasks. GMI-DRL also provides systematic support for communication, computation, and programming for GMIs. GMI-DRL automates DRL scaling while respecting the black-box nature of DRL simulation (which commonly takes months/years of effort for re-design and implementation).

In short, we make the following contributions

- We introduce a novel adaptive-grained parallelism scheme for DRL scaling and propose GPU Multiplexing Instance (GMI) to facilitate such a new scheme.
- We build a GMI-centric adaptive *Coordinator* (§3.1) and a specialized *Communicator* (§3.2) design to meet diverse communication demands among GMIs, to achieve better system performance and GPU utilization.
- We also provide GMI-centric programming support (§3.3) to ease the scaling of diverse DRL-like applications on multi-GPU platforms.
- Extensive experiments show that GMI-DRL outperforms the state-of-the-art DRL acceleration solution (Isaac Gym [24] + MSRL [52]) across various benchmarks.

2 Background

2.1 Deep Reinforcement Learning

Deep Reinforcement Learning (DRL) [25, 39, 49] features heterogeneous computational workloads and diverse communication patterns. Figure 1 exemplifies the workflow of a typical PPO [39] algorithm in DRL. It interleaves the execution of three components: *Environment Simulator*, *Agent*, and *Trainer*. The environment simulator is responsible for generating environment states and action rewards with the help of its internal physics simulation engine (e.g., local motion and robotics control). GPU-based simulators (e.g., Isaac Gym [24]) can hold hundreds or thousands of environments running concurrently. The agent is the key component to making action decisions based on the environment states. It depends on GEMM-based NNs (e.g., Multi-Layer Perceptron) for policy model inference (action prediction). DRL trainer updates the policy model based on experience (with states, actions, and rewards) from agents. There are two major computing modes of DRL in real-world settings.

DRL Serving: Agents interact with environment simulators by applying the pre-trained policy model. The interaction between simulators and agents demonstrates fine-grained state-action sharing. DRL serving is also essential for experience generation for the DRL training process on computers and the real-world evaluation of the DRL policy.

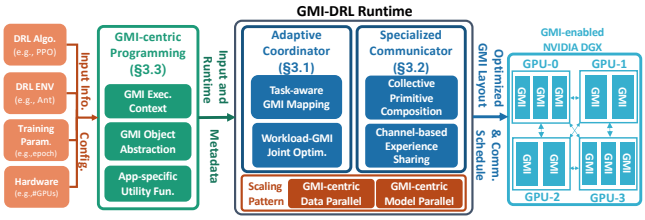


Figure 4: Overview of the GMI-DRL system.

DRL Training: Agents first interact with simulators to collect experience. After a number (e.g., 32) of interactions, agents will communicate with trainers via coarse-grained experience sharing for policy updates. DRL training aims to maximize the accumulated reward of agent actions by updating the policy iteratively. Depending on whether the policy model training happens sequentially or concurrently with each round of agent-simulator interaction, DRL training can be categorized into *synchronized* (e.g., PPO [39]) and *asynchronized* DRL training (e.g., A3C [25]).

2.2 Adaptive-grained Parallelism (AGP)

We propose AGP as a new type of parallel computing paradigm that can adjust the size or granularity of the parallel tasks to optimize performance. While the existing scaling schemes are commonly confined to a certain level of granularity for parallelism. For instance, existing data parallelism explores coarse-grained dataflow-level parallelization, model parallelism explores the middle-grained operator execution concurrence, and operator fusion/grouping explores fine-grained kernel-level parallelization. AGP is complementary to those existing schemes by adding parallelization hierarchies with "configurable" granularity.

AGP can be broadly categorized into two major types: 1) adjust the tasks' workloads (e.g., batch size in DNNs) to fit the hardware; 2) adjust the hardware resource size to fit the given workload. In DRL, the first one is hard to achieve (Figure 2) due to its intrinsically complicated simulator design and mixed workload. We thus advocate solution-2 for performance, resource efficiency, and design complexity. When targeting other workloads, solution-2 can be combined orthogonally with the existing solution-1 for a larger granularity space for diverse settings.

For AGP, *GPU Spatial Multiplexing* is the most potential candidate with adjustable resource granularity and minimized runtime interference. There are two major techniques: 1) Multi-processing Service (**MPS**) [31] can transparently enable cooperative multi-process CUDA applications to share a GPU. MPS provides concurrency among client processes to share a GPU via GPU SMs partitioning. 2) Multi-Instance GPU [30] is a new GPU spatial multiplexing solution introduced in NVIDIA Ampere GPUs (e.g., A100). Multi-Instance GPU offers higher isolation (SMs, memory space/bandwidth) compared to MPS. However, the existing practices of GPU

spatial multiplexing are limited to their initially targeted settings with *independent* and *homogeneous* tasks, like DNN serving [6, 22, 45], thus, unable to offer communication support and strategies for inter-dependent task mapping. Comparatively, we are the first to explore the potential of GPU spatial multiplexing for more challenging DRL applications with *heterogeneous* tasks and intricate *inter-task interactions*.

3 GMI-DRL Design

GMI-DRL (Figure 4) reshapes the existing GPU-based DRL scaling with a new notion of GMIs. To support such new scaling schemes, GMI-DRL introduces an adaptive *Coordinator* (§3.1) to manage the workload-resource mapping and configuration optimization and a specialized *Communicator* (§3.2) to handle diverse inter-task interactions.

3.1 Adaptive Coordinator

The key aspect of AGP is to generate a good mapping choice between DRL components and underlying computing resources. This motivates the GMI-DRL *Coordinator* to systematically combine coarse-grained mapping and fine-grained configuration tuning for maximized overall performance.

3.1.1 Task-aware GMI Mapping

The mapping of DRL tasks on GMIs demands careful consideration of the specialties of various DRL tasks and hardware, which requires the mapping that will not only benefit individual tasks but also complex inter-task interactions. For instance, using dedicated GMI instances for simulators can achieve high simulation task performance. However, it may lead to inferior inter-task performance due to the inefficient data communication between simulators and agents on other GMIs. We, therefore, introduce a *task-aware GMI mapping* strategy to explore the GMI-backed scaling design space.

DRL Serving DRL serving aims to collect enough experience by applying a trained policy model. DRL serving features continuous interaction between simulators and agents. Optimizing DRL serving aims to achieve *low latency* at each round of interaction between the simulator and agent so that experience can be collected quickly. The interaction between simulators and agents relies on *frequent and fine-grained* information exchange for high serving performance. With these observations, we explore the potential mappings with the synergy of data (DP) and model parallelism (MP).

1) *DP-MP Serving* (Figure 5a) uses DP as the backbone for running multiple copies of DRL serving and leverages MP to distribute individual components of each copy to different GMIs. This guarantees the performance of individual components but at a heavy penalty of inter-component performance,

Table 1: Definition of Terms.

Term	Meaning
R_s, R_a, R_t	the size of dominant resources for the simulator, agent, and trainer, respectively.
$\mathbb{I}$	the type of dominant resource: SM or Memory.
S, A, W	the size of a single state, action, and reward vector, respectively.
T_s, T_a, T_t	the execution time for the simulator, agent, and trainer in each DRL training/serving iteration.
BW	the bandwidth of inter-GMI communication.
M_p	the size of the policy model.
m	the number of simulation steps before a training.
n	the total number of GMIs in the current system.
α, β	the size ratio factor when multiple environment simulators are sharing agents or trainers.

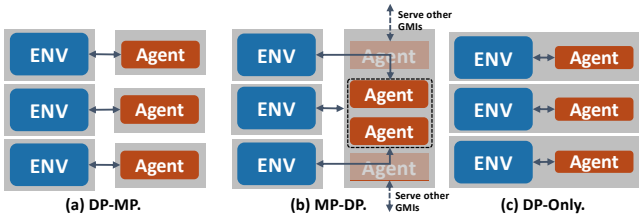


Figure 5: GMI mapping for DRL serving. Note that Grey shadows indicate GMIs.

since the fine-grained, frequent data movement between GMIs would incur high overhead.

2) *MP-DP Serving* (Figure 5b) leverages MP for inter-phase scaling for decoupled simulators and agents and utilizes DP for intra-phase scaling to run multiple copies of simulators and agents. The key insight is that the throughput of simulators and agents is naturally different, where an NN-based agent comes with high throughput and can thus be "shared" among different simulators which have relatively low throughput. With this insight, we can achieve higher resource efficiency by accommodating more simulators and agents to improve the theoretical performance. However, the communication between simulators and agents from different GMIs would still throttle the practical system performance.

3) *DP-only Serving* (Figure 5c) relies on DP for scaling. It runs multiple copies of DRL serving in parallel. In this design, simulators and agents are co-located within the same GMI. Compared with the prior two designs, the *state* data generated from the simulator can be easily shared with the subsequent agent for action prediction via low-cost intra-GMI memory access. However, it comes at the cost of additional resource consumption and potential resource underutilization. Since different GMIs have to maintain a full copy of the simulator and agent, each agent can only serve one simulator.

Serving Resource-Performance Analysis is to quantify the cost and benefits of different mapping solutions to guide our mapping selection. Terms are defined in Table 1. Here, we compare MP-DP and DP-only solutions for illustration. We first determine the dominated resources type ($\mathbb{I}$) based on

Table 2: Comparing MP-DP and DP-only in DRL serving.

Solution	Resource ($R^{\mathbb{I}}$)	Comm. (COM)
MP-DP	$(T_s \cdot R_s + T_a \cdot \alpha \cdot R_a)/(T_s + T_a)$	$2 \cdot S + A + W$
DP-only	$(T_s + T_a) \cdot \max\{R_s, R_a\}/(T_s + T_a)$	0

Equation 1 for R_e, R_a, and R_t, which can be either GPU SM or memory, depending on DRL benchmarks and GPUs.

$$\mathbb{I} = \begin{cases} SM & \frac{R_{SM}}{R_{SMperGPU}} \geq \frac{R_{Mem}}{R_{MemPerGPU}} \\ Memory & Otherwise. \end{cases} \quad (1)$$

R_{SM} and R_{Mem} are computing and memory resource consumption measured on a single DRL serving/training process running exclusively on a GPU. Take the DRL benchmarks from Isaac Gym on DGX-A100 as an example, our empirical study shows that α is around 0.2, $R_s \approx 10R_a$, and $T_s \approx 6T_a$. Thus, the overall serving throughput can be calculated as

$$TOP_{serving} = \frac{R^{\mathbb{I}}_{all}}{R^{\mathbb{I}}} \cdot \frac{1}{T_s + T_a + \frac{COM}{BW}} \quad (2)$$

With Table 2 and $\frac{COM}{BW} \approx 2 \cdot (T_s + T_a)$ based on our profiling, we can estimate that the overall serving throughput of DP-only serving would be higher (about 2.5×) compared with MP-DP serving. This is because the resource penalty (i.e., decrease in parallelism) of simulator-agent co-locating on throughput is minor ($(T_s + T_a) \cdot \max\{R_s, R_a\}/(T_s \cdot R_s + T_a \cdot \alpha \cdot R_a) - 1 \approx 0.16\times$) compared with the throughput benefits (around 3×) by saving inter-GMI data movements. We experimentally validate this in §4.3.

Synchronized DRL Training Synchronized DRL training has two stages, 1) simulator-agent interaction for experience collection and 2) agent-trainer interaction for policy training. These two stages happen sequentially at each iteration and the training performance is sensitive to both stages. The major goal of optimizing synchronized DRL training is to achieve higher model update frequency. Compared with DRL serving, synchronized DRL training is more complex in its components and inter-component interactions. This brings more tradeoffs for GMI mapping. We first discuss the ones that extend the existing DRL serving mapping principles and then introduce the new ones specific to DRL training.

1) *DP-MP and DP-only Training* (Figure 6a and 6b) follow similar scaling principles as the DRL serving. For DP-MP scaling, the size of communication would increase significantly compared with DRL serving due to the additional agent-trainer interaction for experience sharing and inter-trainer interaction for policy gradient synchronization. In DP-only scaling, it aligns the environment, agent, and trainer in the same GMI. Despite that it can enjoy the benefit of saving data movements, it would incur additional memory costs of duplicating both agents and trainers.

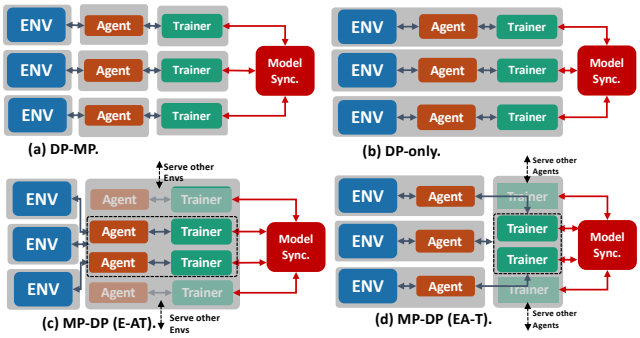


Figure 6: GMI mapping for Synchronized DRL Training scaling. Note that Grey shadows indicate GMIs.

Table 3: DP-MP (EA-T) and DP-only in Sync. DRL training.

Solution	Resource ($R^{\mathbb{I}}$)	Comm. (COM)
DP-MP (EA-T)	$(T_s \cdot R_s + T_a \cdot \alpha \cdot R_a + T_t \cdot \beta \cdot R_t)/(T_s + T_a + T_t)$	$m \cdot (S + A + W) + M_p + 2 \cdot (n-1) \cdot M_p/n$
DP-only	$(T_s + T_a + T_t) \cdot \max\{R_s, R_a, R_t\}/(T_s + T_a + T_t)$	$2 \cdot (n-1) \cdot M_p/n$

2) *DP-MP Training with Selective Pipeline Partitioning* (Figure 6c and 6d) enables the sharing of common resources like agents and trainers among different training instances. There are two major types of DP-MP training based on their training pipeline partitioning. The first one is E-AT, which partitions the DRL training pipeline between simulators (E) and agents (A). Such partitioning enables the sharing of agents and trainers among different simulators. The second type is EA-T, which partitions the DRL training pipeline between agents and trainers (T). It enables sharing of trainers among different training instances. With more component sharing, such as E-AT, the entire system could accommodate more training instances to achieve higher overall theoretical performance. While with less component sharing, such as EA-T, the communication critical path between the simulator and agent is no longer the bottleneck.

Training Resource-Performance Analysis on synchronized DRL training follows similar steps as DRL serving, except that we need to include the impact of the trainer on the pipeline. Here we compare DP-MP (EA-T) and DP-only training and take as an example the DRL benchmarks from Isaac Gym on DGX-A100. Our empirical study shows that α is around 0.2, β is around 0.3, $R_s \approx 10R_a \approx 5R_t$, and $T_s \approx 6T_a \approx 3T_t$, the overall throughput is calculated as

$$TOP_{train} = \frac{R^{\mathbb{I}}_{all}}{R^{\mathbb{I}}} \cdot \frac{1}{T_s + T_a + T_t + \frac{COM}{BW}} \quad (3)$$

With Table 3 and $\frac{COM}{BW} \approx 7 \cdot (T_s + T_a + T_t)$ based on our profiling, we can derive that the overall system throughput of DP-only training would increase evidently (about 5×) compared with DP-MP (EA-T). This is because the resource penalty on throughput ($(T_s + T_a + T_t) \cdot \max\{R_s, R_a, R_t\}/(T_s \cdot R_s + T_a \cdot \alpha \cdot R_a + T_t \cdot \beta \cdot R_t) - 1 \approx 0.5\times$) is offset by throughput benefits

(around 8×) via avoiding unnecessary inter-GMI experience movements. We provide experimental validation in §4.3. For other DRL benchmarks on different platforms, the lightweight profiling over several training rounds is sufficient to get the above parameters to analyze different training mappings.

Asynchronized DRL Training Asynchronized DRL training concurrently executes policy serving for experience collection and policy updates. However, it comes with the staleness of policy parameters that may impact the algorithmic efficiency (e.g., convergence rate). The common practice is to maximize throughput so that all DRL agents can get the model trained with the latest experience. We choose the mapping template from Figure 6d, with decoupled serving and training phase for the minimum communication bottleneck. We use designs in §3.2.1 for latency-optimized inter-trainer synchronization and §3.2.2 for throughput-optimized experience sharing between agents and trainers.

3.1.2 Workload-GMI Joint Optimization

After determining the efficient mapping templates, properly matching DRL workloads with right-size GMIs is also of great importance. However, there are two major challenges: *1) How to model the relation between the DRL training performance and GMI resources?* The performance of a single GMI depends on configurations of both the DRL algorithmic and the GMI resource, which creates a large design space for exploration. *2) How to balance the single-GMI performance and the overall system performance?* The high single-GMI performance does not always imply "optimal" system performance due to its excessive resource consumption and missing effective global planning to jointly optimize resources and performance. We introduce a *workload-GMI joint optimization* to co-optimize the configuration of the DRL tasks and the GMI resources for maximizing system throughput.

Configurations of DRL tasks and GMI resources. There are two key configurations that will impact the performance and resources of GMI-based DRL. i) *num_env* describes the number of concurrent environments running at each GMI. It is equivalent to the batch size in conventional DNN training. The common choices for *num_env* range from 128 to 32,768 considering the algorithmic (convergence) and system (runtime execution) efficiency [24]. Generally, a larger value of *num_env* would lead to higher throughput. However, when all the computation resources of a GMI are fully occupied, further increasing the *num_env* would only increase the memory consumption while being largely throttled by computation resources. This would lead to minor or even no throughput improvement. ii) *GMIperGPU* describes the quota of computation and memory resources assigned to each GMI. Computation resources determine the throughput performance of DRL computation, since the simulation and the training are computationally intensive and resource sensitive. Comparatively, memory resources mainly determine whether the DRL training can be successfully launched on GPUs, since DRL components have their own minimum memory requirements for basic execution without considering performance.

Algorithm 1: Profiling-based GMI Exploration.

```
   input  : DRL_bench, num_GPU
   output : num_env, GMIperGPU
1  best_config = tuple(); max_top = −inf;
2  for GMIperGPU in 10 ... 1 do
3      pre_top = 0; pre_mem = 0;
4      for num_env in [128, 256, 512, ..., 16384, 32768] do
5          ▷ Filter out the GMI OOM/crashing cases..
6          if num_env>=512 && projMem(num_env) >
             GPU_mem/GMIperGPU then
7              break;
8          end
9          ▷ Profile the performance of a GMI..
10         top, mem = profile(DRL_bench, GMIperGPU, num_env);
11         ▷ Initialize tracking variables..
12         if pre_top == pre_mem == 0 then
13             pre_top = top; pre_mem = mem;
14             continue;
15         end
16         ▷ Compute performance/resource changes..
17         R_top = (top − pre_top)/(pre_top);
18         R_mem = (mem − pre_mem)/(pre_mem);
19         Sat = R_top/R_mem;
20         pre_top = top; pre_mem = mem;
21         ▷ Check if the performance saturates..
22         if Sat < α then
23             break;
24         end
25         ▷ Project the overall system throughput..
26         acc_top = estimate(GMIperGPU, num_GPU, top);
27         if acc_top > max_top then
28             max_top = acc_top;
29             best_config = (num_env, GMIperGPU);
30         end
31     end
32 end
33 num_env, GMIperGPU = best_config[0], best_config[1];
```

Workflow of Joint Optimization. We will take the environment specification (e.g., name and type) and the *num_GPUs* as the input, and return *num_env* and *GMIperGPU* as the desired runtime configuration. As illustrated in Algorithm 1, we will iterate through the GMI resource space by adjusting *GMIperGPU* (Line 4). For instance, when *GMIperGPU* is 4, the resources of each GPU will be divided into 4 GMIs evenly. Another design dimension to be explored is the number of environments (*num_env*). These two dimensions will help us to pinpoint the design that can maximize the throughput given the GMI resource budget. To reduce unnecessary searches, we develop new metrics and heuristics for an early stop, as detailed in the following paragraphs.

We monitor memory at the first two steps of *num_env* (128 and 256 are unlikely to trigger out-of-GMI-memory based on our study). Then for all the following steps of *num_env*>= 512, we project its memory (m_p) based on the prior two steps of *num_env* (e_1, e_2) and memory (m_1, m_2), as shown in Equation 4. When m_p exceeds the GMI memory

limits, we avoid searching for even larger *num_env*.

$$m_p = \frac{(e - e_2)}{e_2 - e_1} \cdot (m_2 - m_1) + m_2 \quad (4)$$

The *profile* function will get the throughput (*top*) and memory cost (*mem*) of each configuration. We will then compare the change in computation throughput and the change in memory consumption between its current run and its last run (Line 17 to 18). Our empirical study (§4.3) shows that a GMI tends to saturate its computation capacity when we increase the *num_env* and observe (i) the large increase of memory consumption (ii) the minor improvement of *top*.

Algorithm 1 is currently running offline, given that real-world DRL training usually takes thousands of iterations (hours) for convergence. The major cost of profiling (several minutes) is minor compared to the entire training. We count down from 10 to 1 so as to prioritize (i.e., greedy search) the testing of the upper bound values of the number of GMIs, such that we can early terminate the search, which can make each of our experimental trials worthy and save efforts.

We also perform fine-grained measurement of GMI overhead across our evaluated datasets (§ 4) and policy NNs on the DGX-A100 platform. Specifically, the creation of the GMIs (1, 2, and 4) instance is low-cost (0.05s on average), which is largely minor compared to the typically hours of the DRL training process. Besides, the communication overhead is about 16% on average across GMIs. With the adaptive-grained sub-GPU design, we can better utilize those idle resources that can offset its communication cost brought by the memory isolation. Therefore, we can still achieve evident positive performance gains. per-GMI kernel launch is about 4.5% on average under different numbers of GMIs per GPU (1, 2, and 4), considering both the GPU-based simulation engine and neural network kernels, compared to the direct launch of a single kernel on the entire GPU.

3.2 Specialized Communicator

Scaling DRL via AGP demands handling diverse communication to connect DRL components. However, comprehensive communication support is absent from existing GPU multiplexing techniques [6, 45], communication libraries [28, 40], and schedulers [2, 26]. Such a gap drives the specialized design of GMI-DRL *Communicator*, which tackles two major types of DRL communication: the *inter-copy* communication in GMI-centric data-parallel scaling and the *inter-component* communication in GMI-centric model-parallel scaling.

3.2.1 Collective Primitive Composition

We introduce a novel *GMI-based collective primitive composition* strategy with two major primitives: *1) Inter-process primitive (**IP**).* This primitive treats inter-GMI communication as a pure inter-process synchronization. It firstly moves the policy model gradients from GMI to CPUs and then applies gradient reduction on the CPUs across different processes. *2) Ring primitive (**RP**).* This primitive works for GMI layouts with cross-GPU reduction demands. It counts on the existing inter-GPU communication infrastructure, like NCCL on NVLinks, to synchronize GMIs on different GPUs. Note that NCCL prohibits synchronization among GMIs on the same GPU. With the above primitives, we can generate diverse communication compositions given a GMI layout. There are three major types:

IP-only composition only leverages IPs for connecting GMIs. It can handle arbitrary GMI layouts regardless of their underlying placement on GPUs. As exemplified in Figure 7a, all GMIs are synchronized with each other through IPs in only one step. The theoretical cost of the IP-only solution is $\frac{2(m \cdot n-1) \cdot S}{m \cdot n \cdot B_1}$, based on the standard allreduce operation [28], where B_1 is the bandwidth over PCIe, S is the policy model size, n and m are the number of GPUs and GMIs per GPU, respectively. However, our study shows that this composition can only show advantages in settings with fewer GMIs and small policy models due to the limited throughput of using IPs. In scale-up settings, IP-only composition would 1) incur an excessive amount of data traffic between CPUs and GPUs; 2) idle the high-performance GPU interconnects (e.g., NVLinks) that are available on most modern GPU-based platforms; 3) be throttled by slow CPU-based reduction.

RP-only composition only counts on the RPs for connecting different GMIs. It divides GMIs into separate communication groups that satisfy the NCCL constraints Figure 7b shows that GMIs on different GPUs form multiple non-intersect rings for RP communication (❶). We then use one additional RP to synchronize all partial results from the prior stage of reduction (❷). The theoretical cost of the RP-only solution is $2 * \frac{2(n-1) \cdot S}{n \cdot B_2}$, where B_2 is the bandwidth of NCCL on NVLink and synchronization on different rings of ❶ are parallelized. While this composition can enjoy high-speed inter-GPU communication, it is limited to settings where (#GMIs per GPU) $\leq$ (#GPUs) given the constraint of NCCL (connected GMIs must be on different GPUs). With more GMIs per GPU, there will be more NCCL rings and the communication contentions will also increase, potentially decreasing performance.

Hybrid IP + RP composition: This type of composition mixes IPs and RPs. It offers flexibility to fine-grained primitive combinations. One example of such composition is illustrated in Figure 7c, where we have k GMIs per GPU use RPs and $(m-k)$ GMIs per GPU use IPs for reduction. This composition example consists of two major steps: ❶ is gradient synchronization among GMIs on different GPUs through RPs. ❷ is gradient synchronization among GMIs on the same GPUs (and different GPUs, if possible) through IPs.

The theoretical cost of hybrid IP + RP solution is $\frac{2(n-1) \cdot S}{n \cdot B_2} + \frac{2(n \cdot (m-k)+k-1) \cdot S}{(n \cdot (m-k)+k) \cdot B_1}$. The key to utilizing this hybrid composition is to properly adjust the ratio of IPs and RPs and apply

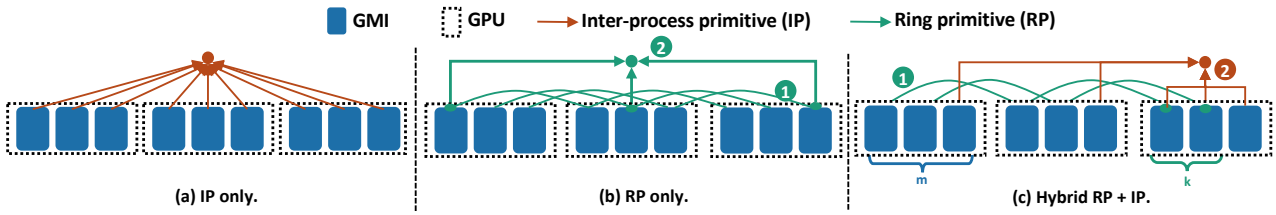


Figure 7: Exemplification of GMI Communication Primitive Composition. Note that the final broadcasting step is omitted for simplicity. n and m are the number of GPUs and GMIs per GPU, respectively. k is a configurable number of GMIs per GPU that uses RPs for communication. B_1 and B_2 are bandwidth by using IPs an RPs, respectively.

them in specific schedules. We next discuss the workflow of generating the effective composition given platform settings.

Overall Composition Workflow: Given an input GMI layout and DRL benchmark, we will generate a composition plan with optimized efficiency. We first analyze its characteristics (e.g., #GMIs per GPU and the GPU interconnects) and performs one-time profiling on basic composition. Based on the analyzed inputs, we will generate all candidate compositions. We then project performance based on profiling results to derive the latency cost of possible compositions. The costs of two basic compositions are IP-only ($Cost_{ip}$) and RP-only ($Cost_{rp}$) compositions measured on n GPUs with 1 GMI per GPU. This will determine the "unit" communication cost given the DRL benchmark and hardware platform. We measure $Cost_{ip}$ (i.e., $\frac{2(n-1)\cdot S}{n\cdot B_1}$) and $Cost_{rp}$ (i.e., $\frac{2(n-1)\cdot S}{n\cdot B_2}$) in raw latency (ms) by averaging 100 runs, where n is the number of GPUs, S is the policy model size, and B_1 and B_2 are the bandwidth of using IPs and RPs, respectively.

We then estimate the cost of (a) and (b) in n GPUs with m GMIs per GPU by scaling the $Cost_{ip}$ and $Cost_{rp}$ based on the ratio of measured and the target problem size. For example, in (a), we have $(n \cdot m)$ instance in the targeted problem size and n in the measured problem size, we then estimate $Cost(a) = \frac{2(n\cdot m-1)\cdot S/(n\cdot m\cdot B_1)}{2(n-1)\cdot S/(n\cdot B_1)} \cdot Cost_{ip} = \frac{(n\cdot m-1)}{m\cdot(n-1)} \cdot Cost_{ip}$. Similarly, we can get the $Cost(b) = 2 * Cost_{rp}$. Note that different RPs can be parallelized if there is no output dependency. For (c), given a parameter k (where $k \leq m$), we can derive $Cost^{(k)}(c) = (\frac{k}{n} + m - k) \cdot Cost_{ip} + Cost_{rp}$, Finally, we select the composition with the minimum projected latency. We provide an experimental demonstration in §4.3.

3.2.2 Channel-based Experience Sharing

The *model*-parallel scaling of DRL counts on connecting different DRL components effectively. Data movement of model-parallel DRL follows a point-to-point paradigm and is mixed with heterogeneous types of data under different granularities. For instance, the environment states of simulators are coarse-grained vectors, while the action/reward of agents is fine-grained scalars. Such heterogeneity and diverse granularity create a challenge to utilize existing GPU interconnections, which prefer fixed-sized coarse-grained data packages.

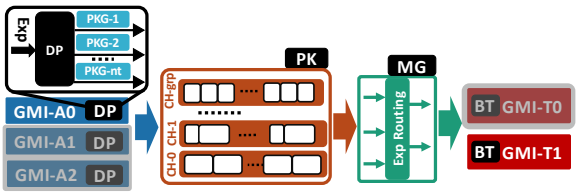


Figure 8: Exemplification of Channel (CH)-based Experience Sharing (A0 to T1). Note that "GMI-A*" is the GMI for DRL agents while "GMI-T*" is the GMI for DRL trainers; "PKG*" are packages decomposed from experiences.

One direct solution is to communicate with heterogeneous data packages. However, it would suffer from high startup costs and underutilized bandwidth for fine-grained transfers. Another direction is to tailor the communication strategies for different data so that we can manage the granularity and heterogeneity of data to maximize communication efficiency. Inspired by this insight and prior efforts on computer networks [17], we develop a *channel-based experience sharing* strategy for model-parallel DRL scaling. Each channel defines a customized way of communication that will manage inter-GMI communication for one type of data or a group of types of data that share similarities (e.g. shapes).

We build such a design with four components (Figure 8). Specifically, Experience *Dispenser* (***DP***) is to decompose the experience data into packages. Experience *Packer* (***PK***) is to group packages into channels and batch package groups in each channel for data movement among GMIs. Experience *Migrator* (***MG***) is to manage the routing of experience from agents to trainers. Experience *Batcher* (***BT***) is to conduct data preparation and model training.

Our initial empirical study shows that the overall performance mainly relies on whether PK can maximize inter-GMI data movement. With multi-channel design in PK, there are two major aspects that are critical to its performance: *1) Which types of packages should be grouped in the same channel?* and *2) How many such groups (i.e., batch size) should be transferred each time?* To address these questions, we define a proxy metric TOP_{mov} (Equation 5) for inter-GMI communication throughput given a package group ($\sum_{j=1}^{g_i} P_j^{(i)}$) and the batch size (*batch*) of package groups. TOP_{mov} will reward the higher bandwidth (due to package grouping) while

penalizing the extra-large packages (due to package concatenation overhead) and too many independent channels (due to inter-channel contention)

$$TOP_{mov} = \max\left\{\min_{i\in[1,grp]}\{\frac{B_i}{Cost_{cat}\cdot Cost_{ch}}\}\right\} \quad (5)$$

where B_i is the bandwidth given the current grouping and batching; g_i is the number of packages in a package group (G_i) and it can vary among groups; grp is the total number of package groups; $P_j^{(i)}$ is the package size (in Bytes) of the j-th package in G_i. Our empirical study shows that $B_i \propto (batch\cdot\sum_{j=1}^{g} P_j)$, $Cost_{cat} \propto (batch\cdot g)$, and $Cost_{ch} \propto grp$. Now, TOP_{mov} can be simplified as

$$TOP_{mov} = \max\left\{\min_{i\in[1,grp]}\{\frac{\sum_{j=1}^{g_i} P_j^{(i)}}{g_i\cdot grp}\}\right\} \quad (6)$$

$$s.t.(\sum_{i=1}^{grp} g_i = nt) \wedge (grp \in [1,nt]) \quad (7)$$

$$\prod_{i,j\in[1,nt]\wedge k\in[1,grp]}^{Constraints} (D_i \in G_k) \wedge (D_j \notin G_k) \quad (8)$$

where nt is the total types of original packages. Equation 8 describes the constraints that package D_i and D_j cannot be in the group/channel G_k due to their unaligned shape/dimension that complicates the concatenation. The maximization of TOP_{mov} will balance between the sizes of package groups and the number of package groups. Given $nt \leq 10$ and low evaluation cost per entry, we address this problem with enumerative combinatorics [11] with deduplication to find possible grouping combinations and backtrack search [47] with pruning to identify possible package concatenation in each group. We finally return the setting with maximum TOP_{mov}.

3.3 GMI-centric Programming Support

GMI-DRL follows a process-based design for performance guarantees and compatibility with GPU spatial multiplexing techniques. Processes offer better isolation with configurations (e.g., environmental variables) and memory address spaces. Listing 1 shows GMI-DRL example for synchronized DRL training. There are three major components: 1) *GMI Context* (Line 3) configures the basic GMI execution settings. For DRL training, MPS is used for communication efficiency. While for DRL serving, MIG is used for a better performance guarantee with minimized interference. 2) *GMI Runtime* (Line 5 to 28) manages the GMI-based DRL runtime. `init` function will set up the GMI based on its `GMI_id` and register a GMI to a global GMI manager. It also attaches GMIs to specific GPUs based on the GPU ids. The GMI will be assigned to a GMI group for collective communication if necessary. Users will implement the `GMI_run` for training.

Our GMI-based design, while demonstrated on A100, can be easily generalized toward other GPUs. This covers a wide spectrum of datacenter GPUs including 1) the older generations including Kepler (e.g., Tesla K40), Pascal (e.g., P100), and Volta (e.g., V100); 2) the newer generation of GPUs including Hopper (e.g., H100/H200), and Blackwell (B200). Our design can also work with desktop GPUs (e.g., RTX20/30/40 and Quadro series) seamlessly.

Listing 1: Illustration of GMI-based Programming.

```
1  import GMI_DRL
2  # import other packages, e.g., PyTorch, Tensorflow ...
3  GMI_DRL.config = {num_GPUs=3, GPU_arch="sm80",
                     NVlink=True, backend="MPS"}
4  # Define GMI tasks.
5  class GMI_base(object):
6      def __init__(self, role, GMI_id, gpu_id):
7          self.mgr = GMI_DRL.GMI_manager.add_GMI(GMI_id)
8          self.group = self.mgr.get_group(GMI_id, gpu_id)
9          self.env = gym('Ant')
10         self.actor_model = nn.Modules(...)
11         self.optimizer = Adam(self.actor_model)
12     def GMI_run(self, n_epsiode, n_step):
13         for j in range(n_epsiode):
14             samples_li, reward_li = list(), list()
15             for i in range(n_step):
16                 state, reward = self.env(state, action)
17                 samples_li.append(state, action)
18                 reward_li.append(reward)
19             self.GMI_train(samples_li, reward_li)
20     def GMI_train(self, samples_li, reward_li):
21         act_pred = self.model(samples_li)
22         act_ref = comp_act_targets(samples_li, reward_li)
23         loss = nn.CrossEntropyLoss(act_pred, act_ref)
24         loss.backward()
25         self.GMI_collective()
26         optimizer.step()
27     def GMI_collective(self):
28         self.mgr.allreduce(self.model, self.group)
```

4 Evaluation

Benchmarks: The DRL benchmark consists of the *simulation environment* and the *policy NN model*. We cover different types of environments, including locomotion simulation [16, 24, 46] (**L**), franka cube stacking [19] (**F**), and robotics hand control [34] (**R**). The details of our evaluated DRL benchmarks are listed in Table 4. "#Dim." is the dimension of environment state vectors. The policy NN models are expressed in "*in_dim:hidden_dim:...:out_dim*".

While there are many DRL algorithms, they can be broadly categorized into *on-* and *off-policy* DRL algorithms [52]. In this evaluation, we build our baseline based on the state-of-the-art MSRL [52] framework and Isaac Gym [24] simulator. Since the MSRL uses OpenAI Gym [3] by default for simulation workload, which is significantly (100×) lower in simulation throughput compared with Isaac Gym. We select the most representative PPO [39] for synchronized (Sync.) DRL training (*Isaac Gym + MSRL PPO as IG (PPO)*), which is the official implementation from Isaac Gym [24]. We also implement the most representative off-policy A3C [1] in Isaac

Table 4: DRL Benchmarks and Policy NN Models.

Benchmark	Abbr.	Type	#Dim.	Policy NN Model
Ant [46]	AT	L	60	60:256:128:64:8
Anymal [16]	AY	L	48	48:256:128:64:12
BallBalance [24]	BB	L	24	24:256:128:64:3
FrankaCabinet [19]	FC	F	23	23:256:128:64:9
Humanoid [46]	HM	L	108	108:200:400:100:21
ShadowHand [34]	SH	R	211	211:512:512:512:256:20

Gym for an asynchronized (Async.) DRL training study *Isaac Gym + MSRL A3C as IG (A3C)*.

Implementations: GMI-DRL is available[1]. For evaluation, we choose several baselines: *1) IG (PPO) in multi-GPU serving* is to directly scale the Isaac Gym serving to multi-GPU platforms with one serving process per GPU. *2) IG (PPO) + NCCL* is for Sync. DRL training in data parallelism and uses NCCL [28] as the communication backend and this is also the only distributed version officially supported by Issac Gym. *3) IG (PPO) + Horovod* is also for Sync. DRL training in data parallelism, and uses Horovod [40] as the communication backend. *4) IG (A3C)* is for study on Async. DRL training, where policy serving and policy training are running concurrently on different GPUs. *5) Ray* is the framework for DRL training. To compare with Ray [27], we select the officially optimized multi-GPU PPO training examples in RLLib [23] of Ray. For all baselines, we manually adjust the simulation batch size (i.e., *num_env*) of each DRL benchmark to reach its peak throughput by using each GPU exclusively. The configurations (*num_env* and *GMIperGPU*) are generated with Algorithm 1.

Platforms & Tools: We use a DGX-A100 [29] (including 8×NVIDIA A100 GPUs and Dual AMD Rome 7742 CPUs with a total of 128 cores@2.25 GHz). For environment simulation, we leverage NVIDIA Isaac Gym [24]. We use PyTorch (v3.8) [35] to build the policy model, and NCCL (v2.8.4) [28] and Gloo [10] (commit `a01540e`) to build the GMI-based communication layer. To measure the performance, we leverage the `nvidia-smi` tool and PyTorch runtime profiler. The throughput metric of DRL serving and Sync. DRL training is the training samples (simulation steps) per second [24].

4.1 Overall Performance

DRL Serving Figure 9a shows how with the help of our GMI-oriented design, the overall throughput of policy serving can be improved (up to 2.62×, averaged 2.08×) across settings with different numbers of GPUs. Based on our detailed profiling results, our GMI-based design can improve GPU utilization (up to 45.7%, averaged 27.9%) compared with the state-of-the-art Isaac Gym. This also demonstrates GMI-DRL to maximize GPU utilization and performance.

Sync. DRL Training We conducted two experiments. *Compared with IG (PPO) + NCCL:* Figure 9b demonstrates that GMI-DRL with the optimal GMI planning can achieve up to 2.07× (averaged 1.69×) throughput compared to Isaac Gym + MSRL with NCCL under the same number of GPUs. In the more complex DRL setting with a higher number of concurrent simulations and state/observation dimensions, GMI-DRL demonstrates more benefits. This is because of our effective resource planning to maximize the performance gains from available GPU resources, resource isolation to minimize execution interference and communication support to synchronize policy gradients among different training copies. In addition, our profiler shows that GMI-DRL improves GPU utilization by up to 40.8% (31.8% on average). Such an improvement can help users save evident cloud service costs under the same training throughput.

Compared with IG (PPO) + Horovod: Figure 9c shows that GMI-DRL outperforms Isaac Gym + MSRL with Horovod backend with up to 2.34× (averaged 1.72×) training performance throughput. We also notice that there are performance differences between NCCL and Horovod baselines. It can be attributed to the way of updating the policy model parameters. The NCCL-based designs rely on separate gradient reduction and PyTorch optimizer, while the Horovod-based designs manage their own distributed optimizer including both gradient reduction and the distributed model update. Besides, we notice the superlinear speedup number achieved by GMI-DRL in this experiment. This is mainly due to the combined effects of scaling up with multiple GMIs per GPU and scaling out with multiple GPUs. This implies that only focusing on optimizing communication is insufficient to exploit the full potential of multi-GPU platforms.

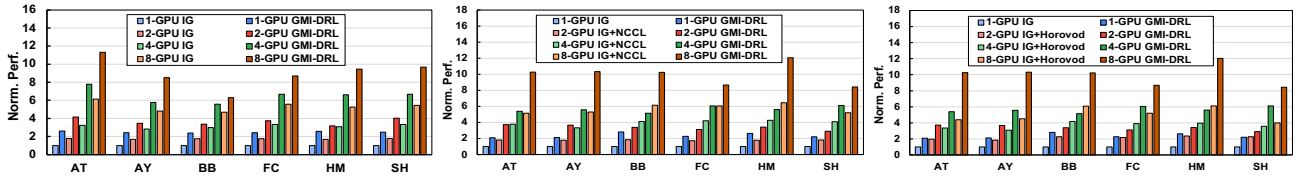
Async. DRL Training We compare GMI-DRL with IG (A3C) in Async. DRL training. We measure *predictions per second* (PPS) and the *training samples throughput* (TTOP) as [25]. Figure 10 shows that with the help of our GMI-based design and the throughput optimized experience transferring technique, GMI-DRL can achieve averaged 1.88× PPS and 1.65× TTOP over non-GMI baseline on 2 and 4 GPUs. This result shows the importance of collaboratively improving the performance of experience collection and experience sharing for Async. DRL training. Our additional study reveals that channel-based experience sharing (§3.2.2) with the searched optimal package grouping plan would contribute to 24% of overall performance improvement.

4.2 Comparison with Ray

We compare GMI-DRL with Ray RLLib (leveraging OpenAI Gym) on its expert-tuned multi-GPU PPO implementation [36]. Specifically, for our GMI-DRL design, we replace our Isaac Gym with OpenAI Gym [3] on several benchmarks (including Ant and Humanoid, which are from MuJoCo [46]).

Our result shows that GMI-DRL outperforms Ray evidently in training throughput (up to 62%). Despite that the Ray actor can leverage partial GPU (in percentage) for policy model

[1] https://gitlab.com/YK-Wang96/gmi-drl-ae.git



(a) (b) (c)

Figure 9: End-to-end computation throughput comparison for (a) *DRL serving*, (b) *Sync. DRL training with NCCL*, and (c) *Sync. DRL training with Horovod*. Note that computation throughput is normalized w.r.t. **Isaac Gym+MSRL** (*IG*) on a single GPU.

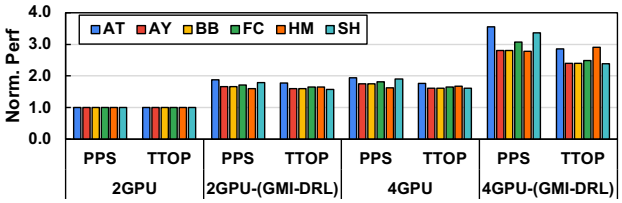


Figure 10: Async. DRL training throughput performance.

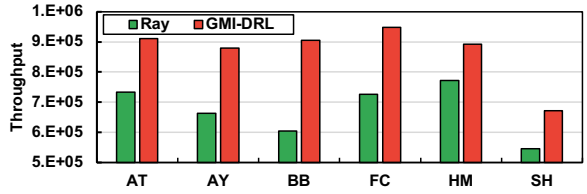


Figure 11: Training comparison with Ray on 8×A100.

inference (action prediction) to collect experience, for scaling policy training, Ray still has to follow the conventional one-to-one mapping between trainers and GPUs and leverage NCCL/Horovod for communication [37]. Therefore, Ray can only utilize GPUs in a fixed yet coarse-grained manner during the training, missing the opportunities to harness spare GPU resources. When we retarget our existing design to other simulators, we need to consider 1) placement of simulators on CPUs or GPUs depending on their property and computation efficiency, and 2) layout/topology of simulators based on their connected channels (e.g., NVLink and PCIe). Our design can flexibly accommodate such change by tailoring design parameters in our existing communicators and coordinators.

4.3 Additional Study

Multi-Node (Machine) Scaling As shown in Table 5, we compute the normalized throughput for *multi-node* w.r.t. *single-node* Sync. DRL training. We consider the settings of 1, 2, 4, and 8 nodes with (a) 1 GPU per node and (b) 2 GPUs per node. The throughput improvement from GMI-DRL remains consistent in the multi-node settings, which reaches an average of 83% of the ideal (linear) scaling performance. This demonstrates that our collective composition and its optimization technique can effectively adapt towards multi-node policy gradient synchronization by considering

Table 5: Multi-node scaling with (a) 1 GPU per Node and (b) 2 GPUs per Node on Sync. DRL training.

1 GPU/Node	**1-node**	**2-node**	**4-node**	**8-node**
BB	1.00×	1.84×	3.28×	7.16×
HM	1.00×	1.71×	2.87×	6.67×
2 GPUs/Node	**1-node**	**2-node**	**4-node**	**8-node**
BB	1.00×	1.83×	3.18×	7.04×
HM	1.00×	1.61×	2.72×	6.44×

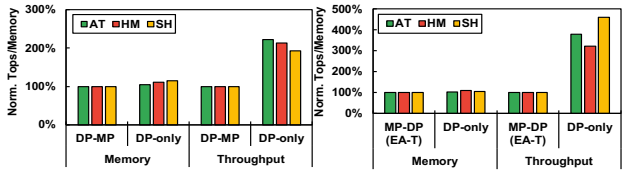
Table 6: Throughput on Collective Composition.

Benchmark	2G3T ($m=3$)			4G3T ($m=3$)		
	Baseline	GMI-DRL	k	Baseline	GMI-DRL	k
AT	319,699	**320,925**	2	529,081	**533,018**	2
HM	415,280	**425,641**	1	621,871	**636,339**	1
SH	283,918	**284,492**	3	485,873	**486,209**	0

the intra- and inter-node GMI layout hierarchy and combining different inter-GMI communication options.

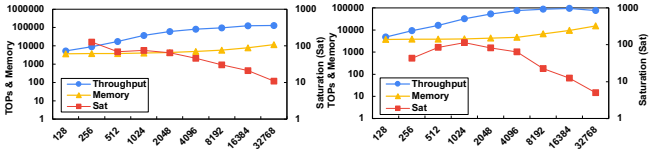
Collective Composition We also demonstrate the effectiveness of GMI-tailored Collective Composition. Our baseline is IP-only. We choose three settings, including (i) 2 GPUs with 3 Trainers per GPU (2G3T); (ii) 4 GPUs with 4 Trainers per GPU (4G4T). Table 6 indicates that our collective composition can bring evident throughput benefits for DRL training on different policy models. Our Algorithm 1 identifies the right choice for k to maximize communication efficiency. This indicates that our communication optimization can pinpoint the composition plan that will maximize the efficiency of different primitives on various types of interconnects.

Task-aware GMI Mapping We empirically demonstrate our mapping choice comparison for DRL serving (Table 2) and Sync. DRL training (Table 3). Results in Figure 12 show that 1) for DRL Serving, our DP-only design achieves significant throughput improvements (average 110%) with minor memory resource penalty (average 9.5%); 2) for Sync. DRL training, co-locating experience collection and trainer achieves significant throughput improvements (287.5% on average) with a minor memory resource penalty (averaged 6.5%) compared with DP-MP (EA-T). Note that for DRL benchmarks (other than Isaac Gym), our theoretical and experimental analysis methodology can still be applied but the mapping decisions could vary depending on the internal simulator implementation and policy model choices, which bring different performance-resource tradeoffs.



(a) (b)

Figure 12: Throughput performance and Memory consumption on (a) DRL serving and (b) Sync. DRL training.



(a) (b)

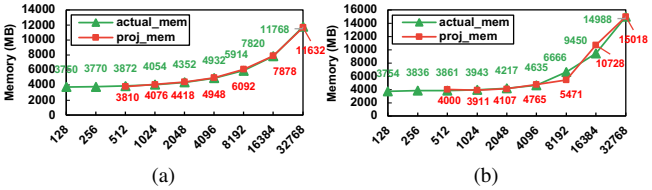
Figure 13: Saturation (*Sat*) Metric on (a) AT and (b) HM.

Saturation Metric & Memory Projection We analyze the Saturation (*Sat*) (Algorithm 1) metric in a single Sync. DRL training instance. Figure 13 shows that with the increasing *#environment* per training instance, the overall throughput will first increase, and then it will increase slowly or even show some decrease (e.g., HM at 32,768 environments). This is mainly due to the complex physics computation in the DRL simulator that cannot scale well to generate large-size data batches. This also justifies using such DRL-specific throughput increase-saturation property to determine an early stop of parameter search. We also evaluate our memory projection model (Equation 4). Figure 14 shows that our memory projection model can effectively predict memory cost given the *#environment* with minor deviation, indicating its usefulness in joint configuration optimization.

5 Related Work

Individual DRL Libraries: Seed RL [9] introduces modern scalable RL agent. Acme [14] provides modularized libraries for building and scaling DRL applications. WarpDrive [21] is a lightweight and open-source library to accelerate single and multi-agent DRL simulation performance on GPUs. Isaac Gym [24] from NVIDIA Omniverse [32] represents the state-of-the-art GPU-based DRL acceleration design, which significantly outperforms its CPU-based counterparts [3] and GPU-based solutions [21] in terms of its simulation throughput. GMI-DRL focuses on adaptive-grained DRL tasks mapping and scheduling are orthogonal and complementary to these existing libraries for specific DRL tasks like simulation.

Holistic DRL Systems: Ray [27] is a distributed computing framework for emerging DL applications including DRL (RLLib). PodRecer [13] introduces a unified scaling architecture for DRL on TPU-based systems. TLeague [43] targets



(a) (b)

Figure 14: Memory Projection on (a) AT and (b) HM.

DRL scaling on hybrid CPU and GPU clusters for Multi-Agent Reinforcement Learning. MSRL [52] is the state-of-the-art DRL computing framework that features dataflow fragments for individual DRL functions on CPUs/GPUs. However, MSRL still relies on off-the-shelf communication infrastructures (e.g., NCCL) for scaling the DRL computation and it can only support very limited options of computation granularity in its DRL fragments on GPUs. For instance, the MSRL fragment plan can only allow one DRL simulator/learner on one GPU. Additionally, MSRL relies on hard-coded manual configuration and requires expert users to adapt to different DRL models/inputs. GMI-DRL opens up an all-new dimension by enabling adaptive-granular parallelism at the sub-GPU level that will enlarge the design space of those existing systems.

6 Discussion

Our work is the first one investigating the design choices of DRL training and serving for a multi-GPU system. We would like to highlight some key design insights and technical contributions.

Observations. Several observations are drawn through our initial investigation: (a) MIG (Multi-instance GPU) allows strict resource isolation through the full hierarchy (memory, cache, interconnect, SMs) to avoid potential resource competition, but has explicit limitations on communication, thus is more suitable for DRL serving; (b) MPS (Multi-Processing Service) does not enforce resource isolation among agents, but can be more flexible at communication, thus is a better choice for training, given the necessity to collect data from distributive GMI instances; (c) Overall, for single-node scaling-up, until the communication hurdle can be well-addressed by the vendor, MPS is more preferred for the implementation.

Architecture. Despite the use of spatial multiplexing in the prior CPU work, the GPU-based spatial multiplexing is relatively new (e.g., MIG). GPU spatial multiplexing is designed to support diverse and independent GPU applications while guaranteeing their quality of service. There is a trend to keep and consistently improve this feature in future hardware (e.g., H100/H200), and other GPU vendors also follow a similar trend (e.g., AMD MI200 GPU requires at least two MPI processes to fully utilize the GPU resources).

Software/Application. GPUs have now become a more generic accelerator, and many real-world applications (e.g.,

DNN inference) intrinsically cannot fully utilize GPUs. To maximize its utilization, existing work leverages MIG for scaling up simple applications, which just matches the initial design objective of MIG. However, we target complex DRL training with GPU underutilization and interaction/communication among different components. These challenges are outside the scope of MPS/MIG and demand careful resource/-workload planning and efficient communication support.

Performance. Our design evidently outperforms the state-of-the-art Isaac Gym and would facilitate the quick/wide deployment of DRL at scale. The performance regarding different DRL design choices, as well as these observations, are not reported by any existing works.

7 Conclusion

This paper introduces GMI-DRL, the first system to scale multi-GPU DRL with adaptive-grained parallelism. GMI-DRL includes a novel design of resource-efficient GPU multiplexing instances (GMIs), highly efficient communication support across GMIs, and an adaptive GMI management strategy to meet various demands of DRL training. Experiments demonstrate GMI-DRL advantages across various settings over the state-of-the-art DRL frameworks.

8 Acknowledgment

We would like to appreciate the great help and support from the USENIX ATC'25 shepherd and anonymous reviewers. This work was supported in part by NSF 2124039. This material is also based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, ComPort: Rigorous Testing Methods to Safeguard Software Porting, under Award Number 78284. The Pacific Northwest National Laboratory is operated by Battelle for the U.S. Department of Energy under Contract DE-AC05-76RL01830.

References

[1] Mohammad Babaeizadeh, Iuri Frosio, Stephen Tyree, Jason Clemons, and Jan Kautz. Ga3c: Gpu-based a3c for deep reinforcement learning. *International Conference on Learning Representations (ICLR)*, 2016.

[2] Zhihao Bai, Zhen Zhang, Yibo Zhu, and Xin Jin. Pipeswitch: Fast pipelined context switching for deep learning applications. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*.

[3] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016.

[4] Dong Chen, Kaian Chen, Zhaojian Li, Tianshu Chu, Rui Yao, Feng Qiu, and Kaixiang Lin. Powernet: Multi-agent deep reinforcement learning for scalable powergrid control. *IEEE Transactions on Power Systems*, 2021.

[5] Jianyu Chen, Bodi Yuan, and Masayoshi Tomizuka. Model-free deep reinforcement learning for urban autonomous driving. In *2019 IEEE intelligent transportation systems conference (ITSC)*, pages 2765–2771. IEEE, 2019.

[6] Seungbeom Choi, Sunho Lee, Yeonjae Kim, Jongse Park, Youngjin Kwon, and Jaehyuk Huh. Multi-model machine learning inference serving with gpu spatial partitioning. *arXiv preprint arXiv:2109.01611*, 2021.

[7] Steven Dalton, Iuri Frosio, and Michael Garland. Accelerating reinforcement learning through gpu atari emulation. *34th Conference on Neural Information Processing Systems (NeurIPS), 2020*.

[8] Ruisheng Diao, Zhiwei Wang, Di Shi, Qianyun Chang, Jiajun Duan, and Xiaohu Zhang. Autonomous voltage control for grid operation using deep reinforcement learning. In *2019 IEEE Power & Energy Society General Meeting (PESGM)*, pages 1–5. IEEE, 2019.

[9] Lasse Espeholt, Raphaël Marinier, Piotr Stanczyk, Ke Wang, and Marcin Michalski. Seed rl: Scalable and efficient deep-rl with accelerated central inference. *arXiv:1910.06591*, 2019.

[10] Facebook. Gloo collective communications library. `github.com/facebookincubator/gloo`.

[11] Ian P Goulden and David M Jackson. *Combinatorial enumeration*. Courier Corporation, 2004.

[12] Shixiang Gu, Ethan Holly, Timothy Lillicrap, and Sergey Levine. Deep reinforcement learning for robotic manipulation with asynchronous off-policy updates. In *2017 IEEE international conference on robotics and automation (ICRA)*, pages 3389–3396. IEEE, 2017.

[13] Matteo Hessel, Manuel Kroiss, Aidan Clark, Iurii Kemaev, John Quan, Thomas Keck, Fabio Viola, and Hado van Hasselt. Podracer architectures for scalable reinforcement learning. *arXiv preprint arXiv:2104.06272*, 2021.

[14] Matt Hoffman, Bobak Shahriari, John Aslanides, Gabriel Barth-Maron, Feryal Behbahani, Tamara Norman, Abbas Abdolmaleki, Albin Cassirer, Fan Yang, Kate Baumli, Sarah Henderson, Alexander Novikov, Sergio Gómez Colmenarejo, Serkan Cabi, Çaglar Gülçehre,

Tom Le Paine, Andrew Cowie, Ziyu Wang, Bilal Piot, and Nando de Freitas. Acme: A research framework for distributed reinforcement learning. *arXiv:2006.00979*, 2020.

[15] Renke Huang, Yujiao Chen, Tianzhixi Yin, Xinya Li, Ang Li, Jie Tan, Wenhao Yu, Yuan Liu, and Qiuhua Huang. Accelerated deep reinforcement learning based load shedding for emergency voltage control. *arXiv:2006.12667*, 2020.

[16] Marco Hutter, Christian Gehring, Dominic Jud, Andreas Lauber, C Dario Bellicoso, Vassilios Tsounis, Jemin Hwangbo, Karen Bodie, Peter Fankhauser, Michael Bloesch, et al. Anymal-a highly mobile and dynamic quadrupedal robot. In *2016 IEEE/RSJ international conference on intelligent robots and systems (IROS)*, pages 38–44. IEEE, 2016.

[17] Ozlem Durmaz Incel. A survey on multi-channel communication in wireless sensor networks. *Computer Networks*, 55(13):3081–3099, 2011.

[18] Dmitry Kalashnikov, Alex Irpan, Peter Pastor, Julian Ibarz, Alexander Herzog, Eric Jang, Deirdre Quillen, Ethan Holly, Mrinal Kalakrishnan, Vincent Vanhoucke, and Sergey Levine. Scalable deep reinforcement learning for vision-based robotic manipulation. In *Conference on Robot Learning*, pages 651–673. PMLR, 2018.

[19] Oussama Khatib. A unified approach for motion and force control of robot manipulators: The operational space formulation. *IEEE Journal on Robotics and Automation*, 3(1):43–53, 1987.

[20] B Ravi Kiran, Ibrahim Sobh, Victor Talpaert, Patrick Mannion, Ahmad A Al Sallab, Senthil Yogamani, and Patrick Pérez. Deep reinforcement learning for autonomous driving: A survey. *IEEE Transactions on Intelligent Transportation Systems*, 2021.

[21] Tian Lan, Sunil Srinivasa, Huan Wang, and Stephan Zheng. Warpdrive: Extremely fast end-to-end deep multi-agent reinforcement learning on a gpu. *arXiv preprint arXiv:2108.13976*, 2021.

[22] Baolin Li, Tirthak Patel, Siddharth Samsi, Vijay Gadepally, and Devesh Tiwari. Miso: exploiting multi-instance gpu capability on multi-tenant gpu clusters. In *Proceedings of the 13th Symposium on Cloud Computing*, pages 173–189, 2022.

[23] Eric Liang, Richard Liaw, Philipp Moritz, Robert Nishihara, Roy Fox, Ken Goldberg, Joseph E Gonzalez, Michael I Jordan, and Ion Stoica. Rllib: Abstractions for distributed reinforcement learning. arxiv e-prints, page. *arXiv preprint arXiv:1712.09381*, 2017.

[24] Viktor Makoviychuk, Lukasz Wawrzyniak, Yunrong Guo, Michelle Lu, Kier Storey, Miles Macklin, David Hoeller, Nikita Rudin, Arthur Allshire, Ankur Handa, and Gavriel State. Isaac gym: High performance GPU based physics simulation for robot learning. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 2)*, 2021.

[25] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning (ICML)*, 2016.

[26] Jayashree Mohan, Amar Phanishayee, Janardhan Kulkarni, and Vijay Chidambaram. Looking beyond {GPUs} for {DNN} scheduling on {Multi-Tenant} clusters. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 579–596, 2022.

[27] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. Ray: A distributed framework for emerging AI applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 561–577, Carlsbad, CA, October 2018. USENIX Association.

[28] Nvidia. Nvidia collective communication library (nccl). developer.nvidia.com/nccl.

[29] Nvidia. Nvidia dgx a100. www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/nvidia-dgx-a100-datasheet.pdf.

[30] Nvidia. Nvidia multi-instance gpu. www.nvidia.com/en-us/technologies/multi-instance-gpu/.

[31] NVIDIA. Nvidia multi-process service. docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf.

[32] NVIDIA. Nvidia omniverse. https://www.nvidia.com/en-us/omniverse/.

[33] OpenAI. Openai chatgpt. https://openai.com/blog/chatgpt/.

[34] OpenAI, Marcin Andrychowicz, Bowen Baker, Maciek Chociej, Rafal Józefowicz, Bob McGrew, Jakub Pachocki, Arthur Petron, Matthias Plappert, Glenn Powell, Alex Ray, Jonas Schneider, Szymon Sidor, Josh Tobin, Peter Welinder, Lilian Weng, and Wojciech Zaremba. Learning dexterous in-hand manipulation. *The International Journal of Robotics Research*, 39(1):3–20, 2020.

[35] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017.

[36] Ray. tuned_examples for ppo. https://github.com/ray-project/ray/blob/master/rllib/tuned_examples/ppo/humanoid-ppo.yaml.

[37] Ray Project Contributors. *Ray Collective Communication Library*. Ray Project, 2025. Accessed: 2025-05-06.

[38] Ahmad EL Sallab, Mohammed Abdou, Etienne Perot, and Senthil Yogamani. Deep reinforcement learning framework for autonomous driving. *Electronic Imaging*, 2017(19):70–76, 2017.

[39] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv:1707.06347*, 2017.

[40] Alexander Sergeev and Mike Del Balso. Horovod: fast and easy distributed deep learning in tensorflow. *arXiv preprint arXiv:1802.05799*, 2018.

[41] SPK Spielberg, RB Gopaluni, and PD Loewen. Deep reinforcement learning approaches for process control. In *2017 6th international symposium on advanced control of industrial processes (AdCONIP)*. IEEE.

[42] Erich Strohmaier, Jack Dongarra, Horst Simon, and Martin Meuer. Top500 the list. https://www.top500.org/.

[43] Peng Sun, Jiechao Xiong, Lei Han, Xinghai Sun, Shuxing Li, Jiawei Xu, Meng Fang, and Zhengyou Zhang. Tleague: A framework for competitive self-play based distributed multi-agent reinforcement learning. *arXiv preprint arXiv:2011.12895*, 2020.

[44] Lei Tai, Giuseppe Paolo, and Ming Liu. Virtual-to-real deep reinforcement learning: Continuous control of mobile robots for mapless navigation. In *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 31–36. IEEE, 2017.

[45] Cheng Tan, Zhichao Li, Jian Zhang, Yu Cao, Sikai Qi, Zherui Liu, Yibo Zhu, and Chuanxiong Guo. Serving dnn models with multi-instance gpus: A case of the reconfigurable machine scheduling problem. *arXiv:2109.11067*, 2021.

[46] Emanuel Todorov, Tom Erez, and Yuval Tassa. Mujoco: A physics engine for model-based control. In *2012 IEEE/RSJ international conference on intelligent robots and systems*, pages 5026–5033. IEEE, 2012.

[47] Peter Van Beek. Backtracking search algorithms. In *Foundations of artificial intelligence*, volume 2, pages 85–134. Elsevier, 2006.

[48] Sen Wang, Daoyuan Jia, and Xinshuo Weng. Deep reinforcement learning for autonomous driving. *arXiv:1811.11329*, 2018.

[49] Wikipedia. Atari 2600. https://en.wikipedia.org/wiki/Atari_2600.

[50] Fangyi Zhang, Jürgen Leitner, Michael Milford, Ben Upcroft, and Peter Corke. Towards vision-based deep reinforcement learning for robotic motion control. *arXiv:1511.03791*, 2015.

[51] Zidong Zhang, Dongxia Zhang, and Robert C Qiu. Deep reinforcement learning for power system applications: An overview. *CSEE Journal of Power and Energy Systems*, 6(1):213–225, 2019.

[52] Huanzhou Zhu, Bo Zhao, Gang Chen, Weifeng Chen, Yijie Chen, Liang Shi, Yaodong Yang, Peter Pietzuch, and Lei Chen. {MSRL}: Distributed reinforcement learning with dataflow fragments. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 977–993, 2023.

*m*Tuner: Accelerating Parameter-Efficient Fine-Tuning with Elastic Tensor

Kezhao Huang Siqi Zhu Mingshu Zhai Liyan Zheng Kinman Lei Jiaao He
Yuyang Jin Jidong Zhai
Tsinghua University

Abstract

With the growing importance of personalized large language models (LLMs) and fine-tuning techniques, parameter-efficient fine-tuning (PEFT) has emerged as a mainstream approach, offering reduced computational and storage demands compared to full-parameter fine-tuning. Compared to pre-training, we find memory efficiency more critical during fine-tuning. Although the overall memory capacity of fine-tuning hardware is typically limited, memory becomes more precious since most parameters are frozen and can be cached for performance optimization. To better utilize memory, we propose Elastic Tensor, an abstraction for dynamic tensor management, enabling flexible control over their availability, accumulation, and release in memory. Elastic tensor defines four key operations for static and runtime tensors with tunable ratios: *gather*, *discard*, *execute*, and *checkpoint*. With elastic tensors, a series of optimizations are enabled, such as improving temporal memory utilization, relaxing data dependence, and accumulating runtime tensors in a memory-adaptive way. We implement mTuner, an end-to-end fine-tuning system based on elastic tensors. Compared with state-of-the-art training and fine-tuning systems, mTuner achieves a throughput improvement of up to 51.2% and 24.8% (28.3% and 14.5% on average) on PCIe and NVLink servers respectively, for LLMs from 7B to 70B. *m*Tuner is publicly available at https://github.com/xxcclong/mTuner.

1 Introduction

Large language models (LLMs) [1,34,43,50] are becoming increasingly prevalent across various applications. Meanwhile, fine-tuning techniques for model personalization [9, 47, 52] have emerged as a critical topic in large-scale model training. Fine-tuning is typically conducted with limited data and modest hardware resources [10, 11, 59], while often requiring the rapid generation of multiple fine-tuned models [13, 17, 61]. However, the immense number of parameters in large language models poses significant challenges to performing full-parameter fine-tuning under these constraints [16, 44, 60].

To overcome this limitation, researchers have introduced parameter-efficient fine-tuning (PEFT). Unlike full-parameter fine-tuning, PEFT leverages pre-trained weights by updating only a small subset of parameters while keeping the majority of the model frozen [15, 22, 28, 37, 54]. This approach significantly reduces computational overhead and storage requirements, as only the updated parameters need to be saved, yet it maintains competitive performance. As a result, PEFT has become the predominant fine-tuning strategy [8, 16, 24].

Parallelization is essential for efficient fine-tuning of LLMs across multiple GPUs and servers, as it enables the distribution of computation and data to meet the high memory and computational demands. Various strategies, including data, tensor, and pipeline parallelism, have been proposed to improve distributed training performance [40, 45]. For better performance, previous work tried to search for the optimal parallel strategy [12, 51, 58] and improve communication efficiency [2, 3, 33, 62]. On the other hand, efficient parallel execution relies heavily on available memory, which not only stores weight parameters to reduce communication overhead but also supports larger batch sizes for increased parallelism. As shown in Figure 1a, fine-tuning throughput improves with higher memory utilization, making it critical to maximize memory usage while staying within hardware limits. Therefore, previous works optimize memory efficiency using swapping [3, 55], offloading [39, 41], and activation checkpointing [36, 42, 48]. However, memory efficiency still remains a key challenge, hindering the full potential of distributed fine-tuning.

The first challenge arises from low temporal memory utilization, which stems from the inherent first-in-last-out pattern of runtime tensor such as activations. This pattern creates *peaks and valleys in memory usage*, where memory is heavily occupied during peaks but is underutilized during valleys, resulting in inefficient usage of available memory resources (Figure 1c).

The second challenge comes from the data dependency between computation and communication. During fine-tuning, communication is often required for runtime tensors, such

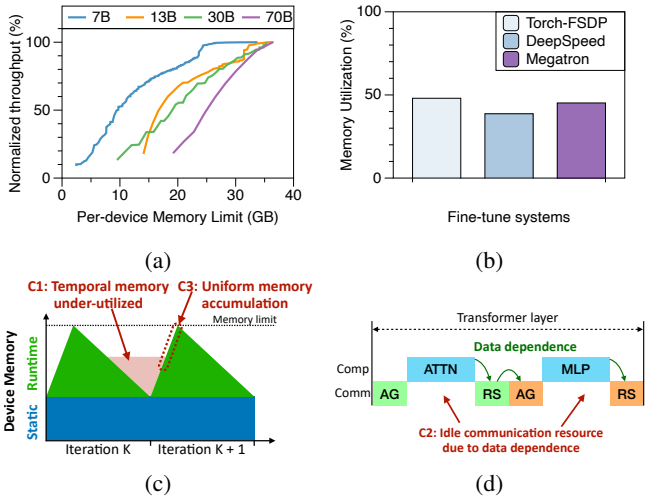


Figure 1: (a) The throughput of fine-tuning models is directly proportional to memory limit; (b) Existing fine-tuning systems have low time-average memory utilization; (c) Low temporal memory utilization and high peak memory caused by uniform memory accumulation; (d) Communication resource waste due to data dependence.

as performing all-gather for activations in tensor parallelism. However, communication can only begin after the relevant data is produced, leading to rigid data dependence. This dependence limits opportunities for overlapping communication with previous computation, causing inefficient utilization of communication resource (Figure 1d).

The third challenge is related to memory accumulation and peak memory consumption. Existing methods accumulate memory without considering the different phases of memory usage. As a result, memory accumulation strategies apply uniformly in both the peak and valley phases, leading to extremely high memory pressure during the peak phases, thus constraining other optimization space related to memory (Figure 1c).

As a result, existing frameworks have low time-average memory utilization as shown in Figure 1b. The underlying root cause of these challenges lies in the static deploying of memory adopted by current parallelization frameworks. Their static scheduling assumes a uniform and large-size pattern of memory, which fails to accommodate the highly dynamic nature of fine-tuning. This dynamism arises from several factors: the runtime tensor being generated and accumulated on-the-fly during computation, the dynamic availability of computational and communication resources, and the variable data dependence that must be resolved at runtime.

To address the above challenges, we propose *elastic tensor*, an abstraction for dynamic tensor management. By defining four core actions and allowing flexible control of memory through tunable ratios, elastic tensor provides a unified method for discovering memory optimizations.

With elastic tensor, a series of optimizations are enabled for fine-tuning. First, we introduce a temporal memory management strategy that leverages frozen parameters in PEFT to reuse idle memory during valley stages, significantly enhancing temporal memory utilization. Second, elastic tensor facilitates the interplay of communication between static and runtime tensors, thereby relaxing data dependencies and improving communication-computation overlap. Third, we propose an adaptive accumulation strategy that dynamically adapts how runtime tensors are stored and accumulated, effectively reducing peak memory pressure and improving overall memory efficiency.

Based on elastic tensor, we develop *m*Tuner, a fine-tuning system with high memory utilization for better PEFT efficiency. *m*Tuner leverages elastic tensor at runtime to maximize memory utilization and reduce communication workload. We evaluate *m*Tuner on transformer-based LLMs with sizes ranging from 7B to 70B parameters on an eight-GPU server. Results show that *m*Tuner improves fine-tuning performance by up to 1.51× (1.28× on average) compared to state-of-the-art training systems, including DeepSpeed [40], Megatron [21], and Flux [2].

In this paper, we have made the following contributions.

- We identify key memory efficiency challenges in LLM fine-tuning and analyze their impact on performance.
- We propose elastic tensor, an abstraction for dynamic tensor management. It supports four core actions with tunable ratios, providing fine-grained control to enable key optimizations in memory efficiency.
- We use elastic tensor to represent and apply three novel optimizations that improve performance by enhancing memory efficiency.
- We evaluate *m*Tuner on various models with parameter sizes ranging from 7B to 70B, achieving improvement of up to 51.2% and 24.8% (28.3% and 14.5% on average) on PCIe and NVLink servers.

2 Background

2.1 LLM Fine-tuning

LLM fine-tuning refers to the process of imparting domain-specific knowledge to a LLM based on a small amount of domain-specific data, building upon a pre-trained model. Fine-tuning holds significant importance in applying LLMs across various industries and domains.

Traditional fine-tuning is similar to pre-training, which updates all parameters. The example is shown in Figure 2(b), all weight parameters are trainable and need to be updated during backward propagation. To make fine-tuning more light-weighted, researchers propose Parameter-Efficient Fine-Tuning (PEFT), which also starts with a pre-trained model,

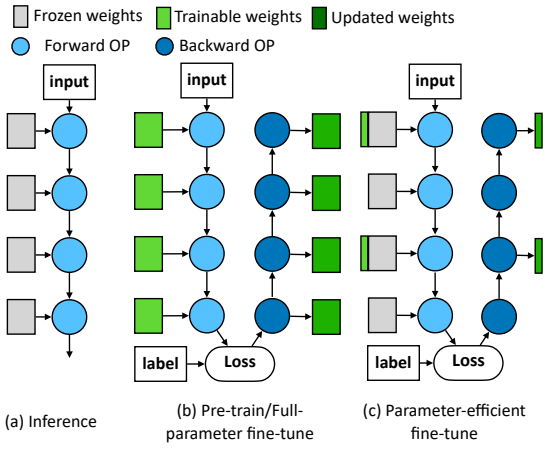


Figure 2: Illustration of (a) inference, (b) pre-training/full parameter fine-tuning, and (c) parameter-efficient fine-tuning (PEFT)

but does not update the whole parameters of the pre-trained model. Instead, it introduces a small number of trainable parameters called adapters into the model structure[1]. As shown in Figure 2(c), during training, the parameters of the pre-trained model (base model) are involved in computations but not updated (frozen); only the parameters of the adapters are updated based on the loss and optimizer state.

Compared with full-parameter fine-tuning, PEFT has the following advantages: 1). It significantly reduces the memory requirements for the optimizer state, needing only the adapter's optimizer state and not that of the base model; 2). As it does not compute gradients for the majority of parameters during backward propagation, it saves computational resources; 3) Since only the parameters of the adapter differ after tuning, saving only the adapter's parameters (along with the original base model's parameters) constitutes the tuned model, facilitating propagation and serving. Consequently, PEFT has become a commonly used method in current fine-tuning practices and is the main focus of this paper.

2.2 Data Communication

In distributed fine-tuning, data communication plays a crucial role in ensuring parallel efficiency across multiple GPUs. The data involved in fine-tuning can be broadly categorized into two types: static tensor and runtime tensor, each with distinct characteristics and communication requirements.

Static tensor primarily consists of model parameters, such as weights and biases, since these tensors persistently reside in memory. Distributed fine-tuning often employs techniques like data parallelism, where static tensor is either split across devices or replicated. For example, in fully-sharded data parallelism (FSDP), each device holds a partition of the parameters, and communication is required when executing a module to get all its weights locally available to perform computation on it.

Runtime tensor refers to intermediate activations, gradients, and other temporary data produced during the forward and backward passes. Unlike static tensor, runtime tensor is highly dynamic, with its communication pattern depending on the execution flow of the model. For instance, in tensor parallelism, runtime tensors such as activations needs to be gathered or scattered across devices during computation. Moreover, runtime tensor communication is constrained by data dependence, meaning communication can only occur after the relevant computation has been completed, which limits opportunities for overlapping communication with computation.

2.3 Memory Challenges in LLM fine-tuning

Fine-tuning involves dynamic memory patterns due to runtime tensors, such as intermediate activations and gradients. These runtime tensors often dominate memory consumption and directly affect the achievable batch size, sequence length, and ultimately the training throughput. On the other hand, memory can be used for caching static tensor to reduce communication overhead. To improve memory efficiency during fine-tuning, it is essential to address several challenges arising from the complex interplay of computation, communication, and memory accumulation. Specifically, we highlight three major challenges: inefficient utilization of idle memory during temporal valleys, waste of communication resources caused by data dependence, and high peak memory pressure due to inflexible accumulation strategies.

Ignoring valley memory: Low temporal memory utilization . One major challenge in LLM fine-tuning is the inefficient utilization of memory due to the fluctuating nature of temporal memory usage. During fine-tuning, runtime tensor follows a first-in-last-out pattern, which creates peaks and valleys in memory consumption. At peak moments, memory is highly occupied, while during valleys, a significant portion of memory remains idle. This inefficiency becomes even more pronounced when larger contexts or batch sizes are used, as runtime tensors occupy a larger fraction of the total memory. Without strategies to utilize idle memory during valleys, overall memory efficiency remains suboptimal.

Runtime tensor dependence: Waste of communication resources Another critical issue arises from the rigid data dependence during fine-tuning, which leads to underutilization of communication resources. Fine-tuning often requires communication for runtime tensor, such as gathering activations in tensor parallelism. However, communication resources are heavily constrained during these phases, as communication for runtime tensor can only occur after the corresponding tensor has been produced. In contrast, during other phases

[1] PEFT can also select some parameters from the pre-trained model and set as trainable. It has the same pattern with a large number of weights frozen and a small number of weights trainable

that involve only computation, communication resources are largely idle. This strict data dependence prevents preemptive communication, resulting in significant waste of available communication resources. Efficiently addressing this imbalance between computation and communication is key to improving fine-tuning performance.

Inflexible memory accumulation: High peak memory consumption . A third challenge is the rigid method used to accumulate runtime tensor under different memory status. Previous approaches do not account for the varying characteristics of peak and valley phases in memory usage. As a result, they apply the same memory accumulation strategy throughout, leading to excessively high memory pressure during peak phases. This accumulation method leaves little room for optimization at peaks, where memory is already a critical constraint. Consequently, peak memory usage becomes a major bottleneck, limiting the scalability of fine-tuning large models. Addressing this challenge requires flexible memory management strategies that adapt to the different phases of memory utilization.

3 Elastic Tensor

We propose the concept of elastic tensor, an abstraction for dynamic tensor management, where all tensors (e.g., weight parameters, activations, etc.) are treated as dynamic entities, of which real storage can be adjusted during execution. *m*Tuner enables elastic tensor management by providing the following four actions for tensors. These actions control all changes to tensor memory utilization during fine-tuning, thus affect the overhead (computation and communication) related to tensor memory changes.

1. *Gather*: For both static and runtime tensors, it makes remote data partitions locally available. It calls cross-device communication (all-gather) to fetch remote data and increases the ratio that represents how much data is locally available, which varies from $\frac{1}{D}$ (D is the number of devices, indicating tensors evenly partitioned) to 100% (tensors fully replicated on each device).

2. *Discard*: It drops gathered tensor to save memory and decreases the ratio of it with no overhead.

3. *Execute*: It performs model computation and generates runtime tensors. A ratio indicating the fraction of the input batch processed, ranging from 1/B (processing a single sample) to 100% (processing all samples).

4. *Checkpoint*: It saves generated runtime tensor for gradient computation. A ratio describes how many runtime tensors of the module is saved, ranging from 0% (no tensor is stored) to 100% (all runtime tensors in this module are stored).

For dynamic elasticity, *m*Tuner not only can perform these actions with various ratio, but also flexibly set the time for triggering. At the start and end of executing each operation, *m*Tuner can insert actions to change memory usage and utilize communication resource.

*m*Tuner performs operations such as partitioning and communicating tensors to achieve memory management for elastic tensor, just like the existing works such as distributed tensor representation, memory allocator, and different memory caching policies. However, elastic tensor enables conversions between operations that were previously unrelated. For example, trading off between storage and communication efficiency. This allows for a unified representation and transformation between memory, compute, and communication, which in turn optimizes memory planning and execution strategies under memory constraints.

4 Optimizations Enabled by Elastic Tensor

By leveraging elastic tensor, we can flexibly adjust the storage ratio and execution behavior of static and dynamic tensors. This enables further improvements in memory utilization under strict memory constraints and allows memory to be used for relaxing data dependence, thereby better utilizing communication resources and optimizing fine-tuning throughput.

4.1 Temporal Memory Adjustment

As discussed in Section 2.3, the first-in-last-out pattern of runtime tensors results in peaks and valleys in memory usage, which further leads to low temporary memory utilization. *m*Tuner uses elastic tensor to dynamically cache and reuse tensors at the memory valleys to solve this problem.

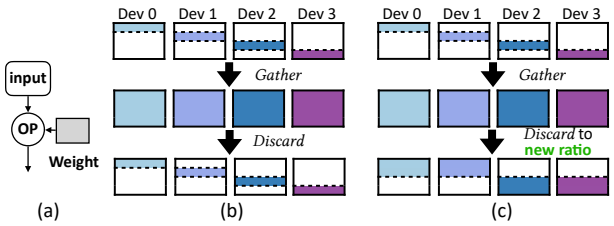


Figure 3: Illustration of *Gather* and *Discard* for weights. When performing computation on an operator (a), FSDP implementation will *Gather* all its weights, perform computation, and *Discard* them to the original ratio (b); In *m*Tuner, *Discard* can be adjusted to other ratio (c);

For fully-sharded data parallel, as shown in Figure 3(a) and (b), during the execution of an operator, each device acquires its complete weights by *gathering*, which is then used for computations. The *gathered* weights are *discarded* after that. As the device possesses complete weight during computation, *m*Tuner can re-adjust the ratio of the weight **without incurring additional communication overhead**. In the example

of Figure 3(c), prior to computation, each device only possess $\frac{1}{D}$ of the weight for the operator, which is then adjusted to $\frac{2}{D}$ after computation. Such dynamic adjustments can be made each time when the weight is used for computation.

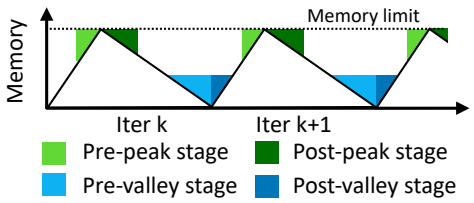


Figure 4: Memory stages resulted by generated runtime tensors

*m*Tuner schedules elastic tensors to improve temporary memory utilization by dynamic adjustment according to current and predicted memory usage trends. As illustrated in Figure 4, during the valley region, frozen weights from the pre-valley stage have a minimal reuse distance. Consequently, *m*Tuner *discards* less of these weights, thereby increasing the ratio of tensors retained in the pre-valley stage. This strategy facilitates the reuse of these weights and reduce the communication cost for *gathering* during forward computation in the post-valley stage. To restore memory efficiency before the pre-peak stage, *m*Tuner compensates by *discarding* more weights at this point, returning the memory state to its base level.

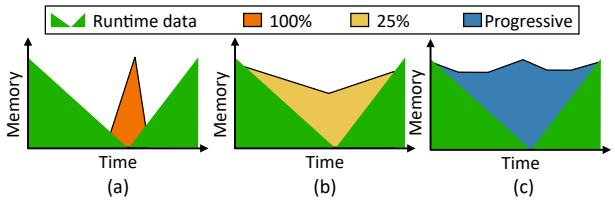


Figure 5: Filling memory valley with single size ((a) and (b)) and progressively changing ratio of elastic tensors (c)

As shown in Figure 5(a), caching frozen weights at a high ratio (e.g., 100%) during the valley can quickly exhaust the available memory, leaving insufficient space to cache other tensors. Conversely, caching frozen weights at a low ratio allows more tensors to be cached but fails to fully utilize the available memory during the valley (Figure 5(b)). Additionally, frequent memory allocation during the peak can lead to fragmentation, increasing the risk of out-of-memory (OOM) errors. To maximize the temporal utilization of available memory, elastic tensor gradually adjusts the discard ratio in the pre-valley region. As in Figure 5(c), closer to the valley, a higher ratio of tensors is cached, while nearer to the peak, the caching ratio is reduced to ensure efficient memory usage and minimize fragmentation risk.

4.2 Dependence-relaxed Communication

Another common parallel strategy in fine-tuning is tensor parallelism (TP). Unlike data parallelism, TP does not require *gathering* weight parameters; instead, it partitions and *gathers* activations for parallel execution, making activation communication a main performance bottleneck. Though communication resources are highly constrained during the TP communication phase, they remain underutilized during other phases dominated by computation.

This underutilization arises due to the data dependency of activations: communication can only begin after activations are generated, preventing it from being initiated in advance. Prior works [2,33] attempt to mitigate this issue by fusing activation communication with computation, breaking workloads into fine-grained tasks to enable partial overlap. However, data dependency still limits the full utilization of available communication resources.

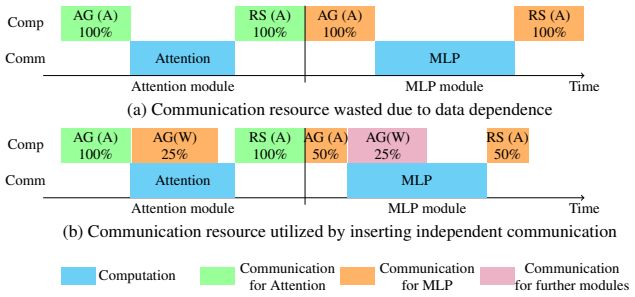


Figure 6: Timeline for attention and MLP module w/ and w/o dependence-relaxed communication

We find that, though data dependence for activation exists, static data, which is independent from execution, can be used to trade for activation communication. Therefore, *m*Tuner can relax the data dependence by elastically inserting communication for weight parameters. The enlarged weight parameter can reduce the communication workload for activation. Figure 6 is an example for an attention module followed by an MLP module in a Transformer layer. (a) shows the communication resource utilization constrained by data dependence of activations. However, in (b), *m*Tuner can insert the communication of weight while performing computation for attention module. With a larger portion of local weights, it can reduce the range and workload of communication for activation in MLP module, thus leads to faster all-gather and reduce-scatter. In MLP module, it can also insert communication for weights of further modules.

Here is a quantitative analysis of the communication savings achieved by dependence-relaxed method. We take input with batch size at B, sequence length at S, hidden size at H, and FFN hidden size F for example. At the start of the MLP module, original method performs a gather ranging from 8 GPUs to collect activations from $[B,S,\frac{H}{8}]$ into $[B,S,H]$, then performs MLP on devices each storing weight parameters

of $[H, \frac{F}{8}]$ and $[\frac{F}{8}, H]$, and performs reduce-scatter to reduce the tensor $[B,S,H]$ into $[B,S,\frac{H}{8}]$. However, with weight parameters prefetched into $[H, \frac{F}{4}]$ and $[\frac{F}{4}, H]$ (25% of the total number of parameters), the communication range is reduced from 8 GPUs to 4 GPUs. The communication at the start and end of TP is the all-gather and reduce-scatter for tensor size at $[\frac{B}{2}, S, \frac{H}{4}]$ and $[\frac{B}{2}, S, H]$, which is reduced by 50%. For scenarios that batch size is tightly constrained and cannot be split, *m*Tuner can split on the dimension of sequence for MLP module.

The percentage of weight prefetched to reduce activation communication is determined by the idleness of communication resource. The communication is fully overlapped and leads to low overhead. On the other hand, with communication of weight and activation interchanged, the efficiency of computation for TP module is improved: The reduction dimension for matrix multiplication gets larger, benefitting the parallelism and the use of TensorCore for GPU.

4.3 Adaptive Data Accumulation

Runtime tensors such as activations are generated and accumulated during the forward phase, and they are consumed and released during the backward phase. The accumulated tensors are part of the memory usage bottleneck. Some methods manipulate on the accumulated memory for better memory utilization. However, as the manipulation uniformly regards the memory at different stages (transformer layers), it causes high activation memory peak usage, therefore limits other optimizations related to memory. *m*Tuner uses elastic tensor to manage the generation and storage of tensors adaptively using the action of *execution* and *checkpoint*.

We observe that runtime tensors generated by different layers exhibit varying lifespans: deeper layers, which are closer to the loss computation, complete backward computations and discard runtime tensors earlier, without affecting computations in shallower layers. Consequently, initiating backward computations sooner for these deeper layers can shorten tensor lifetimes and enable earlier memory release, thereby reducing their residency time. To enable earlier backward execution, *m*Tuner adopts a non-uniform prioritization strategy for deeper layers: it can selectively prioritize partial execution on input tensors (by splitting along the batch dimension) and perform corresponding backward executions for those parts.

As shown in Figure 7(a), for training a 4-layer model on four input samples, the typical procedure involves processing them together, executing four forward computations followed by four backward computations. Shown in Figure 7(b), this process results in the accumulation of activations generated by the four samples across the four layers of forward computations in memory, peaking at the end of the forward stage. *m*Tuner's optimization is shown in Figure 7(c), it can divide execution along the batch dimension, splitting the four samples into two groups of two samples each. From the third

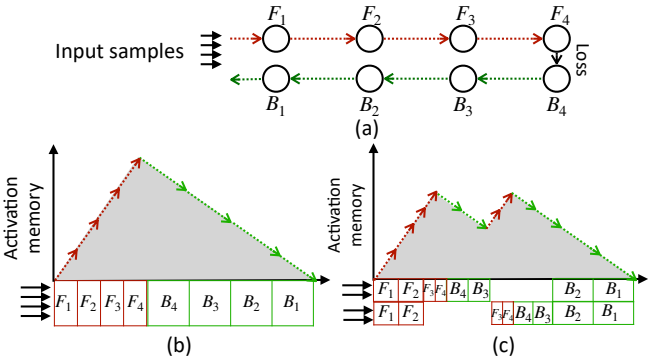


Figure 7: Increasing execution priority for samples to reduce peak activation memory usage; (a) Input samples and computation workflow; (b) Uniform execution priority for all layers; (c) Prioritizing execution priority for part of the samples

layer's forward execution (F_3), only a subset of samples firstly perform forward computation. After completing the forward computations for these samples, instead of executing the forward computations for the remaining samples, *m*Tuner prioritizes the corresponding backward computations for the completed samples, thereby reducing the duration for which these activations are stored in memory. We can observe that the height of the peak has been effectively reduced.

The peak memory usage can be arbitrarily adjusted using this method. Let the total batch size be B, the batch size after splitting be b, and the model consists of L layers. Among them, the l layers closer to the peak adjust the backward priority. Suppose the original peak height is H, then the new peak height is calculated as:

$$H_{new} = \frac{l}{L}\frac{b}{B}H + (1 - \frac{l}{L})H \quad (1)$$

This formula indicates that as the ratio of b to B decreases, the height of the portion with adjusted execution priority becomes lower. Additionally, as more layers are adjusted in terms of peak memory usage, the final peak memory usage will also be reduced. When $l = L$ and $b = 1$, it is equivalent to running the entire model with a batch size of 1, resulting in the lowest peak memory usage.

Moreover, this method enables fine-grained control over the execution batch size. In cases where the specified batch size cannot fully utilize available memory, it is possible to modify the execution priority and batch size of only certain layers, increasing the peak memory usage to fully exploit available memory and further enhance memory utilization.

In terms of performance, the fine-grained data accumulation impacts both computational efficiency and memory usage. From a computational perspective, larger batch sizes generally improve parallelism and enhance GPU utilization. However, in fine-tuning large models, the batch dimension of most operators is determined by the product of batch size and sequence length. Since the sequence length is already large (typically

at least 4K), GPU resources are already well-utilized, and increasing the batch size has limited additional benefit. On the memory side, fine-grained batch splitting allows for more efficient memory usage by better filling available memory and reducing communication overhead. As for accuracy, batch splitting has no impact on the final model performance. This is because gradients from different samples are accumulated before each optimization step. Whether gradients are computed using a single large batch or multiple smaller batches, the accumulation process remains mathematically equivalent.

5 Elastic Tensor Schedule Discovery

The elastic tensor significantly expands the optimization space for memory management during fine-tuning. However, identifying the optimal elastic configuration for a given hardware environment and model setup is challenging due to the large search space. To address this, *m*Tuner adopts a profiling-based approach to measure the time and memory overhead of each module under various partitioning strategies. Additionally, it introduces a search algorithm that simultaneously accounts for peak and valley memory constraints to determine the globally optimal strategy.

5.1 Elastic Tensor Representation

```
Forward:
  gather: 100% # get full MLP param from 50%
  discard: 12.5% # fully partition MLP param
  execute: 50% # execute half of a batch
  checkpoint: 0% # store none of activation
Backward:
  gather: 100% # get full param from 12.5%
  discard: 50% # discard half of param
  execute: 50% # execute half of a batch
# Forward_execute should equal to
    Backward_execute
# The input ratio for Forward_gather is the
    output ratio of Backward_discard (vice
    versa)
```

Listing 1: Example elastic tensor representation on an MLP module on an 8-GPU servers

*m*Tuner represents the schedule for elastic tensor by setting the actions and ratio for each module. To consider both peak and valley memory, we represent elastic tensor for forward and backward pass respectively.

Listing 1 demonstrates the elastic representation for an MLP module on an 8-GPU system. During the forward pass, the elastic tensor gathers the full MLP parameters from a partial partition at 50%, discards part of the parameters to 12.5% after computation, and processes half of the batch without storing any activations to minimize memory consumption. In the backward pass, it retrieves the full parameters from a highly partitioned state of 12.5%, but discards 50% of the parameters after computation, and processes the same batch fraction (50%) as in the forward pass. As it discards less data during backward pass, it has higher valley memory consumption and leads less gathering workloads at forward pass (50% $\rightarrow$ 100%).

5.2 Holistic Schedule Searching

To find an elastic schedule that minimizes the iteration time under memory constraints, the key challenge is that two kinds of memory, peak memory and valley memory, are utilized while constrained by memory limit. Peak memory is traditionally considered, as the runtime tensors are accumulated. Valley memory is the new problem in *m*Tuner, which indicates how *m*Tuner make use of idle memory as runtime tensors are consumed in backward phase.

In addition to memory constraints, there are interdependencies between transformer layers that complicate the scheduling process. Specifically, data required by later layers can be prefetched by earlier layers, meaning that a scheduling decision for one layer influences both memory availability and execution time of subsequent layers. Furthermore, two distinct types of resources—computation resources and communication resources—must be considered, and they can overlap. The interplay of these factors makes it challenging to fully utilize available memory and minimize execution time simultaneously.

To address these challenges, we propose a dual-memory dynamic programming (DP) approach that searches for the minimal execution time while respecting both peak and valley memory constraints. This DP algorithm evaluates different implementation options for each transformer layer and computes the optimal schedule by tracking memory usage at each step. We also improve resource utilization by overlapping communication with computation. By allowing communication to be brought forward, we hide communication latency behind computation.

Algorithm 1 shows the implementation of dual-memory DP. At each layer, multiple implementation choices are available, each with a unique combination of execution time, peak memory, and valley memory consumption. The dual-memory DP approach explores these options recursively, keeping track of both types of memory consumption at each step. For a given layer, the algorithm evaluates every possible implementation choice and updates the schedule only if it results in a lower execution time while respecting the memory limits. The implementation choices includes different parallel strategies and the ratio of stored weights and activations. It maintains a multidimensional state space where each state represents the cumulative execution time under specific peak and valley memory usage. The state transition involves updating execution time by adding the time of the selected implementation and adjusting memory consumption accordingly.

Algorithm 1 Elastic tensor schedule search

Input: Number of layers L, total GPU memory M_{total}, peak memory $M_p^i(j)$, valley memory $M_v^i(j)$, execution time $T^i(j)$ for each layer i and implementation j, number of implementations N_i for each layer i
$dp(i, m_p, m_v) \leftarrow \infty$, $\forall i \in \{1, \ldots, L\}, m_p, m_v \in \{0, \ldots, M_{total}\}$
$dp(0, m_p, m_v) \leftarrow 0, \forall m_p, m_v \in \{0, \ldots, M_{total}\}$ ▷ Initialize
for $i \in \{1, \ldots, L\}$ **do**
  for $m_p \in \{0, \ldots, M_{total}\}$ **do**
    for $m_v \in \{0, \ldots, M_{total}\}$ **do**
      for $j \in \{1, \ldots, N_i\}$ **do**
        if $M_p^i(j) \leq m_p$ **and** $M_v^i(j) \leq m_v$ **then**
          $m'_p \leftarrow m_p - M_p^i(j)$ ▷ remaining peak memory
          $m'_v \leftarrow m_v - M_v^i(j)$ ▷ remaining valley memory
          $dp(i, m_p, m_v) \leftarrow \min(dp(i, m_p, m_v), dp(i - 1, m'_p, m'_v) + T^i(j))$ ▷ Update DP state
$result \leftarrow \min_{m_p, m_v \in \{0, \ldots, M_{total}\}} dp(L, m_p, m_v)$
Backtrack to determine the selected implementation x_i for each layer
Output: Selected implementations $x_1, x_2, \ldots, x_L$ and total execution time

To reduce the search space of DP, we first prune the set of possible implementations for each layer, retaining only those that are likely to yield optimal efficiency. For example, when balancing between communication and storage, although *m*Tuner supports a continuous range of communication-to-storage ratios from 0% to 100%, we observe that significant changes in communication efficiency occur only when the number of involved devices changes. Thus, *m*Tuner only considers storage ratios of $\frac{1}{N}$, where N is an integer greater than or equal to 1. Furthermore, to further limit the number of states, *m*Tuner discretizes memory consumption by rounding it to integer values. This discretization strikes a balance between search efficiency and solution accuracy, significantly reducing the computational complexity of the DP search while maintaining efficiency.

*m*Tuner and its search method are applicable to models like Transformers, which execute sequentially and perform back-propagation of gradients. This applicability is independent of the specific operators used in the model (e.g., different types of attention mechanisms or operations like convolutions instead of MLPs). Moreover, applications like PEFT, which involve frozen parameters, provide *m*Tuner with more opportunities to cache parameters and reduce computation. In contrast, full-parameter fine-tuning, where parameters are distributed across multiple devices, introduces additional communication overhead, which diminishes the optimization benefits of the method.

6 Evaluation

*m*Tuner is an end-to-end fine-tuning system based on PyTorch [35] and Torch-FSDP [57]. It statically and dynamically modifies the model's execution plan to transform a pre-trained model into a distributed, memory-adaptive PEFT model. Before running the model, *m*Tuner utilizes a wrapping method to partition the weights at the granularity of modules (embedding, MLP, and attention, etc.). It also adds trainable modules and parameters based on the user-defined PEFT scheme while freezing other parameters. During runtime, *m*Tuner employs hook methods using pre/post-forward/backward hooks of each module to perform actions. As a result, *m*Tuner can be applied to any LLM model structure without requiring additional modifications from the user.

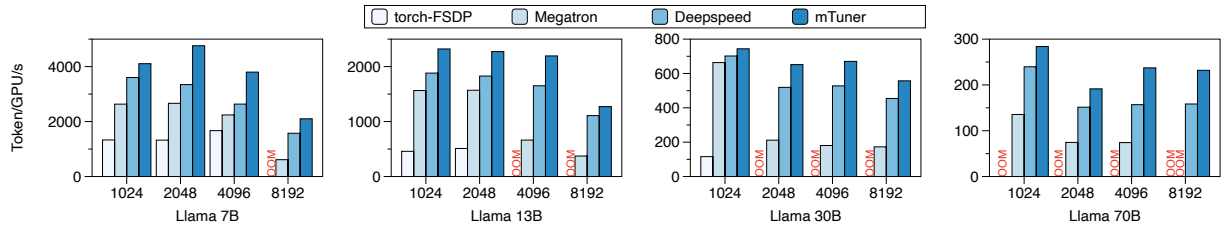
We evaluate *m*Tuner to answer the following questions:

- Can *m*Tuner outperform existing fine-tune systems under various conditions, including model size, sequence length, and hardware settings?
- How elastic tensor influence the memory consumption and performance?
- How searched elastic tensor schedule improve the fine-tuning throughput, respectively for memory peaks and valleys?
- How is the overhead of runtime adjustment of elastic tensor?

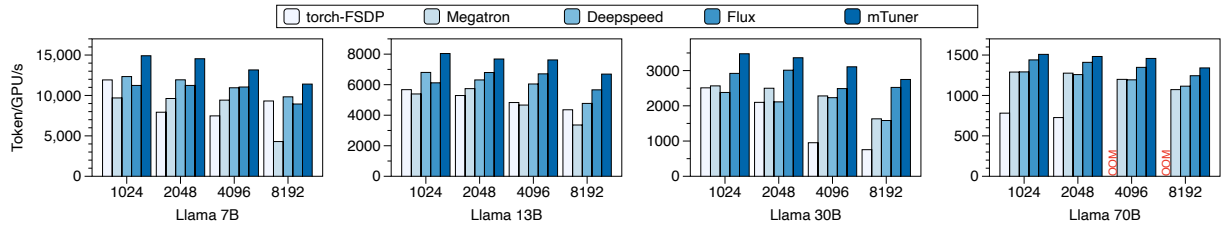
6.1 Setup

Baselines and software configuration. We compare *m*Tuner with state-of-the-art training frameworks including Torch-FSDP@2.1.0 [57], DeepSpeed@0.15 [40], Megatron@0.9.0 [21], and Flux [2]. DeepSpeed employs various levels of ZeRO optimizations to reduce memory consumption and enhance performance. Torch-FSDP shares a similar core idea with DeepSpeed, and is implemented with torch-native components. Megatron is able to concurrently apply 3-D parallelization, including data parallel, tensor parallel, and pipeline parallel, to partition the model and reduce communication cost. Both systems are widely used and applied to pre-training and fine-tuning. Flux overlaps computation and communication by fine-grained partition, and is representative for state-of-the-art communication kernels. For a fair comparison, we run all baselines by setting the maximum batch size to saturate available device memory. They are all based on PyTorch@2.5, CUDA@12.1, FlashAttention@2.4.2 [5, 6], and NCCL@2.21.

Hardware configuration. We tested performance on two types of typical servers equipped with PCIe and NVLink respectively for cross-GPU communication. The PCIe server has eight NVIDIA A100-PCIe-40GB GPUs. They are connected via tree-like PCIe: Every four of them are connected



(a) Single PCIe-server with A100 GPUs



(b) Four NVLink-servers with H100 GPUs

Figure 8: Overall results on various sequence length; The higher throughput (Token/s) is better.

to the same NUMA, and cross NUMA communication goes through QPI between NUMAs. The NVLink server has eight NVIDIA H100-SXM-80GB GPUs connected via NVLink, four servers in total. Compared with PCIe, NVLink has much higher communication throughput.

LLMs and input data. We found that the scale of the LLM (number of parameters) has the most impact on performance, while other computational characteristics are similar across transformer-based models. Therefore, shown in Table 1, we tested different sizes of the Llama 2 model [50], with parameters of 7 billion, 13 billion, 33 billion, and 70 billion. For the input data, we tested samples with different sequence lengths ranging from 1024 to 8192. We use LoRA [15] as the PEFT config, which is commonly used [31] for PEFT.

Table 1: Information of tested models

	num_layer	hidden_size	intermediate_size
Llama 2 7B	32	4096	11008
Llama 2 13B	40	5120	13824
Llama 2 30B	64	6656	17920
Llama 2 70B	80	8192	28672

6.2 Overall Results

Figure 8 illustrates the overall performance of *m*Tuner across different models, input sequence length, and hardware configurations. Figure 8(a) shows the performance on PCIe-server, where the communication is the bottleneck. DeepSpeed is always the best among baselines due to its efficient computation-communication overlapping and communication operators. *m*Tuner can improve the throughput by 28.3% over DeepSpeed. On the other hand, *m*Tuner can achieve an average speedup of 4.15× over Torch-FSDP, on which *m*Tuner is built upon. In smaller models such as Llama 2 7B, where memory is more abundant, *m*Tuner can fully leverage the available memory, resulting in a speedup of 40% and the throughput exceeding 4000 tokens per second. Even in larger models like 30B and 70B, where the model parameters alone consume a significant portion of the memory, *m*Tuner still achieves a 27% speedup, demonstrating the effectiveness of our proposed method across various model sizes.

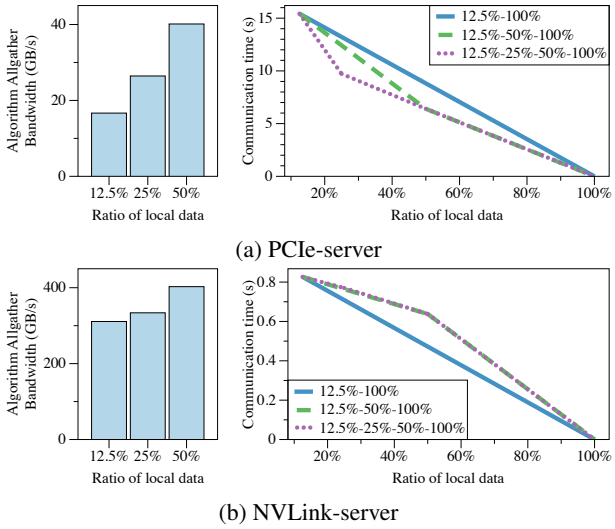
Figure 8(b) shows the performance on four NVLink-servers. Compared with the best baseline, *m*Tuner achieves 14.5% speedup. Flux is the best baseline on most scenarios due to its efficient communication kernel designed for NVLink. *m*Tuner gets improvement over it due to its implementation of better memory utilization and relaxed data dependence.

We find *m*Tuner more effective on inputs with larger sequence lengths. This is because they result in smaller batch sizes and larger activation memory consumption, which further leads to increased communication overhead for each sample with respect to weights. In the case of larger sequences ($\geq$4096), *m*Tuner achieves an average acceleration of 34%.

When comparing PCIe-server and NVLink-server, we find from Figure 8(b) that *m*Tuner achieves a higher speedup ratio in the PCIe-server configuration (28.3% vs. 14.5%). This can be attributed to the more severe communication bottleneck in the PCIe-server setup, while *m*Tuner primarily reduces communication overhead by increasing memory utilization.

6.3 Ablation Study

Ratio of local data influences communication efficiency. Figure 9 shows how varying ratio of local data affects the communication bandwidth of allgather operation. And it shows



(a) PCIe-server

(b) NVLink-server

Figure 9: The communication bandwidth and time with growing ratio of local data on PCIe and NVLink servers

how different schemes to increase the local data ratio influence the overall communication time. For instance, in the case of `12.5%-25%-50%-100%`, the process begins by increasing the local data ratio of elastic tensors in a module from 12.5% to 25%. Once all tensors reach 25% local data, the ratio is further increased to 50%, and so on. When all elastic tensors achieve 100% local data, the communication time effectively drops to zero.

As shown in Figure 9(a), communication bandwidth improves significantly when the elastic tensor has a low initial local data ratio (e.g., 12.5%). Increasing the local storage ratio from 12.5% to 25% yields a 59% bandwidth improvement, from 16GB/s to 26GB/s. However, further increasing the local ratio from 25% to 50%, despite consuming more memory, results in only a 54% speedup. Consequently, the schedule `12.5%-25%-50%-100%` achieves the shortest communication time by efficiently balancing memory usage and communication overhead. In contrast, for the NVLink server shown in Figure 9(b), increasing the local data ratio provides minimal improvement in communication bandwidth. Therefore, the schedule `12.5%-100%`, which directly minimizes communication by avoiding it for specific tensors, achieves the best memory-to-communication efficiency.

Larger batch size leads to less per-sample communication workload. Figure 10 shows how the memory of activation, adjusted with batch size, influences the efficiency of computation and communication. The bars shows under different batch sizes, the iteration time for communication and computation. We find that the communication time is invariant to the batch size, as it is only related to the model size. Therefore, the communication overhead averaged to each sample can be reduced when increasing the batch size. The computation, however, is proportional to batch size as the computational

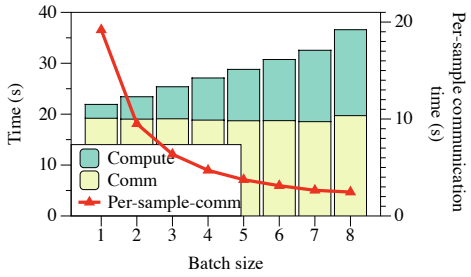


Figure 10: Computation and communication time with different batch sizes on PCIe-server for Llama 2 70B; Per-sample communication time decreases with batch size.

resource is saturated.

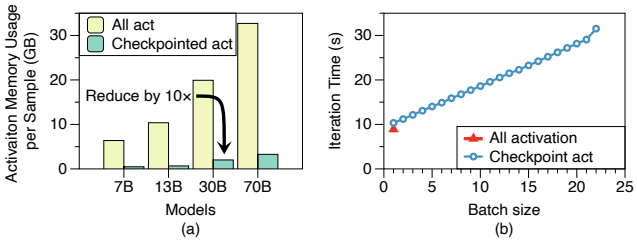


Figure 11: Memory usage of all and checkpointed activations for different models (a) and the execution time comparison (b) on PCIe server

Activation checkpointing saves memory and increases batch size. Figure 11 demonstrates the effect and overhead of activation checkpointing. Figure 11(a) illustrates the memory consumption of all generated activations and the checkpointed activations, with batch size of 1 and sequence length of 1024. For models of different scales, *m*Tuner can reduce the required memory for activations by a factor of 10. This reduction is particularly crucial for Llama 2 70B because preserving all activations would require over 30GB of device memory. Combined with the storage of weights on each device, this would lead to out-of-memory (OOM) errors, making it infeasible even with a batch size of 1.

Figure 11(b) presents the overhead introduced by activation checkpointing on 30B model. Both approaches can be executed with a batch size of 1. However, as the batch size increases, only the approach of checkpointing activations remains viable. By comparing the runtime, the recomputation overhead introduced by checkpointing activations is minimal. This is primarily because the execution time is dominated by communication. And *m*Tuner mitigates the additional communication introduced by activation checkpointing through reuse distance, reducing the runtime overhead associated with activation checkpointing.

Relaxed data dependence improves tensor parallel efficiency. Figure 12 illustrates the benefits of using elastic

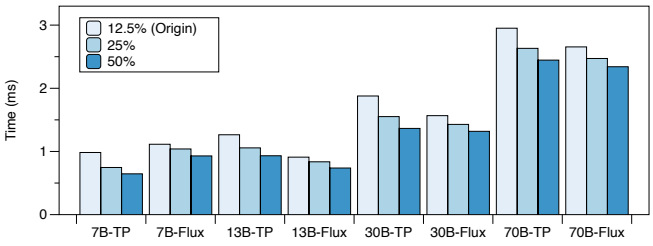


Figure 12: Execution time of an MLP module with tensor parallel size at 8. Data dependence is relaxed with weight parameters *gathered* to ratio of 25% and 50%

tensor to relax data dependence, improving TP efficiency. As shown, increasing the ratio of weights prefetched via early *gather* results in performance gains across various scenarios. Specifically, prefetching 25% and 50% of the weights reduces the overall execution time to 87.6% and 79.0%, respectively. Notably, even though Flux achieves overlapping of TP communication and computation through fine-grained splitting, relaxed data dependence further reduces execution time to 92.3% and 84.2% (corresponding to 25% and 50% prefetch weight, respectively).

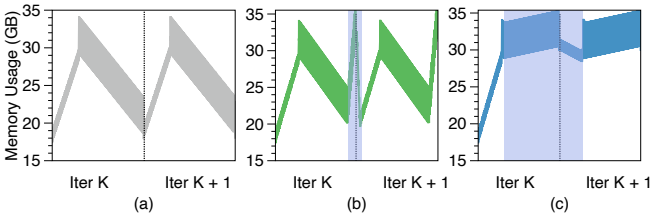


Figure 13: Memory-time curve; (a) No fill in valley area; (b) Filling valley area with 100% elastic tensors; (c) Progressively filling valley area; The colored are valley areas with elastic tensors filled

Temporal memory adjustment increases overall memory utilization. Figure 13 illustrates the changes in memory during the runtime process, with each subfigure representing two iterations. From Figure 13(a), we find that during the forward phase of each iteration, the memory usage keeps increasing because the activations used as checkpoints accumulate in memory. However, during the backward phase, the memory decreases in a zigzag manner. This is because when recomputing based on checkpoints, a significant number of activations that was discarded during the forward propagation is generated, and this memory is immediately consumed during the backward computation.

From Figure 13(b), if *m*Tuner caches and reuses the elastic tensors at the ratio of 100% at the valley area, the memory at the valley quickly fills up, resulting in only a small portion of the operators' weights being optimized, while a significant amount of time is underutilized in memory. Figure 13(c) demonstrates that by progressively filling the valley with elastic tensors, most of the weights can be increased in size for less communication while filling up the memory valley.

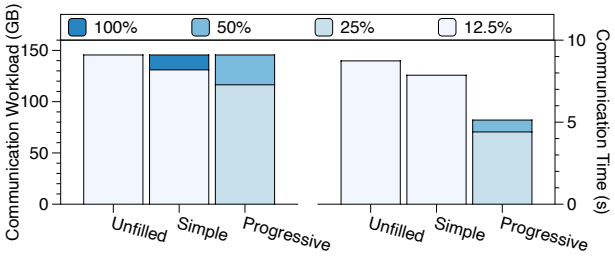


Figure 14: Distribution of elastic tensor ratio and resulted communication time

Temporal memory adjustment decreases communication overhead. Figure 14 illustrates the distribution of elastic tensor ratio and communication time during the forward phase for these three methods. Communication workload refers to the total weights collected, so different filling methods (including unfilling one) shows the same communication workload. We find that the non-progressive filling method only makes 10.3% of the workload to be communicated with elastic tensor at ratio of 100%, reducing the overall communication time by 10% through these locally available weights. However, by using the progressive method to fill the valleys, 80.2% of the weights can be optimized with 25% elastic tensor with 18.0% optimized with 50% elastic tensor, reducing the overall communication time by 41%.

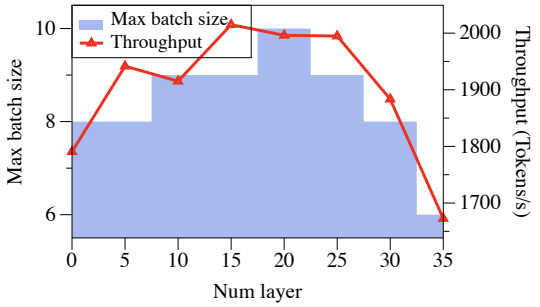


Figure 15: Max batch size and corresponding throughput change with the number of layers at pre-peak stage with high backward priority

Adaptive data accumulation can increase max batch size. Figure 15 illustrates the maximum batch size achievable with *m*Tuner for Llama 2 70B. By adjusting the backward priority of some layers in the pre-peak stage and reducing the peak height, *m*Tuner can increase the batch size per iteration. Additionally, to reduce the overhead of repeated computation for small batches in the flattened peak stage, *m*Tuner increases the elastic tensor of the flattened peak section. Consequently, when more layers are flattened, the memory overhead increases, resulting in a decrease in the maximum batch size.

Moreover, since the averaged communication workload is related to the batch size, the throughput reaches its maximum when the batch size is at its maximum, resulting in a 12% improvement in throughput compared to unflattened peaks.

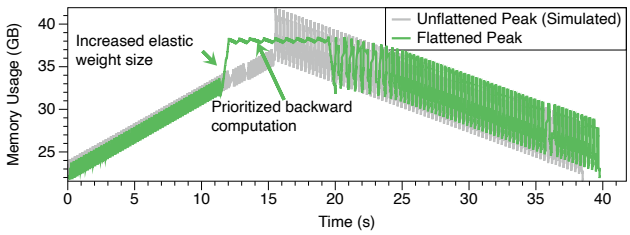


Figure 16: Memory to time curve for training Llama 2 70B at batch size of 10 with and without adaptive runtime tensor accumulation; The unflattened curve suffers from OOM (over 40GB memory usage), and is simulated using profile results.

Peak memory usage is reduced by adaptive data accumulation. As shown in Figure 16, the height of peak can be effectively reduced by adaptive data accumulation. To reduce the overhead of repeated execution at the peak, *m*Tuner increases the elastic tensor size at the pre-peak stage. Though compared to the simulated normal execution, non-uniform execution still introduces overhead and leads to larger execution time, the advantages, such as increased batch size and reduced per-sample communication overhead, still contribute positively to the overall performance.

Table 2: Schedule search time analysis

Time	7B	13B	30B	70B
Search (s)	20.3	25.3	34.5	147.9
Fine-tuning (h)	2.3	4.3	10.0	23.0

Schedule search overhead is acceptable compared with fine-tuning time. Table 2 presents the time required for schedule search. The search time scales with model size, but grows only moderately. We compare the search time with the overall fine-tuning time that tunes the model with one billion tokens on H100 server. The fine-tuning takes hours to finish, which makes the search overhead acceptable.

7 Related Work

Parameter-efficient fine-tuning. Parameter-efficient fine-tuning (PEFT) [8, 16, 24] encompasses three main approaches: adapter-based, prompt-based, and BitFit. Adapter-based methods [7, 14, 15, 25, 37] introduce a small set of trainable modules into the model architecture. A prominent example is low-rank adapters (LoRA) [7, 15], which inject low-rank matrix multiplications as bypasses to efficiently update the model. Prompt-based methods [22, 23, 28] insert trainable tokens into the input prompt, allowing fine-tuning by learning these tokens without manually modifying the prompt. BitFit [54] takes a minimalist approach by updating only the model's bias parameters during fine-tuning. PEFT has demonstrated remarkable effectiveness across various domains, including finance [53], healthcare [29], code [20], and mathematics [27], establishing itself as the leading approach for cost-efficient fine-tuning. QLoRA [7], AdaLoRA [56], and similar approaches [26, 49] differ in their specific definitions of trainable parameters, yet they all fall within the broader framework of PEFT, where only a subset of model parameters is updated. As such, our proposed method is compatible with and can be applied to optimize these techniques.

Systems for fine-tuning. To mitigate hardware limitations during fine-tuning, systems such as FTPipe [10], MPress [59], and Mobius [11] have proposed more efficient pipeline parallelism techniques tailored to the characteristics of PCIe. PetS [60] improves serving efficiency by sharing a single base model across multiple PEFT instances, while S-LoRA [44] offloads PEFT adapters to CPU memory, enabling the deployment of multiple PEFT models on a single GPU. Despite their advancements, these systems do not exploit the frozen weight property inherent to PEFT for optimizing parallel fine-tuning, nor do they provide fine-grained control over memory utilization.

Optimizing static tensors in parallel training. Existing approaches optimize static tensors, such as weight parameters and optimizer states, by leveraging parallel training techniques. DeepSpeed [39, 40, 46] introduces ZeRO, which enhances data parallelism by partitioning static tensors across devices or offloading to CPU memory. To mitigate communication overhead, subsequent works [3, 30] enhance overlapping mechanisms to hide latency effectively. Gpipe [18], Pipedream [32], and MPress [59] employ pipeline parallelism, dividing weights at the layer level across devices and enabling pipelined parallel execution with minimal communication overhead. However, this approach is prone to pipeline bubbles, leading to device idleness. Zero-bubble-pipeline [38] eliminates bubbles by adjusting the pipeline schedule, but accumulated activations still cause significant memory pressure. Tensor parallelism [2, 21, 33] takes a different approach by partitioning weights along an additional dimension, allowing multiple devices to collaboratively process the same sample. Systems like Megatron [21, 45] and Alpa [58, 62] combine data, pipeline, and tensor parallelism, either manually or through automated search, to identify optimal parallel strategies. Despite these efforts, existing methods focus primarily on static memory optimization through parallelism and largely overlook dynamic memory constraints and utilization caused by runtime tensors. Consequently, they lack mechanisms for adaptive, fine-grained memory optimization.

Optimizing runtime tensors with checkpointing and offloading. Checkpointing activation [42] is an important tech-

nique to optimize activation memory, which is also utilized in *m*Tuner for elastic tensor implementation. It is firstly proposed in [4], and widely used in training frameworks like Megatron [21] and MPress [59]. There are works [19] trying to find optimal checkpointing strategy considering model structure. However, these activation optimizations are independent with static tensor optimization, while *m*Tuner focuses on the interplay between them.

8 Conclusion

In this paper, we first observed that memory plays a crucial role in the performance of parameter-efficient fine-tuning. However, existing approaches have low memory utilization due to static scheduling of memory. To address this, we propose *elastic tensors* as an abstraction to dynamically adjust the memory sizes of tensors based on memory usage and the fine-tuning process. Based on elastic tensor, we developed *m*Tuner, a system that enables efficient fine-tuning of large models on multi-GPU servers. Compared to existing state-of-the-art systems, *m*Tuner achieves an improvement of up to 51.2% and 24.8% (28.3% and 14.5% on average) on PCIe and NVLink servers.

Acknowledgments

We would like to thank the anonymous reviewers and our shepherd, Abhilash Jindal, for their valuable suggestions. This work is supported by the National Key R&D Program of China under Grant 2023YFB3001704, NSFC for Distinguished Young Scholar (62225206), Beijing Natural Science Foundation (L242017), Strategic Priority Research Program of Chinese Academy of Sciences (XDB0500103), and Tsinghua University Initiative Scientific Research Program. Jidong Zhai is the corresponding author of this paper (zhaijidong@tsinghua.edu.cn).

References

[1] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020.

[2] Li-Wen Chang, Wenlei Bao, Qi Hou, Chengquan Jiang, Ningxin Zheng, Yinmin Zhong, Xuanrun Zhang, Zuquan Song, Chengji Yao, Ziheng Jiang, Haibin Lin, Xin Jin, and Xin Liu. Flux: Fast software-based communication overlap on gpus through kernel fusion, 2024.

[3] Chang Chen, Xiuhong Li, Qianchao Zhu, Jiangfei Duan, Peng Sun, Xingcheng Zhang, and Chao Yang. Centauri: Enabling efficient scheduling for communication-computation overlap in large model training via communication partitioning. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS '24, page 178–191, New York, NY, USA, 2024. Association for Computing Machinery.

[4] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. Training deep nets with sublinear memory cost. *CoRR*, abs/1604.06174, 2016.

[5] Tri Dao. FlashAttention-2: Faster attention with better parallelism and work partitioning. 2023.

[6] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. FlashAttention: Fast and memory-efficient exact attention with IO-awareness. In *Advances in Neural Information Processing Systems*, 2022.

[7] Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. Qlora: Efficient finetuning of quantized llms. *CoRR*, abs/2305.14314, 2023.

[8] Ning Ding, Yujia Qin, Guang Yang, Fuchao Wei, Zonghan Yang, Yusheng Su, Shengding Hu, Yulin Chen, Chi-Min Chan, Weize Chen, Jing Yi, Weilin Zhao, Xiaozhi Wang, Zhiyuan Liu, Hai-Tao Zheng, Jianfei Chen, Yang Liu, Jie Tang, Juanzi Li, and Maosong Sun. Delta tuning: A comprehensive study of parameter efficient methods for pre-trained language models. *CoRR*, abs/2203.06904, 2022.

[9] Jesse Dodge, Gabriel Ilharco, Roy Schwartz, Ali Farhadi, Hannaneh Hajishirzi, and Noah Smith. Fine-tuning pretrained language models: Weight initializations, data orders, and early stopping. *arXiv preprint arXiv:2002.06305*, 2020.

[10] Saar Eliad, Ido Hakimi, Alon De Jagger, Mark Silberstein, and Assaf Schuster. Fine-tuning giant neural networks on commodity hardware with automatic pipeline model parallelism. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 381–396. USENIX Association, July 2021.

[11] Yangyang Feng, Minhui Xie, Zijie Tian, Shuo Wang, Youyou Lu, and Jiwu Shu. Mobius: Fine tuning large-scale models on commodity GPU servers. In Tor M.

Aamodt, Natalie D. Enright Jerger, and Michael M. Swift, editors, *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2023, Vancouver, BC, Canada, March 25-29, 2023*, pages 489–501. ACM, 2023.

[12] Hao Ge, Fangcheng Fu, Haoyang Li, Xuanyu Wang, Sheng Lin, Yujie Wang, Xiaonan Nie, Hailin Zhang, Xupeng Miao, and Bin Cui. Enabling parallelism hot switching for efficient training of large language models. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, pages 178–194, 2024.

[13] Michael A Hedderich, Lukas Lange, Heike Adel, Jannik Strötgen, and Dietrich Klakow. A survey on recent approaches for natural language processing in low-resource scenarios. *arXiv preprint arXiv:2010.12309*, 2020.

[14] Neil Houlsby, Andrei Giurgiu, Stanislaw Jastrzebski, Bruna Morrone, Quentin de Laroussilhe, Andrea Gesmundo, Mona Attariyan, and Sylvain Gelly. Parameter-efficient transfer learning for NLP. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA*, volume 97 of *Proceedings of Machine Learning Research*, pages 2790–2799. PMLR, 2019.

[15] Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, and Weizhu Chen. Lora: Low-rank adaptation of large language models. *CoRR*, abs/2106.09685, 2021.

[16] Shengding Hu, Ning Ding, Weilin Zhao, Xingtai Lv, Zhen Zhang, Zhiyuan Liu, and Maosong Sun. Opendelta: A plug-and-play library for parameter-efficient adaptation of pre-trained models. *arXiv preprint arXiv:2307.03084*, 2023.

[17] Gabriel Huang, Issam Laradji, David Vazquez, Simon Lacoste-Julien, and Pau Rodriguez. A survey of self-supervised and few-shot object detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 45(4):4071–4089, 2022.

[18] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Xu Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V. Le, Yonghui Wu, and Zhifeng Chen. Gpipe: Efficient training of giant neural networks using pipeline parallelism. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pages 103–112, 2019.

[19] Paras Jain, Ajay Jain, Aniruddha Nrusimha, Amir Gholami, Pieter Abbeel, Kurt Keutzer, Ion Stoica, and Joseph Gonzalez. Checkmate: Breaking the memory wall with optimal tensor rematerialization. In Inderjit S. Dhillon, Dimitris S. Papailiopoulos, and Vivienne Sze, editors, *Proceedings of Machine Learning and Systems 2020, MLSys 2020, Austin, TX, USA, March 2-4, 2020*. mlsys.org, 2020.

[20] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues? *CoRR*, abs/2310.06770, 2023.

[21] Vijay Korthikanti, Jared Casper, Sangkug Lym, Lawrence McAfee, Michael Andersch, Mohammad Shoeybi, and Bryan Catanzaro. Reducing activation recomputation in large transformer models. *CoRR*, abs/2205.05198, 2022.

[22] Brian Lester, Rami Al-Rfou, and Noah Constant. The power of scale for parameter-efficient prompt tuning. In Marie-Francine Moens, Xuanjing Huang, Lucia Specia, and Scott Wen-tau Yih, editors, *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 7-11 November, 2021*, pages 3045–3059. Association for Computational Linguistics, 2021.

[23] Xiang Lisa Li and Percy Liang. Prefix-tuning: Optimizing continuous prompts for generation. *CoRR*, abs/2101.00190, 2021.

[24] Vladislav Lialin, Vijeta Deshpande, and Anna Rumshisky. Scaling down to scale up: A guide to parameter-efficient fine-tuning. *CoRR*, abs/2303.15647, 2023.

[25] Haokun Liu, Derek Tam, Mohammed Muqeeth, Jay Mohta, Tenghao Huang, Mohit Bansal, and Colin Raffel. Few-shot parameter-efficient fine-tuning is better and cheaper than in-context learning. In Sanmi Koyejo, S. Mohamed, A. Agarwal, Danielle Belgrave, K. Cho, and A. Oh, editors, *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*, 2022.

[26] Shih-Yang Liu, Chien-Yi Wang, Hongxu Yin, Pavlo Molchanov, Yu-Chiang Frank Wang, Kwang-Ting Cheng, and Min-Hung Chen. Dora: Weight-decomposed low-rank adaptation. In *Forty-first International Conference on Machine Learning*, 2024.

[27] Tiedong Liu and Bryan Kian Hsiang Low. Goat: Fine-tuned llama outperforms GPT-4 on arithmetic tasks. *CoRR*, abs/2305.14201, 2023.

[28] Xiao Liu, Kaixuan Ji, Yicheng Fu, Zhengxiao Du, Zhilin Yang, and Jie Tang. P-tuning v2: Prompt tuning can be comparable to fine-tuning universally across scales and tasks. *CoRR*, abs/2110.07602, 2021.

[29] Zhengliang Liu, Yiwei Li, Peng Shu, Aoxiao Zhong, Longtao Yang, Chao Ju, Zihao Wu, Chong Ma, Jie Luo, Cheng Chen, Sekeun Kim, Jiang Hu, Haixing Dai, Lin Zhao, Dajiang Zhu, Jun Liu, Wei Liu, Dinggang Shen, Tianming Liu, Quanzheng Li, and Xiang Li. Radiology-llama2: Best-in-class large language model for radiology. *CoRR*, abs/2309.06419, 2023.

[30] Kshiteej Mahajan, Ching-Hsiang Chu, Srinivas Sridharan, and Aditya Akella. Better together: Jointly optimizing ML collective scheduling and execution planning using SYNDICATE. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 809–824, Boston, MA, April 2023. USENIX Association.

[31] Sourab Mangrulkar, Sylvain Gugger, Lysandre Debut, Younes Belkada, Sayak Paul, and Benjamin Bossan. Peft: State-of-the-art parameter-efficient fine-tuning methods. https://github.com/huggingface/peft, 2022.

[32] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, Phillip B. Gibbons, and Matei Zaharia. Pipedream: generalized pipeline parallelism for DNN training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019*, pages 1–15. ACM, 2019.

[33] NVIDIA. Transformer engine. https://github.com/NVIDIA/TransformerEngine, 2025.

[34] OpenAI. GPT-4 technical report. *CoRR*, abs/2303.08774, 2023.

[35] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

[36] Xuan Peng, Xuanhua Shi, Hulin Dai, Hai Jin, Weiliang Ma, Qian Xiong, Fan Yang, and Xuehai Qian. Capuchin: Tensor-based GPU memory management for deep learning. In James R. Larus, Luis Ceze, and Karin Strauss, editors, *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020*, pages 891–905. ACM, 2020.

[37] Jonas Pfeiffer, Aishwarya Kamath, Andreas Rücklé, Kyunghyun Cho, and Iryna Gurevych. Adapterfusion: Non-destructive task composition for transfer learning. In Paola Merlo, Jörg Tiedemann, and Reut Tsarfaty, editors, *Proceedings of the 16th Conference of the European Chapter of the Association for Computational Linguistics: Main Volume, EACL 2021, Online, April 19 - 23, 2021*, pages 487–503. Association for Computational Linguistics, 2021.

[38] Penghui Qi, Xinyi Wan, Guangxing Huang, and Min Lin. Zero bubble (almost) pipeline parallelism. In *The Twelfth International Conference on Learning Representations*, 2024.

[39] Samyam Rajbhandari, Olatunji Ruwase, Jeff Rasley, Shaden Smith, and Yuxiong He. Zero-infinity: breaking the GPU memory wall for extreme scale deep learning. In Bronis R. de Supinski, Mary W. Hall, and Todd Gamblin, editors, *International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2021, St. Louis, Missouri, USA, November 14-19, 2021*, page 59. ACM, 2021.

[40] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters. In *KDD '20: The 26th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, Virtual Event, CA, USA, August 23-27, 2020*, pages 3505–3506. ACM, 2020.

[41] Jie Ren, Samyam Rajbhandari, Reza Yazdani Aminabadi, Olatunji Ruwase, Shuangyan Yang, Minjia Zhang, Dong Li, and Yuxiong He. Zero-offload: Democratizing billion-scale model training. In Irina Calciu and Geoff Kuenning, editors, *2021 USENIX Annual Technical Conference, USENIX ATC 2021, July 14-16, 2021*, pages 551–564. USENIX Association, 2021.

[42] Tim Salimans and Yaroslav Bulatov. Gradient checkpointing. https://github.com/cybertronai/gradient-checkpointing, 2025.

[43] Teven Le Scao, Angela Fan, Christopher Akiki, Ellie Pavlick, Suzana Ilic, Daniel Hesslow, Roman Castagné, Alexandra Sasha Luccioni, François Yvon, Matthias Gallé, Jonathan Tow, Alexander M. Rush, Stella Biderman, Albert Webson, Pawan Sasanka Ammanamanchi, Thomas Wang, Benoît Sagot, Niklas Muennighoff, Albert Villanova del Moral, Olatunji Ruwase, Rachel Bawden, Stas Bekman, Angelina McMillan-Major, Iz Beltagy, Huu Nguyen, Lucile Saulnier, Samson Tan, Pedro Ortiz Suarez, Victor Sanh, Hugo Laurençon, Yacine Jernite, Julien Launay, Margaret Mitchell, Colin Raffel, Aaron Gokaslan, Adi Simhi, Aitor Soroa, Alham Fikri

Aji, Amit Alfassy, Anna Rogers, Ariel Kreisberg Nitzav, Canwen Xu, Chenghao Mou, Chris Emezue, Christopher Klamm, Colin Leong, Daniel van Strien, David Ifeoluwa Adelani, and et al. BLOOM: A 176b-parameter open-access multilingual language model. *CoRR*, abs/2211.05100, 2022.

[44] Ying Sheng, Shiyi Cao, Dacheng Li, Coleman Hooper, Nicholas Lee, Shuo Yang, Christopher Chou, Banghua Zhu, Lianmin Zheng, Kurt Keutzer, Joseph E. Gonzalez, and Ion Stoica. S-lora: Serving thousands of concurrent lora adapters, 2023.

[45] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *CoRR*, abs/1909.08053, 2019.

[46] Siddharth Singh, Olatunji Ruwase, Ammar Ahmad Awan, Samyam Rajbhandari, Yuxiong He, and Abhinav Bhatele. A hybrid tensor-expert-data parallelism approach to optimize mixture-of-experts training. In Kyle A. Gallivan, Efstratios Gallopoulos, Dimitrios S. Nikolopoulos, and Ramón Beivide, editors, *Proceedings of the 37th International Conference on Supercomputing, ICS 2023, Orlando, FL, USA, June 21-23, 2023*, pages 203–214. ACM, 2023.

[47] Chi Sun, Xipeng Qiu, Yige Xu, and Xuanjing Huang. How to fine-tune bert for text classification? In *Chinese Computational Linguistics: 18th China National Conference, CCL 2019, Kunming, China, October 18–20, 2019, Proceedings 18*, pages 194–206. Springer, 2019.

[48] Zhenbo Sun, Huanqi Cao, Yuanwei Wang, Guanyu Feng, Shengqi Chen, Haojie Wang, and Wenguang Chen. Adapipe: Optimizing pipeline parallelism with adaptive recomputation and partitioning. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS '24, page 86–100, New York, NY, USA, 2024. Association for Computing Machinery.

[49] Chunlin Tian, Zhan Shi, Zhijiang Guo, Li Li, and Cheng-Zhong Xu. Hydralora: An asymmetric lora architecture for efficient fine-tuning. *Advances in Neural Information Processing Systems*, 37:9565–9584, 2024.

[50] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton-Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurélien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. Llama 2: Open foundation and fine-tuned chat models. *CoRR*, abs/2307.09288, 2023.

[51] Colin Unger, Zhihao Jia, Wei Wu, Sina Lin, Mandeep Baines, Carlos Efrain Quintero Narvaez, Vinay Ramakrishnaiah, Nirmal Prajapati, Pat McCormick, Jamaludin Mohd-Yusof, et al. Unity: Accelerating {DNN} training through joint optimization of algebraic transformations and parallelization. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 267–284, 2022.

[52] Mitchell Wortsman, Gabriel Ilharco, Jong Wook Kim, Mike Li, Simon Kornblith, Rebecca Roelofs, Raphael Gontijo Lopes, Hannaneh Hajishirzi, Ali Farhadi, Hongseok Namkoong, et al. Robust fine-tuning of zero-shot models. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 7959–7971, 2022.

[53] Hongyang Yang, Xiao-Yang Liu, and Christina Dan Wang. Fingpt: Open-source financial large language models. *CoRR*, abs/2306.06031, 2023.

[54] Elad Ben Zaken, Yoav Goldberg, and Shauli Ravfogel. Bitfit: Simple parameter-efficient fine-tuning for transformer-based masked language-models. In Smaranda Muresan, Preslav Nakov, and Aline Villavicencio, editors, *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers), ACL 2022, Dublin, Ireland, May 22-27, 2022*, pages 1–9. Association for Computational Linguistics, 2022.

[55] Junzhe Zhang, Sai-Ho Yeung, Yao Shu, Bingsheng He, and Wei Wang. Efficient memory management for gpu-based deep learning systems. *CoRR*, abs/1903.06631, 2019.

[56] Qingru Zhang, Minshuo Chen, Alexander Bukharin, Nikos Karampatziakis, Pengcheng He, Yu Cheng, Weizhu Chen, and Tuo Zhao. Adalora: Adaptive budget allocation for parameter-efficient fine-tuning. *arXiv preprint arXiv:2303.10512*, 2023.

[57] Yanli Zhao, Andrew Gu, Rohan Varma, Liang Luo, Chien-Chin Huang, Min Xu, Less Wright, Hamid Shojanazeri, Myle Ott, Sam Shleifer, Alban Desmaison, Can Balioglu, Bernard Nguyen, Geeta Chauhan, Yuchen Hao, and Shen Li. Pytorch FSDP: experiences on scaling fully sharded data parallel. *CoRR*, abs/2304.11277, 2023.

[58] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P. Xing, Joseph E. Gonzalez, and Ion Stoica. Alpa: Automating inter- and intra-operator parallelism for distributed deep learning. In *16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2022, Carlsbad, CA, USA, July 11-13, 2022*, pages 559–578. USENIX Association, 2022.

[59] Quan Zhou, Haiquan Wang, Xiaoyan Yu, Cheng Li, Youhui Bai, Feng Yan, and Yinlong Xu. Mpress: Democratizing billion-scale model training on multi-gpu servers via memory-saving inter-operator parallelism. In *IEEE International Symposium on High-Performance Computer Architecture, HPCA 2023, Montreal, QC, Canada, February 25 - March 1, 2023*, pages 556–569. IEEE, 2023.

[60] Zhe Zhou, Xuechao Wei, Jiejing Zhang, and Guangyu Sun. {PetS}: A unified framework for {Parameter-Efficient} transformers serving. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 489–504, 2022.

[61] Fuzhen Zhuang, Zhiyuan Qi, Keyu Duan, Dongbo Xi, Yongchun Zhu, Hengshu Zhu, Hui Xiong, and Qing He. A comprehensive survey on transfer learning. *Proceedings of the IEEE*, 109(1):43–76, 2020.

[62] Yonghao Zhuang, Hexu Zhao, Lianmin Zheng, Zhuohan Li, Eric P. Xing, Qirong Ho, Joseph E. Gonzalez, Ion Stoica, and Hao Zhang. On optimizing the communication of model parallelism. *CoRR*, abs/2211.05322, 2022.

A Artifact Appendix

Abstract

This artifact includes the source codes and experiments for replicating the evaluations in this paper.

Description & Requirements

How to access

*m*Tuner is publicly available at https://github.com/xxcclong/mTuner.

Hardware dependencies

- NVIDIA Tesla H100/A100 GPU
- Memory > 400GB
- Disk space > 100GB

Software dependencies

We list the most important software we used:

- CUDA 12.1
- Torch==2.5.1+cu121
- triton==3.1.0

They can be installed by following the instructions.

Evaluation workflow

Using scripts here to run all experiments.

Major Claims

- *m*Tuner achieves throughput improvements on PCIe and NVLink servers for LLMs from 7B to 70B.
- Search overhead of *m*Tuner is kept to tens of seconds.

Please refer to the README to reproduce the results.

JENGA: Enhancing LLM Long-Context Fine-tuning with Contextual Token Sparsity

Tuowei Wang
Tsinghua University

Xingyu Chen
Tsinghua University

Kun Li
Microsoft Research

Ting Cao
Microsoft Research

Ju Ren*
Tsinghua University

Yaoxue Zhang
Tsinghua University

Abstract

The escalating demand for long-context applications has intensified the necessity of extending the LLM context windows. Despite recent fine-tuning approaches successfully expanding context lengths, their high memory footprints, especially for activations, present a critical practical limitation. Current parameter-efficient fine-tuning methods prioritize reducing parameter update overhead over addressing activation memory constraints. Similarly, existing sparsity mechanisms improve computational efficiency but overlook activation memory optimization due to the phenomenon of *Shadowy Activation*.

In this paper, we propose JENGA, the first LLM fine-tuning system that explores and exploits a new token-level sparsity mechanism inherent in long-context scenarios, termed *Contextual Token Sparsity*. JENGA minimizes redundant token involvement by assessing the informativeness of token embeddings while preserving model accuracy. Specifically, JENGA introduces three key techniques: (1) Token Elimination, dynamically identifying and excluding redundant tokens across varying inputs and layers. (2) Pattern Prediction, utilizing well-trained predictors to approximate token sparsity patterns with minimal overhead. (3) Kernel Optimization, employing permutation-free and segment-based strategies to boost system performance. We implement JENGA as an end-to-end fine-tuning system compatible with various LLM architectures and other optimization techniques[1]. Comprehensive evaluations demonstrate that JENGA reduces memory consumption by up to 1.93× and achieves up to 1.36× speedups, outperforming state-of-the-art fine-tuning systems.

1 Introduction

As the demand for comprehensive document analysis [13, 26], extended multi-turn dialogues [76], and intricate codebase handling [51, 54] grows, large language models (LLMs) with larger context windows are becoming integral to AI applications. Despite their utility, LLMs [1, 31, 52] are typically pre-trained with fixed context windows, such as the 4K token limit in Llama2 [67]. When these models encounter inputs exceeding this limit, their performance deteriorates markedly [18]. The discrepancy between the fixed context window during pre-training and the increasingly extended inputs during inference has emerged as a critical challenge in real-world deployments.

Table 1: Memory footprint (GB) comparison across different fine-tuning methods. LoRA and LongLoRA are representative of PEFT and sparsity-based methods, respectively ($S = 4K$).

Model	Llama2-7B	Llama3-8B	Mistral-7B	OPT-6.7B
Naive	67.9	78.4	73.4	63.8
LoRA	39.2	43.4	39.3	36.1
LongLoRA	41.3	43.9	39.3	38.1
JENGA	**31.3**	**34.5**	**31.4**	**30.0**

Recent studies [43, 53, 68, 73] show that the context window of pre-trained LLMs can be extended through fine-tuning on longer sequences. Unfortunately, managing these extended sequences imposes substantial resource challenges, particularly in terms of memory consumption. For instance, Position Interpolation [10] extends Llama models [66] from 2K to 32K but requires 128 A100 80GB GPUs, mainly due to memory limitations. Rather than model parameters, the primary memory bottleneck in long-context fine-tuning arises from *activations* [11, 35, 79], which include intermediate results and gradients that scale proportionally with sequence length.

Although various techniques have been proposed to improve the efficiency of fine-tuning, the substantial demands of activation memory remain largely unaddressed. By adapting only a minimal number of parameters, parameter-efficient fine-tuning (PEFT) methods [28, 29, 39, 78] reduce the memory requirement of parameter updates but leave activation memory unoptimized. Furthermore, recent works [12, 24, 50, 70] incorporate diverse sparse mechanisms to approximate standard dense attention. Despite achieving considerable compu-

*Corresponding author (renju@tsinghua.edu.cn)
[1]Source code: https://github.com/Pairshoe/Jenga-AE

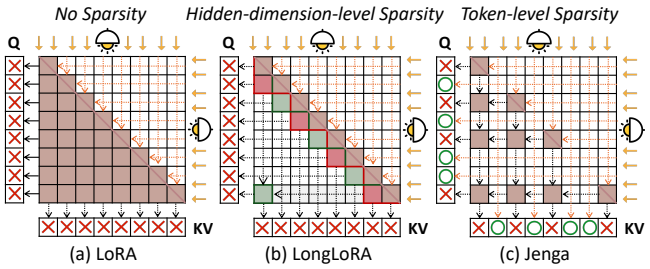


Figure 1: Illustration of shadowy activation. (a) LoRA performs full attention computation without incorporating sparsity. (b) LongLoRA adopts two shifted local attention patterns (colored in red and green) to approximate full attention, targeting sparsity at the hidden-dimension level. (c) JENGA employs token-level sparsity, minimizing token involvement and achieving activation savings compared to other methods.

tational savings, these methods fail to offer additional memory reduction, as highlighted in Table 1.

This limitation arises because existing sparsity mechanisms primarily focus on sparsity within the hidden dimension of individual tokens. As shown in Figure 1(b), while each token interacts with a reduced subset of tokens compared to dense attention, the entire token sequence remains involved. Crucially, once a token participates in the computation, regardless of the extent of usage, its activations are retained in memory—a phenomenon we term ***Shadowy Activation***. Given the large number of tokens involved in long-context fine-tuning, this observation prompts a critical question: Can we propose a novel LLM fine-tuning scheme with *minimal token involvement* that optimizes *both memory and computational efficiency*?

In this paper, we present **JENGA**, an efficient system for enhancing long-context fine-tuning of LLMs. The design of JENGA is rooted in an intuitive yet profound observation: natural language exhibits significant redundancy, particularly in long-context scenarios. This redundancy, proved to be ubiquitous in linguistic studies [60, 71], has been utilized across multiple LLM-related domains, including data engineering [22, 23], prompt compression [32, 40], and inference optimization [38, 61, 63, 82]. Most importantly, standard full attention can be effectively approximated by focusing on interactions among a small subset of the most informative tokens in a long text sequence. However, to our knowledge, no research has leveraged this approach to improve fine-tuning efficiency. Therefore, the key insight of JENGA is to *identify and retain only the most informative tokens, enabling long-context fine-tuning with minimized token involvement*.

However, this is not a long-hanging fruit, as this token-level sparsity exhibits unique characteristics in LLM long-context fine-tuning. Specifically, the token embeddings with the highest informativeness vary across different inputs and layers, a mechanism we term ***Contextual Token Sparsity***. Harnessing this mechanism effectively and efficiently demands heuristic designs that address both algorithmic and system-level considerations. Specifically, several challenges must be tackled:

Challenge 1: Identification. While long-context texts inherently exhibit significant redundancy, accurately identifying these redundant tokens is non-trivial. Eliminating informative tokens risks degrading model performance while retaining excessive tokens results in inefficient resource utilization.

Challenge 2: Detection. Given the dynamic nature of contextual token sparsity, the optimal sparsity patterns vary across different inputs and layers. This necessitates a selection mechanism capable of adapting to these variations at runtime while incurring minimal computational and memory overhead.

Challenge 3: Performance. The variability in token selection across layers incurs additional token movements, increasing memory access costs. Besides, naive gradient computation of model loss amplifies activation memory usage, resulting in inefficient memory peaks, especially in long-context scenarios.

JENGA consists of three fundamental techniques designed to address these emerging challenges, respectively:

Information-driven Token Elimination. JENGA evaluates token informativeness by analyzing its interactions with other tokens, selectively excluding less informative ones from computation. Moreover, JENGA exploits sparsity variations across layers by applying layer-specific thresholds, which maximizes resource efficiency while preserving model accuracy.

Context-aware Pattern Prediction. JENGA utilizes a neural-network-based approach for predicting optimal sparsity patterns that adapt to the context of the current input. Besides, JENGA introduces a technique that elastically transforms the predictor's parameter size, minimizing the memory and computational costs associated with the prediction process.

High-performance Kernel Optimization. JENGA adapts a permutation-free strategy to eliminate unnecessary global memory data movement during token selection and padding across different layers. Furthermore, JENGA develops a segmented-based gradient computation method, effectively alleviating activation memory peak in long-context scenarios.

We evaluate JENGA across two representative families of LLMs and on three distinct GPU architectures. The results demonstrate that JENGA achieves a memory reduction of 1.93× in end-to-end fine-tuning, compared to the state-of-the-art fine-tuning systems, while maintaining model accuracy.

To the best of our knowledge, JENGA is the first fine-tuning system that exploits the inherent token-level sparsity in long-context scenarios. This innovation allows JENGA to handle long-context sequences while demanding resources for fewer tokens. In summary, we make the following contributions:

- We identify a new sparsity mechanism, contextual token sparsity, inherent in long-context fine-tuning, enabling optimizing both memory and computational efficiency.

- We develop three key techniques that identify, detect, and perform contextual token sparsity, respectively. Our

design provides a heuristic fine-tuning scheme, encompassing both algorithmic and system-level optimizations.

- We implement these techniques as an end-to-end fine-tuning system compatible with various LLM architectures, enabling seamless integration with other optimization techniques to further enhance performance.
- We conduct comprehensive evaluations of JENGA across diverse LLM families and hardware platforms. The results show that JENGA achieves 1.93× memory savings and 1.36× speedups compared to the state-of-the-art.

2 Background and Motivation

2.1 Efficient LLM Fine-tuning

Fine-tuning, the process of adapting pre-trained LLMs to diverse downstream applications, is pivotal for their effective deployment. Particularly, fine-tuning a pre-trained LLM on longer text sequences is essential for extending its pre-defined context window size, enabling support for long-context scenarios. However, this process is typically resource-intensive, with the inclusion of long text sequences considerably amplifying both computational and memory requirements.

To improve fine-tuning efficiency, one promising direction is parameter-efficient fine-tuning (PEFT) methods [28, 29, 39, 78], which adapt pre-trained models by updating only a subset of parameters while maintaining model performance. A representative example is low-rank adaption (LoRA) [29], which freezes the pre-trained model weights and injects smaller, trainable low-rank matrices into each transformer block. By reducing the number of trainable parameters, LoRA markedly alleviates the memory demands of optimizer states, driving its widespread adoption in practical applications.

Another research direction [6, 24, 77, 83] focuses on exploiting the inherent sparsity [17] within attention mechanisms by employing various sparsity patterns to the standard dense attention. The central insight is that only a limited subset of interactions is critical, allowing the remainder to be safely disregarded with minimal impact on accuracy. More recent works [12, 70] integrate these two lines of research. Notably, LongLoRA [12] extends LoRA by incorporating a new shifted sparsity pattern, enabling efficient scaling of context length. By substituting global dense attention with two groups of shifted sparse local attention, LongLoRA achieves less training time compared to LoRA. However, subsequent analysis reveals that neither LoRA nor LongLoRA adequately addresses the memory bottleneck in long-context fine-tuning, as activations scale proportionally with sequence length.

2.2 Analysis: Fine-tuning Memory Breakdown

Here we provide a detailed analysis of memory consumption during LLM fine-tuning process. Beginning with vanilla fine-tuning, we extend our discussion to include two representative approaches: LoRA, a PEFT method, and LongLoRA, a sparsity-based technique. Our analysis demonstrates that current efficient fine-tuning techniques fall short of addressing memory limitations, particularly in long-context scenarios.

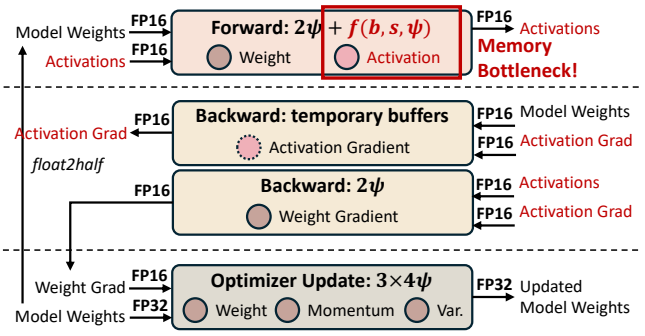


Figure 2: Memory breakdown during LLM mixed-precision fine-tuning. Compared to consistent model states, activations scale with both input batch size and sequence length, becoming the primary bottleneck in long-context scenarios.

Vanilla. The memory consumption of vanilla fine-tuning primarily consists of two parts: *model states* and *residual states*, as listed in Figure 2. The first part, *model states*, includes parameters, gradients, and optimizer states. In mixed-precision fine-tuning [47], parameters and gradients are stored in FP16, while optimizer states are stored in FP32. For modern optimizers like AdamW [46], the optimizer states include the parameters, momentum, and variance. Given model parameter size as ψ, the memory required for all model states is approximately 16ψ, which remains constant for a fixed model.

The second part of memory consumption comprises activations, temporary buffers, and fragmented memory. Among these, activations, which include the intermediate results and gradients stored during the forward pass, consume a considerable portion of memory. The memory required for activations is not only dependent on the model parameter size (ψ) but also scales with the input batch size (b) and sequence length (s). As detailed in Table 2, in long-context scenarios, the memory consumption of activations can easily exceed that of model states, emerging as the dominant memory bottleneck.

LoRA. Different from the vanilla, the pre-trained model parameters remain frozen in LoRA, and only the injected low-rank matrices are updated during fine-tuning. Consequently, the memory consumption for gradients and optimizer states

Table 2: Activation memory usage (compared to model states) of GPT-3 175B [8] across different sequence lengths.

Model States	$s = 4K$	$s = 8K$	$s = 16K$	$s = 32K$	$s = 64K$
$16 \times 175B = 2.8T$	937G	3.42T	13.0T	50.8T	201T
-	0.34×	1.22×	4.65×	18.14×	71.6×

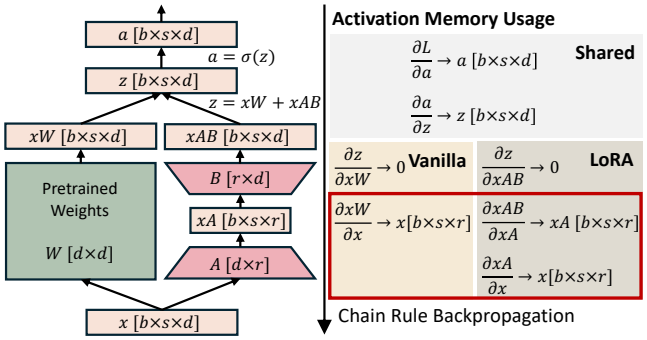


Figure 3: An illustrated example of LoRA for showcasing the activation memory usage of PEFT. Beyond the shared parts, LoRA requires storing more activations than vanilla.

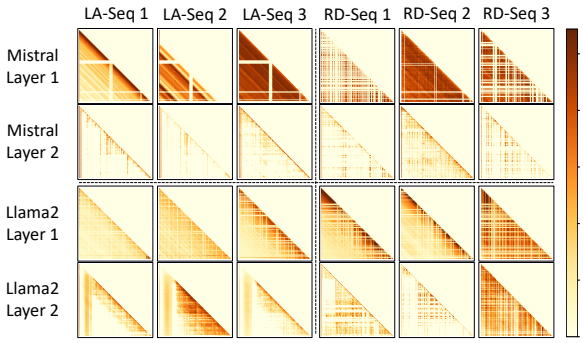


Figure 4: Visualization of attention scores across different models, datasets, layers, and sequences (darker is higher).

is significantly reduced. However, this reduction does not extend to activation memory, which emerges as the new memory bottleneck. As illustrated in Figure 3, LoRA not only fails to alleviate activation memory usage but increases it instead. This is because the trainable low-rank matrices are deeply embedded within the model structure. Gradient computation for these matrices requires a traversal nearly identical to the vanilla, following the chain rule in backpropagation. Similar issues also exist in other PEFT methods.

LongLoRA. Building upon LoRA, LongLoRA proposes a shifted sparse attention (S^2-Attn) mechanism for further efficiency. As depicted in Figure 1(b), S^2-Attn partitions the input tokens into two groups and performs attention individually in each group. For half of the attention heads, tokens are shifted by half group size to enable information exchange across groups. While LongLoRA achieves computational savings compared to LoRA, it fails to provide additional memory reduction. This limitation arises from the fact that the sparsity mechanism in LongLoRA, as well as other sparsity-based techniques, operates exclusively on the hidden dimensions of token embeddings. While such methods reduce the computational burden associated with individual tokens, they cannot entirely exclude tokens from the computation. However, once a token is involved in the computation, its activations are stored regardless of its usage. We refer to this phenomenon as ***Shadowy Activation***, analogous to how an object casts a shadow by blocking light. In this context, "blocking" refers to cases where a token is unnecessary for token 1 (meaning its activation could be discarded) but is needed by token 2 (requiring its activation to be preserved). The presence of shadowy activation leaves the activation bottleneck still unresolved.

3 Insight: Contextual Token Sparsity

To address the emerging challenges discussed in § 2.2, we introduce a fresh perspective that explores and exploits a token-level sparsity mechanism during LLM long-context fine-tuning. Our approach is grounded in the intuitive yet profound observation that natural language is inherently redundant [60, 71]. Specifically, as highlighted in several studies [38, 42, 61, 63, 72, 82], standard full attention can be effectively approximated by focusing on significant interactions among a limited set of query, key, and value values, involving only a subset of tokens in the sequence. Notably, this redundancy becomes even more pronounced in long-context scenarios. As presented in Table 3, the proportion of significant interactions in attention scores decreases as sequence length increases. These insights open up a compelling opportunity to optimize LLM long-context fine-tuning by identifying and retaining only the most informative tokens. Through directly reducing token involvement, shadowy activation constraints are naturally alleviated, enabling activation memory to achieve comparable benefits to computational savings.

We term this novel sparsity mechanism within LLM long-context fine-tuning as ***Contextual Token Sparsity***. Through comprehensive evaluations, we identify two key unique characteristics of this mechanism: (1) *Token-wise.* As depicted in Figure 4, a grid-like distribution of attention scores is observed across different models and datasets, confirming that token embeddings in the sequence exhibit varying levels of importance. This insight enables the exclusion of less valuable tokens, naturally leading to reductions in both memory usage and computational overhead. (2) *Contextual.* Figure 4 further demonstrates that the distribution of valuable tokens dynamically shifts based on input texts and varies across model layers, even for a given model and dataset. This dynamic nature underscores the need for a system capable of accurately

Table 3: Sparsity ratios (the proportion of values below 0.3 of the maximum) of attention scores across sequence lengths.

Seq len.	4K	6K	8K	10K	12K	14K	16K
Llama2	38.6%	48.8%	48.1%	65.3%	69.5%	62.2%	69.6%
Llama3	44.6%	46.0%	54.7%	52.3%	57.8%	50.2%	58.9%

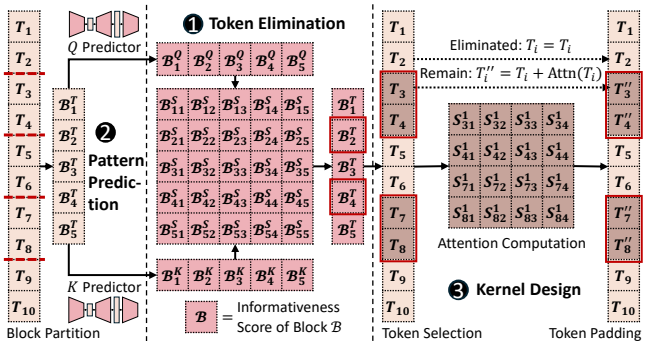


Figure 5: JENGA overview. At each layer, token embeddings are first partitioned into blocks and fed into the pattern predictors (❷). Using the predicted informativeness scores from these predictors, the token elimination algorithm (❶) effectively identifies and retains only the most informative tokens for processing. Optimized kernels (❸) then efficiently perform token selection, computation, residual addition, and padding.

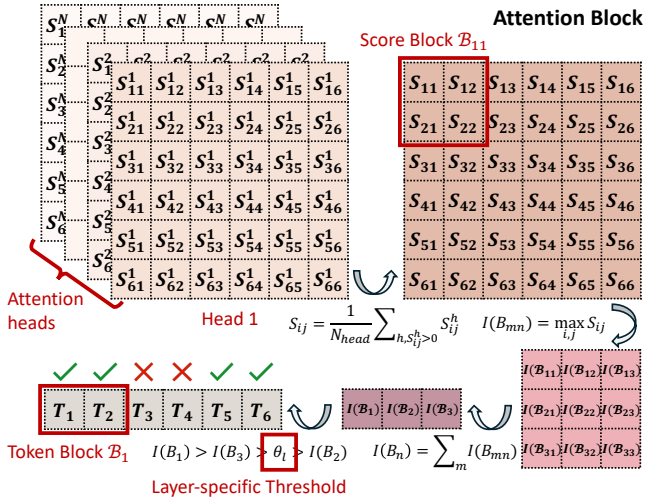


Figure 6: Token elimination algorithm. Attention scores are first aggregated across different heads and partitioned into multiple score blocks. The maximum value within each score block is defined as its informativeness score, which is then aggregated along the column. These resulting scores are compared against a layer-specific threshold to determine whether the corresponding tokens should be retained.

identifying and efficiently exploiting this sparsity in real-time during runtime. Serving as the cornerstone of our system, contextual token sparsity first introduces token-level sparsity into long-context fine-tuning, enabling LLMs to handle larger context windows while effectively involving fewer tokens.

4 System Design

4.1 Overview of JENGA

We propose JENGA, an efficient system designed to enhance LLM long-context fine-tuning by systematically exploring and exploiting contextual token sparsity. Figure 5 presents an overview of JENGA, which is built upon three key techniques:

Information-driven Token Elimination (§ 4.2). To determine whether a token is redundant, we first establish a formal definition for the informativeness of a given token. Building on this definition, JENGA utilizes a score-based algorithm that dynamically identifies and eliminates redundant tokens within the attention block. The algorithm performs in a block-wise manner and is further refined by adopting a layer-specific threshold, which ensures both effectiveness and efficiency. Additionally, we extend this approach to the MLP block, ensuring consistency across various components of the model.

Context-aware Pattern Prediction (§ 4.3). JENGA employs a neural-network-based approach to predict the token sparsity patterns, bypassing the need for costly full attention score computation. Once adequately trained, these predictors can accurately approximate the token informativeness based on contextual inputs. To minimize the overhead introduced by predictors, JENGA utilizes an elastic size transformation technique, optimizing both memory and computational efficiency.

High-performance Kernel Optimization (§ 4.4). JENGA delves into kernel-level optimizations to maximize system performance. To minimize unnecessary global memory movement, JENGA introduces a permutation-free strategy that fuses token selection, token padding, and residual addition directly into the computation pipeline. Moreover, JENGA incorporates a segment-based method to alleviate activation memory peaks, enabling efficient gradient computation for long sequences. By tightly coupling these kernel-level optimizations with algorithmic designs, JENGA fully exploits contextual token sparsity for enhancing LLM long-context fine-tuning.

4.2 Information-driven Token Elimination

To fully exploit contextual token sparsity, it is essential to accurately identify the redundant tokens across different inputs and layers. Discarding informative tokens may impact model accuracy while retaining excessive tokens leads to resource inefficiency. To tackle these challenges, JENGA proposes an information-driven algorithm that dynamically identifies and eliminates redundant tokens while preserving accuracy.

Token Informativeness. We begin by defining a token's informativeness based on its interactions with other tokens within the embedding space. In the attention mechanism, the attention score S_{attn} is commonly used to quantify the interaction between tokens [25, 42, 72]. Specifically, the attention score term S_{ij}, calculated as $Q_i K_j$, represents the interaction between token i and token j. Inspired by this, we define the informativeness of a token $I(T)$ by considering its interaction

with all other tokens in the long-context sequence:

$$I(T_j) = \sum_{i \neq j} S_{ij} = \sum_{i \neq j} Q_i K_j \tag{1}$$

where the sum aggregates the attention scores across all tokens i in the sequence, excluding the token j itself.

Block-wise Elimination. Building on the concept of token informativeness, the next step involves eliminating redundant tokens based on their informativeness scores. To optimize alignment with hardware characteristics, JENGA performs token elimination in a block-wise manner. As shown in Figure 6, JENGA partitions the attention scores along the token dimension into multiple *score blocks* $\mathcal{B}^S$. Following aggregation across attention heads, the maximum value within each block is selected as the informativeness score $I(\mathcal{B}^S)$. Notably, during aggregation, JENGA sums only the positive attention scores. Since attention scores undergo a softmax operation, negative values have a negligible impact on the final result but may offset the influence of positive values if included. By excluding negative scores, the aggregation process preserves the integrity of informativeness and ensures robust token elimination. The entire process can be formalized as:

$$I(\mathcal{B}^S_{mn}) = \max_{S_{ij} \in \mathcal{B}^S_{mn}} S_{ij} = \max_{S_{ij} \in \mathcal{B}^S_{mn}} \frac{\sum_{h, S^h_{ij} > 0} S^h_{ij}}{N_{\text{head}}} \tag{2}$$

where S^h_{ij} represents the attention score between token i and token j in attention head h, and N_{head} is the total number of attention heads, serving as the scaling factor. This block-wise token elimination preserves the original informativeness of long-context sequences for two main reasons. First, compared to long-context input sequences, the block size used in our approach is relatively small. Given the sparse distribution of important tokens across long-context sequences, most token blocks contain no important tokens and can be safely eliminated without compromising accuracy. Second, in cases where important tokens do fall within a block, our informativeness score for token blocks $I(\mathcal{B}^S)$ is designed to effectively identify and retain these blocks. This is because the difference in informativeness scores $I(T)$ for important versus unimportant tokens is large enough that important tokens cannot be averaged out by other unimportant tokens in the block. As a result, our method ensures that any block containing important tokens is preserved, regardless of how many such tokens it contains. We argue that maintaining fine-tuning accuracy is more critical than the negligible computational cost of retaining a few additional unimportant tokens.

Layer-specific Threshold. The informativeness scores of *token blocks* $\mathcal{B}^T$ are computed by aggregating the corresponding score blocks, i.e. $\mathcal{B}^T_n = \sum_m \mathcal{B}^S_{mn}$. These scores are then compared against a threshold to determine whether the tokens within the block should be eliminated. Particularly, JENGA further refines the token elimination algorithm by adopting a layer-specific threshold. The key insight is that different layers within LLMs exhibit varying sparsity patterns. As demonstrated in Figure 7, the average informativeness scores of token blocks vary greatly across different model layers, indicating that a universal threshold applied across all layers is suboptimal. Algorithm 1 outlines JENGA's approach, which initializes a default threshold for all layers based on score profiling and then fine-tunes these values to align with the unique sparsity characteristics of each layer.

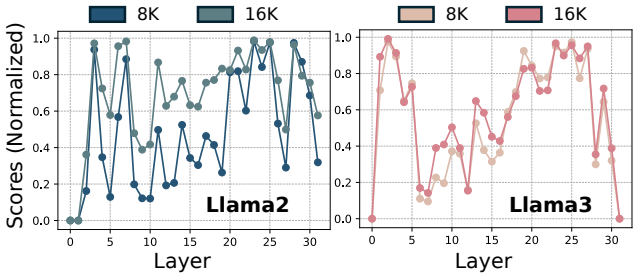


Figure 7: The average informativeness scores (normalized) of tokens blocks across different layers.

Extend to MLP Block. Additionally, we extend token elimination to the MLP block. Analogous to attention scores, JENGA utilizes intermediate activations within the MLP block to evaluate the informativeness of each token. This extension can be viewed as a variant of the widely studied neuron sparsity within MLP blocks [37, 41, 48], ensuring compatibility with various MLP block structures: for ReLU-based structure [2], the activations are the outputs of the ReLU layer, while for SiLU-based structure [20], the activations correspond to the element-wise multiplication of the gate projection (after SiLU) and the up projection. These techniques enable JENGA to adapt token elimination seamlessly across diverse model components and configurations.

Algorithm 1: Layer-Specific Threshold Optimization

Input : Model layers $\mathcal{L} = \{L_1, L_2, \ldots, L_n\}$
Output : Layer thresholds $\mathcal{T} = \{T_1, T_2, \ldots, T_n\}$

```
// Step 1: Threshold Initialization
```
foreach *layer* $L_i \in \mathcal{L}$ **do**
```
    // Average scores across token blocks
```
    $T_i \leftarrow \text{avg}\big(I(\mathcal{B}^T)\ \forall\ \mathcal{B}^T \in L_i\big)$;

```
// Step 2: Threshold Fine-Tuning
```
foreach *layer* $L_i \in \mathcal{L}$ **do**
```
    // Compute gradient with finite changes
```
    $G_i \leftarrow \big(\text{acc}(T_i + \varepsilon) - \text{acc}(T_i - \varepsilon)\big)/2\varepsilon$;
```
    // Update threshold based on gradient
```
    $T_i \leftarrow T_i + \eta \cdot G_i$;

return $\mathcal{T}$

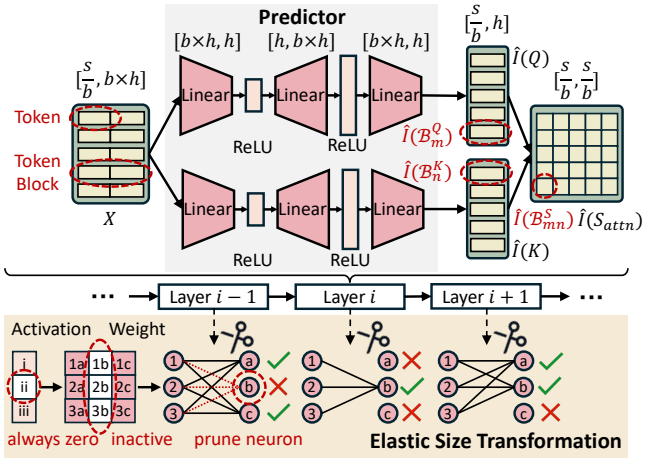


Figure 8: Pattern prediction process. Each layer is equipped with two predictors to approximate Q and K, respectively. Taking token embeddings as input (organized in token blocks), each predictor outputs the informative score, $\hat{I}(B_m^Q)$ and $\hat{I}(B_n^K)$, for each token blocks. These scores are then multiplied to compute the informative scores, $\hat{I}(B_{mn}^S)$, for blocked attention scores. Besides, elastic size transformation is employed to independently minimize the predictor size for each layer.

4.3 Context-aware Pattern Prediction

Although the exact sparsity patterns can be directly derived from the full attention scores, computing and storing these scores is prohibitively expensive, with complexity scaling quadratically with the sequence length. Furthermore, due to the dynamic nature of contextual token sparsity, the optimal sparsity patterns can only be determined at runtime, varying across different inputs and layers. To address these challenges, JENGA employs a set of lightweight neural networks as predictors. By taking contextual embeddings as inputs, these predictors infer sparsity patterns accurately and efficiently.

Neural-network-based Predictor. As illustrated in Figure 8, JENGA deploys a pair of predictors in each layer to approximate the informativeness scores of queries Q and keys K, respectively. Each predictor consists of three trainable low-rank matrices, with ReLU activation function applied between successive matrices. The inputs to the predictors are token embeddings X, which contain contextual information and are organized into blocks to align with the block-wise elimination. By extracting the representative embedding from each block, the predictors output the approximate informativeness scores, $\hat{I}(Q)$ and $\hat{I}(K)$. These scores are then multiplied to approximate the informativeness of attention scores $\hat{I}(S_{\text{attn}})$:

$$\hat{I}(S_{\text{attn}}) = \hat{I}(Q)\hat{I}(K)^T,\ \hat{I}(\mathcal{B}_{mn}^S) = \hat{I}(\mathcal{B}_m^Q)\hat{I}(\mathcal{B}_n^K)^T \qquad (3)$$

When Q and K predictors are well trained, $\hat{I}(S_{\text{attn}})$ can provide a close estimation of accurate informativeness scores $I(S_{\text{attn}})$.

With a limited training dataset, these predictors can quickly converge and perform well in prediction, as evidenced by prior studies [45, 62, 70]. Particularly, the predictor in JENGA first processes each token individually and then aggregates these individual predictions into a unified outcome. This strategy restricts the predictor's size to the dimension of a single token block rather than the full long-context sequence, thereby streamlining the design and mitigating prediction overhead.

Elastic Size Transformation. To further reduce the size of the predictors, JENGA utilizes an elastic size transformation technique that dynamically prunes neurons in the predictors based on their individual sparsity characteristics. Inspired by prior work on activation-based pruning studies [37, 41, 69], this design leverages the properties of the ReLU activation function, which introduces a large number of zero elements into the intermediate activation of the predictors. When an activation element is zero, its corresponding neurons (i.e., rows or columns of the model weights) become inactive and can be safely disregarded. Innovatively, JENGA integrates this mechanism into its custom-designed predictors. Specifically, JENGA tracks the zero frequency of intermediate activation elements during training and periodically prunes neurons associated with the highest zero frequencies. Without relying on any prior assumptions, elastic size transformation adaptively determines the optimal size for each predictor, effectively reducing both computational and memory overhead.

Comprehensive Overhead Analysis. We conclude with an analysis of the overhead introduced by predictors during both training and inference. In *offline training*, the primary bottleneck lies in obtaining the informativeness of attention scores $I(S_{\text{attn}})$. Thanks to the block-wise manner, we seamlessly integrate our custom training kernel into the state-of-the-art FlashAttention [15, 16]. This integration eliminates the need for explicit computation and storage of the full attention scores. Instead, $I(S_{\text{attn}})$ is derived online, leading to a memory complexity that grows linearly with sequence length.

In *online inference*, given sequence length s, head dimension h, and block size b, the computational overhead consists of two parts: (1) Prediction of $I(Q)$ or $I(K)$ with a complexity of $O(sh^2)$; (2) Prediction of $I(S_{\text{attn}})$ with a complexity of $O(s^2/b^2)$. In long-context scenarios, the second part becomes the dominant factor. However, this overhead can be effectively mitigated by increasing the block size b. On the memory side, the primary overhead arises from the linear weights within predictors, with a complexity of $O(bh^2)$. Importantly, this complexity remains constant relative to the model configurations. Finally, thanks to the elastic size transformation, both computational and memory complexities are reduced by a sparsity factor, leading to an average reduction of 50%.

4.4 High-performance Kernel Optimization

Focusing on token-level sparsity, JENGA introduces minimal modifications to the original fine-tuning dynamics, enabling seamless reuse of existing optimized computational flows.

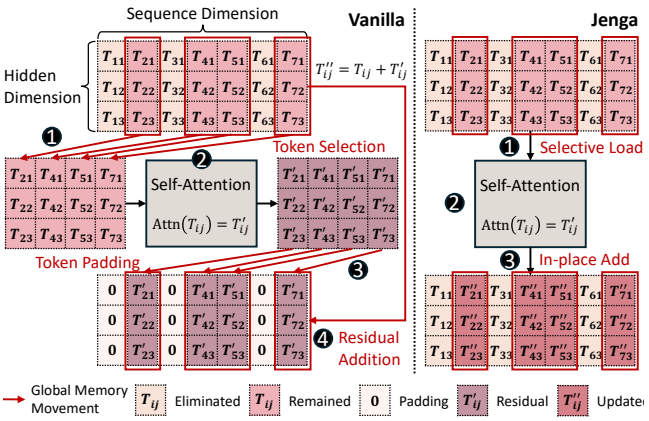


Figure 9: Comparison between naive token movement and JENGA. Highlighted by the red lines, the naive kernel incurs substantial global memory movement costs. JENGA develops a permutation-free strategy through kernel fusion.

However, two key challenges hidden in JENGA undermine its performance. First, the variability in sparsity patterns across layers necessitates iterative token selection and padding, leading to considerable costly global memory movement. Second, the extensive vocabulary size of LLMs necessitates substantial activation memory to compute the output loss gradient for each token, particularly in long-context scenarios. JENGA incorporates several hardware-efficient techniques at the kernel level, effectively mitigating these bottlenecks.

Permutation-free Token Movement. The dynamic nature of contextual token sparsity leads to varying sparsity patterns across different layers, involving distinct subsets of tokens. As illustrated in Figure 9, at each layer, a set of less informative tokens are eliminated and the remained tokens are re-permuted to serve as the attention block inputs. Then, the attention outputs are padded with zeros to maintain dimensional consistency and are finally residually added to the original inputs. The processes of token selection, token padding, and residual addition involve extensive data movement in global memory, which incurs high memory access latency.

JENGA develops a permutation-free strategy that fuses all unnecessary permutation operations with the attention computations. Instead of materializing re-permuted tokens, JENGA directly loads the selected tokens from original inputs. Furthermore, JENGA performs an in-place addition of attention outputs to the original inputs, simultaneously completing token padding and residual addition in a single step. During backpropagation, the original inputs are recomputed when calculating the gradient of the attention weights. Since the input embedding matrix can be recovered by subtracting the self-attention output from the output embedding matrix, the overhead is minimal. This streamlined approach eliminates unnecessary memory allocation and expensive global memory movement, significantly enhancing system performance.

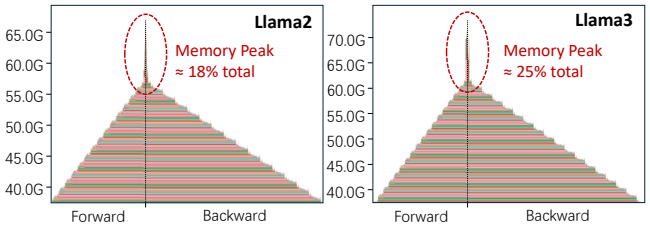


Figure 10: Memory peak during loss gradient computation, exacerbated by the large vocabulary size and long context. The x-axis represents the timeline of a fine-tuning epoch.

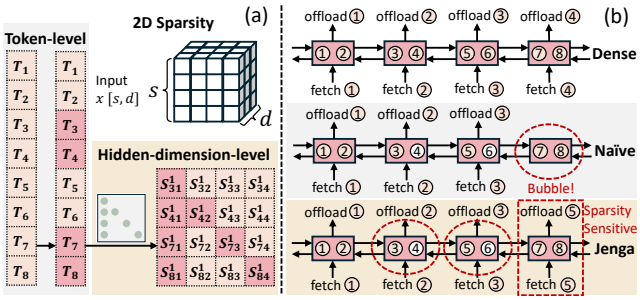


Figure 11: Two available extensions of JENGA: (a) Two-dimensional Sparsity and (b) Sparsity-sensitive Offload.

Segment-based Peak Cutting. LLMs are typically autoregressive, predicting the probability distribution of the next token given all preceding tokens. During fine-tuning, for each token in the input sequence, the model generates a probability distribution over the next token and computes the losses between predictions and ground truth. For LLMs with a large vocabulary size and long context window, this process imposes a significant surge in activation memory usage, as shown in Figure 10. While these activations are transient, the resulting memory peak elevates the upper bound of LLM fine-tuning, imposing stricter demands on GPU memory resources. Existing implementations [64, 65] commonly use column-wise parallelism to distribute the vocabulary across multiple GPUs. However, this optimization is only effective in multi-GPU configurations and provides no benefit for single-GPU setups.

To address this issue, JENGA adopts a segment-based peak-cutting strategy that partitions the token sequence into smaller, manageable segments during final loss computation. Instead of performing a forward pass over the entire sequence and retaining all intermediate activations, JENGA processes each segment independently and later aggregates their gradients. Activations for each segment are discarded immediately after the corresponding gradient computation is completed. Consequently, the activation memory peak is reduced to $1/N$ when the sequence is divided into N segments. This approach greatly alleviates memory pressure on individual GPUs and remains compatible with existing multi-GPU optimizations.

Table 4: Hardware platform configurations.

Platform	GPUs	Memory	FP32 TFLOPS	BF16 TFLOPS
Platform A	1× A800	80GB	19.5	312
Platform B	1× A40	48GB	37.4	150
Platform C	4× 4090	24GB	82.6	82.6

Table 5: Model configurations.

Model	# Params.	Def Len.	Seq Len.
OPT [81]	350M/1.3B/2.7B/6.7B	2K	2K-64K
Llama2 [67]	7B	4K	4K-64K
Llama3 [19]	8B	8K	4K-64K

5 Implementation and Extension

We implement JENGA with over 3000 lines of Python and C++ code. Given the minimal changes to the original fine-tuning dynamics, JENGA is compatible with a wide range of LLM architectures without requiring any code changes. Besides, JENGA can seamlessly integrate with other techniques:

Extension 1: Two-dimensional Sparsity. As illustrated in Figure 11(a), after applying token-level sparsity in JENGA, the remained tokens can further benefit from existing hidden-dimensional sparsity techniques. This natural combination of sparsity mechanisms across two dimensions, which we term *2D-Sparsity*, provides more granular control over the model's resource allocation, leading to a significant reduction in both activation memory and computational costs.

Extension 2: Sparsity-sensitive Offload. JENGA enhances existing offload-based techniques [27, 30, 55, 56] by incorporating contextual token sparsity into the optimization process. As depicted in Figure 11(b), we develop a sparsity-sensitive offloading strategy that adapts to the varying sparsity ratios across different layers. This approach enables the seamless transfer of larger data volumes between the CPU and GPU, effectively alleviating GPU memory constraints.

6 Evaluation

6.1 Experimental Setup

Hardware. We conduct experiments on three representative platforms, as listed in Table 4, covering both data-center workstations and desktop professional GPUs. Memory measurements are primarily evaluated on Platform A, as it offers the largest GPU memory capacity, with results being largely insensitive to GPU arithmetic performance. For speedup evaluations, we adhere to common practices by employing mixed-precision techniques [47], utilizing both BF16 and FP32.

Models. The models used for evaluation are detailed in Table 5. We choose models from two of the most popular LLM families: OPT and Llama. These models vary in architecture, parameter size, and default context window size. Evaluations on these models provide a robust demonstration of the scalability and versatility of JENGA's optimizations.

Dataset. We use the RedPajama [14] dataset for long-context fine-tuning, following the setup from LongLoRA. We evaluate the perplexity (PPL) of fine-tuned models on the book corpus dataset PG19 [57] and the cleaned Arxiv Math proof-pile dataset [3] to assess long-context modeling performance. Besides, we evaluate our method on LongBench [5] benchmark, following an instruction-tuning on dataset LongAlign-10k [4]. These tasks cover multiple critical long-text application areas, ensuring a comprehensive evaluation of our approach.

Baselines. We compare JENGA with two state-of-the-art fine-tuning methods, LoRA [29] and LongLoRA [12]. These methods represent the two dominant optimization directions: parameter-efficient fine-tuning and sparsity-based fine-tuning, respectively. For speedup analysis, we primarily compare with LoRA, leaving LongLoRA for reference. This distinction is because the two methods focus on orthogonal sparsity dimensions, making a direct and fair comparison challenging.

Metrics. Memory evaluations are conducted at the time step immediately following the forward pass, which generally corresponds to the memory peak. For execution time evaluation, we measure the time per fine-tuning step. To emphasize the effectiveness of JENGA, activation recomputation and offloading techniques are excluded unless explicitly stated. All metrics are averaged over 10 repeated trials to ensure reliability.

6.2 End-to-End Performance

Memory Footprint. We evaluate the memory efficiency of JENGA across various models and sequence lengths, as shown in Figure 12. The results reveal that JENGA achieves average memory savings of 38.2% and 50.5% compared to LoRA across six different models, with sequence lengths of 4K and 8K, respectively. Similar benefits are observed in comparison to LongLoRA, as the presence of shadowy activations renders its sparsity mechanism ineffective in benefiting memory usage (even slightly increased). Furthermore, the results reveal that for a fixed model, JENGA's memory efficiency improves as sequence length increases. This aligns with the observation that longer text sequences typically exhibit greater redundancy. The enhanced efficiency of JENGA largely extends the fine-tuning sequence length achievable under GPU memory constraints. Without activation recomputation and offloading, both LoRA and LongLoRA are limited to a sequence length of 16K (32K) when fine-tuning OPT 1.3B (350M). Instead, JENGA doubles this capacity, supporting sequence lengths of up to 32K (64K) on a single A800 GPU.

Execution Time. The minimized token involvement in JENGA also brings computational savings during long-context fine-tuning. Figure 13 presents the execution time and corresponding speedups of JENGA during fine-tuning different models at a sequence length of 4K. The results show that

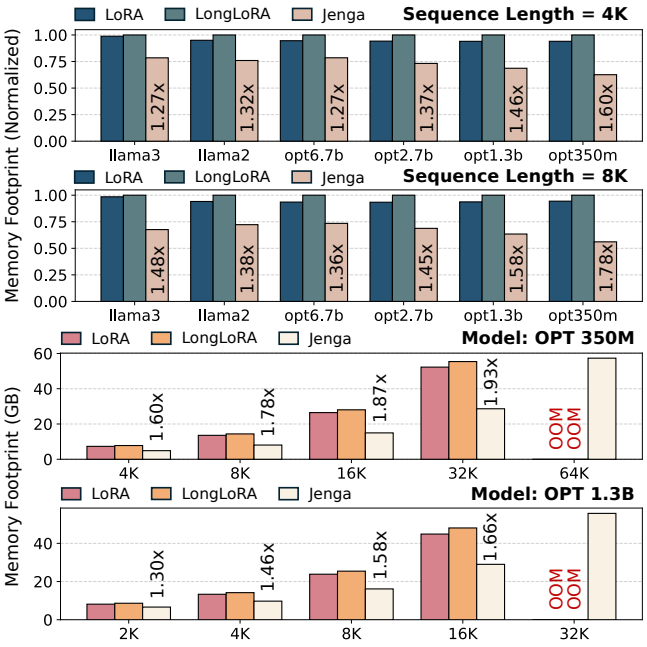


Figure 12: Memory footprints comparison on A800.

JENGA achieves computational efficiency comparable to LongLoRA, achieving an average speedup over LoRA of 10.8% and 8.6% on two platforms, respectively. We also observe that LongLoRA may perform slower than LoRA in some cases. This is primarily due to the difficulty in fully utilizing hardware computational capacity for sparsity operations when the sequence length is insufficient. In contrast, JENGA introduces minimal modifications to the original computational flow, allowing it to effectively translate computational savings into practical speedups. Further evaluations of JENGA on longer sequence lengths (with recomputation) reveal additional performance gains, achieving up to 1.36× speedups.

Accuracy Evaluation. We test the impact of JENGA on model accuracy by comparing it with the original LoRA. First, we measure test perplexity of fine-tuned Llama2 7B on two representative long-context datasets, PG19 and Proof-Pile. As shown in Table 7, JENGA incurs only a minimal increase in perplexity scores compared to the original LoRA, across varying sequence lengths. Additionally, we evaluate JENGA on LongBench benchmark, which contains tasks from multiple key long-text application areas. Table 6 demonstrates that JENGA achieves accuracy comparable to the original LoRA. These evaluations collectively confirm that the inherent redundancy in long-context sequences can be effectively exploited for performance efficiency without compromising accuracy.

6.3 Ablation Study

Fine-grained Performance Breakdown Figure 14 presents a detailed performance breakdown of JENGA, covering both

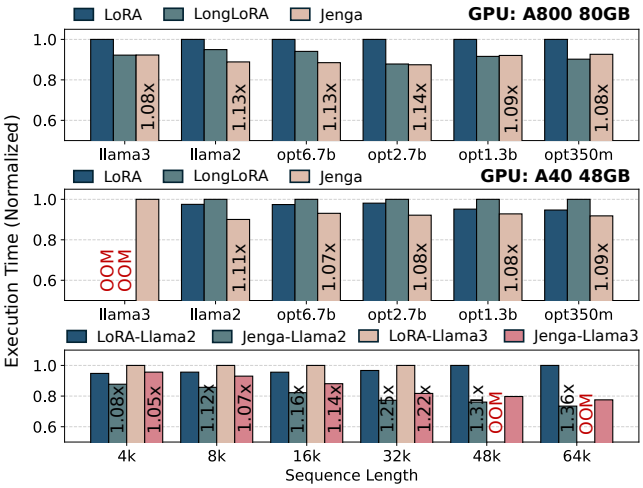


Figure 13: End-to-end speedup of JENGA on A800 and A40.

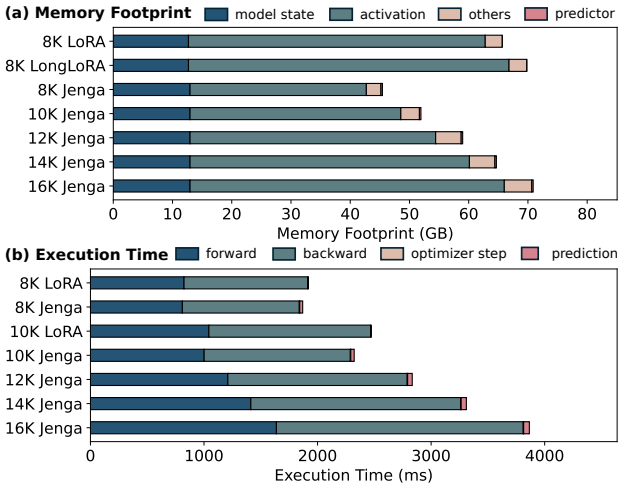


Figure 14: Performance breakdown of Llama2 fine-tuning: (a) Memory footprint and (b) Execution time.

memory and computational aspects. For the memory aspect, the results show that JENGA effectively reduces activation memory consumption compared to both LoRA and LongLoRA. Although the predictors introduced incur additional memory usage, their overhead is minimal, ensuring that the overall memory reduction is preserved. Besides, this reduction is consistent across varying sequence lengths, with the decrease in activation memory scaling linearly with sequence length. For the computational aspect, JENGA also achieves computational gains over LoRA, as the reduced token involvement leads to decreased computation during both the forward and backward phases. Similarly, the computational overhead of predictors is negligible in the context of the overall process.

Technique 1: Token Elimination. We delve into the layer level to analyze the effectiveness of our information-driven

Table 6: Comparative analysis of model accuracy on the LongBench benchmark (higher is better).

Tasks	mfqa_zh	mfqa_en	gov_report	triviaqa	vcsum	qmsum	musique	2wikimqa	repobench
Origin	23.45	23.22	**27.44**	**84.60**	**13.30**	**22.64**	4.63	9.01	**52.00**
Ours	**23.53**	**24.74**	25.92	82.59	13.02	20.33	**5.73**	**10.14**	48.32
Tasks	qasper	hotpotqa	multi_news	pr_zh	pr_en	trec	lsht	dureader	lcc
Origin	15.94	9.40	**24.43**	**10.0**	20.0	**68.0**	21.0	**23.69**	**71.28**
Ours	**17.68**	**9.55**	22.53	8.00	**22.0**	**68.0**	**25.0**	21.37	70.32

Table 7: Perplexity of PG19 (PG) and Proof-Pile (PP) datasets.

Seq len.	8K	10K	12K	14K	16k
PG-Origin	6.95	6.92	6.91	6.90	6.87
PG-Ours	7.11	7.12	7.08	7.13	7.08
PP-Origin	2.68	2.64	2.59	2.59	2.57
PP-Ours	2.79	2.77	2.72	2.72	2.70

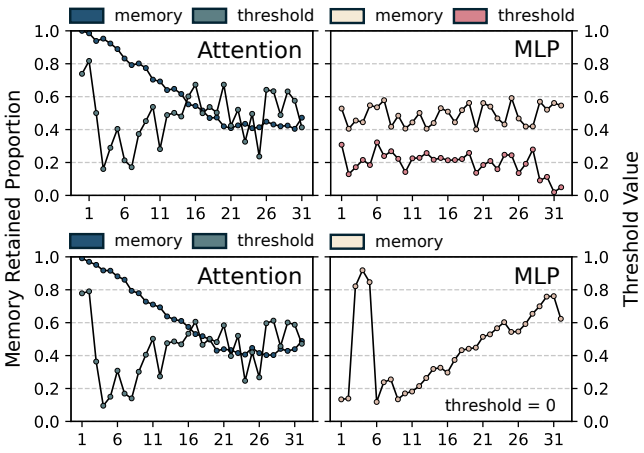


Figure 15: Memory footprint and corresponding threshold across layers on Llama2 7B (upper) and OPT 6.7B (lower).

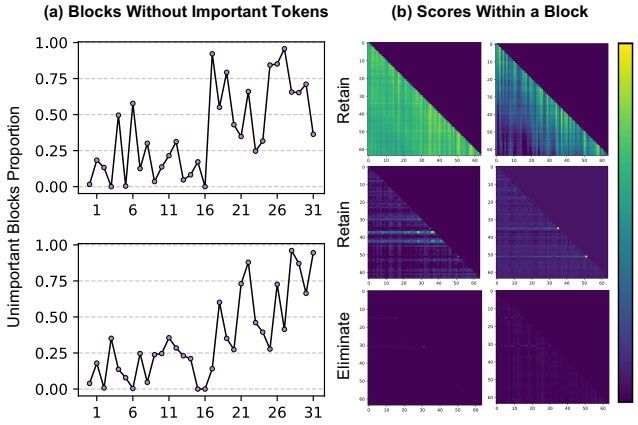


Figure 16: (a) Proportion of blocks without important tokens and (b) Distribution of attention scores within a token block.

token elimination algorithm. Figure 15 presents the memory consumption and corresponding threshold values across different layers in both self-attention and MLP blocks. Evaluations are conducted on both Llama2 and OPT models, considering their distinct MLP block architecture, which respectively uses SiLU and ReLU as activation functions. The results indicate that the token elimination algorithm achieves average memory savings of 38.3% (38.0%) on Attention block and 51.1% (54.8%) on MLP block for Llama2 (OPT) model. Besides, the application of layer-specific thresholds allows for varying degrees of reduction across layers, maximizing the exploitation of token-level sparsity while preserving model accuracy.

Additionally, we investigate the fundamental reason behind the effectiveness of our information-driven token elimination algorithm. Figure 16(a) illustrates the proportion of unimportant blocks across different layers. A block is considered unimportant if its informativeness score is less than 10% of the maximum (excluding outliers) within the same layer. The results reveal that a large proportion of token blocks do not contain any important tokens, especially in the deeper layers of the model. This can be attributed to the relatively small block size used (e.g., 64 in our evaluation settings), compared to the length of long-context input sequences. As the model depth increases, important tokens become more sparsely distributed, resulting in most token blocks containing no important tokens. These blocks can be safely eliminated without compromising model accuracy.

Moreover, we visualize the attention scores within individual blocks to analyze how the tokens within a token block affect its informativeness. Figure 16(b) presents three representative distributions of attention scores within token blocks. The results show that token blocks are generally classified as important when the majority of their tokens are informative, and as unimportant when none are. Particularly, even when only a few tokens in a block are important, the block is still considered as important. This is because the difference in informativeness scores for important versus unimportant tokens is large enough that important tokens cannot be averaged out under our block selection strategy.

Technique 2: Pattern Prediction. Figure 17(a) presents the training loss of predictors across two models and two datasets.

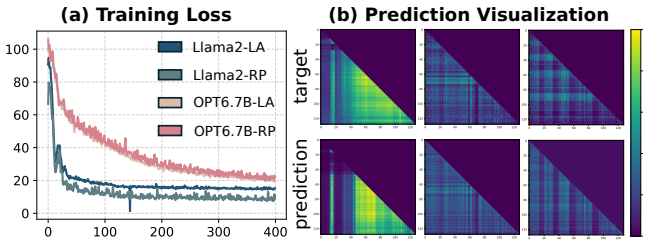


Figure 17: (a) Training loss curve on LongAlign (LA) / RedPajama (RP) and (b) Prediction visualizations of predictors.

The results show that the predictors converge quickly during offline training, requiring fewer than 400 epochs, an acceptable overhead given the following expensive LLM fine-tuning. We then calculate the recall metrics for evaluating the accuracy of predictors, achieving an impressive average of 95.13%. Particularly, we provide visual comparisons of the predictions against the ground truth. As depicted in Figure 17(b), the predicted attention scores closely approximate the ground truth, effectively identifying redundant tokens with high accuracy.

To assess the effectiveness of elastic size transformation, we measure the parameter sizes of predictors across various model layers. Table 8 details that by exploiting the inherent sparsity within predictors, their parameter sizes can be uniformly reduced across layers, on average 64.6%. This reduction in predictor size minimizes prediction overhead, aligning consistently with the findings in the performance breakdown.

Technique 3: Kernel Optimization. We benchmark the performance of kernels optimized with permutation-free token movement against their naive implementation under various sequence lengths, as depicted in Figure 18. The results reveal that both kernel fusion strategies, selective load and in-place addition, effectively enhance performance. The overall speedups increase with sequence length, ranging from 10× to over 50×. This improvement primarily stems from the reduction in global memory movement and temporary data allocation. These findings highlight the critical importance of high-performance kernel design, which serves as a robust foundation for the algorithmic framework of JENGA.

Meanwhile, Figure 19 illustrates the memory consumption during fine-tuning with and without the segment-based peak-cutting technique. In the naive implementation, gradient computation requires about 10GB of temporary activation memory, resulting in an inefficient memory peak. By partitioning the gradient computation into smaller segments, the sharp peak is divided into multiple, much smaller memory peaks, achieving an additional 15% of memory savings.

Table 8: Parameter size (in millions) of predictors across layers of Llama2, with or without elastic size transformation.

# Layer	1	5	10	15	20	25	30
Origin	12.58	12.58	12.58	12.58	12.58	12.58	12.58
Pruned	8.19	8.19	8.13	8.10	8.08	8.07	8.10
Pct.	65.1%	65.1%	64.6%	64.5%	64.2%	64.1%	64.5%

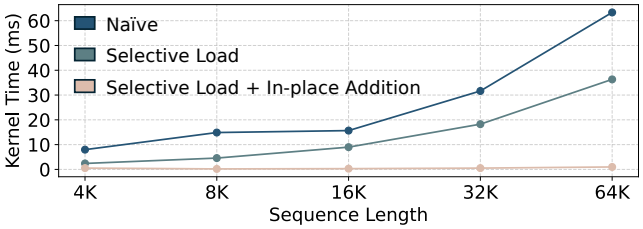


Figure 18: Performance of JENGA's permutation-free kernel.

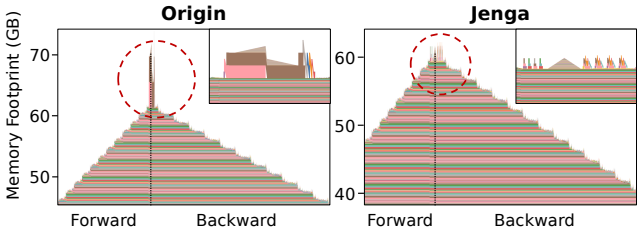


Figure 19: Memory usage peak in loss gradient computation. The x-axis represents the timeline of a fine-tuning epoch.

6.4 Extension Evaluation

Extension 1: Two-dimensional Sparsity. Building upon JENGA, we explore applying existing hidden-dimension-level sparsity techniques [70] to the remaining tokens during attention computation. Figure 20(a) shows that this two-dimensional sparsity further improves the computational efficiency of JENGA, achieving up to 2.04× speedups on Llama2.

Extension 2: Sparsity-sensitive Offload. We compare the performance of our sparsity-sensitive offloading against a naive uniform offloading strategy. Unlike the naive approach, JENGA considers the varying sparsity ratios across layers, allowing for the offloading of more activations or a reduction in data transfer latency. Figure 20(b) shows that this technique achieves an average speedup of 1.22× speedups on Llama2.

6.5 Scalability Analysis

We conclude by analyzing the strong scalability of JENGA on 4×4090 GPUs. Figure 21 shows that JENGA's performance scales proportionally with GPU number across different models and sequence lengths. The scalability is achieved because JENGA seamlessly minimizes token involvements and introduces no extra communication overhead. These results highlight JENGA's potential for deployment in large-scale systems.

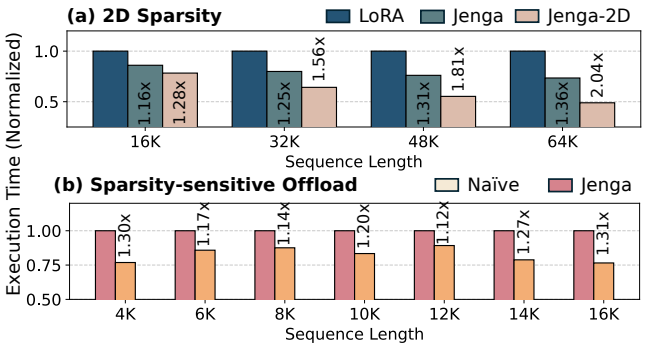


Figure 20: Performance improvements from two extensions.

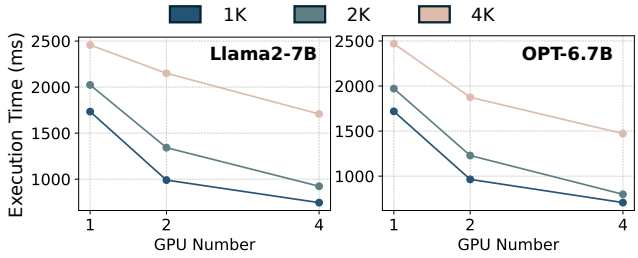


Figure 21: Strong scalability evaluation of JENGA.

7 Related Work

Optimizations for Activation Memory. As a primary memory bottleneck in LLM training or fine-tuning, activation memory consumption has been the focus of extensive research, which can be categorized into three main approaches: The first is *activation recomputation* [11,35,36], designed to avoid storing activations during the forward pass but recomputing them during the backward pass. The second is *activation offloading* [27,30,55,56], which asynchronously transfers activations from GPU to CPU and prefetches them back before required. The last is *activation compression* [9,21,44], which reduces the activation memory size through quantization or pruning. However, these methods primarily trade memory for additional computation or communication, rather than fundamentally reducing memory demands. In contrast, JENGA directly minimizes activation memory requirements and can be seamlessly combined with all these optimizations.

Optimizations for Long-context Fine-tuning. To effectively extend the context window to longer sequences, some methods [10,18,43] focus on optimizing the fine-tuning algorithm design. Besides, some methods [59,68,74] explore strategies for modifying the position embeddings of LLMs to handle longer context. All these efforts are complementary to JENGA.

Beyond effectiveness, some recent methods [80,84] are proposed to mitigate the substantial fine-tuning overheads for efficiency. Particularly, Parameter-efficient fine-tuning methods [28,29,39,78] first offer an effective solution by reducing the number of trainable parameters and memory usage while achieving comparable performance to full fine-tuning. Furthermore, a series of sparsity-based methods [12,70] are proposed to further reduce the computation costs by exploiting the inherent sparsity within attention mechanism [17]. However, while PEFT methods greatly cut down the memory consumption of optimizer states, the activation memory emerges as the primary bottleneck. Although existing sparsity mechanisms deliver notable computational gains, the presence of shadowy activation prevents comparable benefits for activation memory. Instead, JENGA achieves the best of both worlds by identifying and exploiting contextual token sparsity.

Optimizations for Token Utilization. Sharing the same high-level idea of JENGA, several studies also explore leveraging the inherent redundancy that existed in natural language, including data engineering [22,23], prompt compression [32,40], and inference optimization [38,61,63,82]. Notably, some works [7,25,33,34,49,58,75] propose eliminating tokens during inference to reduce model latency. Compared to existing token-elimination methods, JENGA offers two key innovations. First, prior works focus exclusively on small-model inference, where only a single request is processed at a time and there is no interference between the sparsity patterns of different tokens. JENGA is the first to extend attention-based token elimination to LLM long-context fine-tuning. Second, prior works typically prune tokens permanently throughout the computation of an entire epoch. While this strategy may be acceptable for inference with small models, it is not suitable for fine-tuning LLMs since the importance of a token can vary across different layers. To address this, JENGA proposes a novel sparsity mechanism-Contextual Token Sparsity.

8 Conclusion

We propose JENGA, an efficient system designed to optimize long-context fine-tuning for LLMs. Our approach introduces a novel sparsity mechanism within LLM long-context fine-tuning, termed contextual token sparsity. To systematically exploit this mechanism, we develop three key techniques that identify, predict, and exploit this sparsity, achieving both memory savings and performance speedups over state-of-the-art methods. Compression embodies intelligence, with sparsity serving as a potent form of compression. We envision JENGA inspiring broader exploration of sparsity for advancing LLMs.

9 Acknowledgement

We sincerely thank our shepherd Rohan Kadekodi and anonymous reviewers for their insightful feedback. The work is supported in part by the National Natural Science Foundation of China under Grant No. 62432004, and by a grant from the Guoqiang Institute, Tsinghua University.

A Artifact Appendix

Abstract

This artifact accompanies the ATC'25 paper titled JENGA*: Enhancing Long-Context Fine-tuning of LLMs with Contextual Token Sparsity*. It includes all necessary code, scripts, datasets, and pre-trained weights required to fully reproduce the experimental results, including figures and tables. The artifact also provides helper scripts for setup verification and quick result demonstration, and it supports both quick and full-scale experiment reproduction workflows.

Scope

This artifact allows users to validate all empirical claims made in the paper, including memory and runtime performance, prediction accuracy, and scalability of the proposed JENGA method on long-context fine-tuning tasks. It can be used to inspect all end-to-end results, perform ablation studies, and test generalization across models and datasets.

Contents

The artifact contains the following components:

- **Model and Predictor Weights (`checkpoints/`)**: Pre-trained model and predictor weights.
- **Dataset (`dataset/`)**: All datasets used in the experiments.
- **Log Files (`logs/`)**: Raw logs for generating paper figures.
- **Output Figures (`output_figures/`)**: Plots reproduced from logs.
- **Experiment Scripts (`scripts/`)**: Scripts to reproduce each figure/table in the paper.
- **Source Code (`src/`)**: Core implementation of JENGA.

Hosting

The artifact is hosted at a public Github repository. All resources, including models and datasets, can be obtained from:

- **Artifact Repository:** https://github.com/Pairshoe/Jenga-AE.
- **Model Weights:** Models Link (peft_model.zip and predictor.zip)
- **Hugging Face Models:** Detailed in Table 9.
- **Datasets:** Datasets Link (dataset.zip)

Table 9: Download links for required base models and their respective target directories.

Destination Path	Source URL
`checkpoints/llama3`	meta-llama/Meta-Llama-3-8B
`checkpoints/llama2`	meta-llama/Llama-2-7b-hf
`checkpoints/opt-6.7b`	facebook/opt-6.7b
`checkpoints/opt-2.7b`	facebook/opt-2.7b
`checkpoints/opt-1.3b`	facebook/opt-1.3b
`checkpoints/opt-350m`	facebook/opt-350m
`checkpoints/opt-125m`	facebook/opt-125m

Requirements

- **Software:** Python 3.10, PyTorch, Flash Attention (install separately).
- **Hardware (for full experiments):**
  - **Minimum:** 1 NVIDIA A800 or A40 GPU.
  - **For scalability experiments:** 4 NVIDIA 4090 GPUs.

A.1 Installation

1. Install dependencies:
   ```
   pip install -r requirements.txt
   ```
2. Install Flash Attention:
   ```
   pip install flash-attn -no-build-isolation
   ```
3. Install Jenga from source:
   ```
   pip install -e .
   ```

A.2 Experiment Workflow

1. Environment Setup Verification

Run the script to verify correct setup:
```
bash hello_world.sh
```

2. Quick Reproduction

Run the script to plot figures from raw data:
```
bash RUNME-a.sh
```

3. In-depth Reproduction

3.1 One-step Execution: Reproduce all results in a single run (runtime around 5 hours).
```
bash RUNME-b-a800.sh # for A800
bash RUNME-b-a40.sh # for A40
bash RUNME-b-4x4090.sh # for 4x4090
```

3.2 Separate Execution: For fine-grained control, individual scripts per experiment are summarized in Table 10.

Table 10: Generated output folders and corresponding scripts for reproducing results.

Generated Output Folder Name	Figure/Table	Script Path	Runtime
output_figures/end2end/memory	Figure 12	scripts/end2end-memory/run.sh	40 mins
output_figures/end2end/time-a800	Figure 13	scripts/end2end-time/run.sh (A800)	50 mins
output_figures/ablations/memory-breakdown	Figure 14 (Upper)	scripts/ablation-mem-breakdown/run.sh	10 mins
output_figures/ablations/time-breakdown	Figure 14 (Lower)	scripts/ablation-time-breakdown/run.sh	10 mins
output_figures/ablations/algorithm	Figure 15	scripts/ablation-algorithm/run.sh	5 mins
output_figures/ablations/predictor	Figure 16 (Left)	scripts/ablation-predictor/run.sh	3 hours
output_figures/ablations/segment	Figure 18 (memory_viz)	scripts/ablation-segment/run.sh	5 mins
output_figures/extension/2d	Figure 19 (Upper)	scripts/extension-2d/run.sh	25 mins
output_figures/extension/offload	Figure 19 (Lower)	scripts/extension-offload/run.sh	30 mins
output_figures/scalability	Figure 20	scripts/scalability/run.sh (4x4090)	30 mins
logs/end2end/accuracy/longbench	Table 6	scripts/end2end-longbench/run.sh	3 hours
logs/end2end/accuracy	Table 7	scripts/end2end-ppl/ppl.sh	6 hours

References

[1] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

[2] AF Agarap. Deep learning using rectified linear units (relu). *arXiv preprint arXiv:1803.08375*, 2018.

[3] Zhangir Azerbayev, Hailey Schoelkopf, Keiran Paster, Marco Dos Santos, Stephen McAleer, Albert Q. Jiang, Jia Deng, Stella Biderman, and Sean Welleck. Llemma: An open language model for mathematics, 2023.

[4] Yushi Bai, Xin Lv, Jiajie Zhang, Yuze He, Ji Qi, Lei Hou, Jie Tang, Yuxiao Dong, and Juanzi Li. LongAlign: A recipe for long context alignment of large language models. In *Findings of the Association for Computational Linguistics: EMNLP 2024*, pages 1376–1395, Miami, Florida, USA, November 2024. Association for Computational Linguistics.

[5] Yushi Bai, Xin Lv, Jiajie Zhang, Hongchang Lyu, Jiankai Tang, Zhidian Huang, Zhengxiao Du, Xiao Liu, Aohan Zeng, Lei Hou, et al. Longbench: A bilingual, multitask benchmark for long context understanding. *arXiv preprint arXiv:2308.14508*, 2023.

[6] Iz Beltagy, Matthew E Peters, and Arman Cohan. Longformer: The long-document transformer. *arXiv preprint arXiv:2004.05150*, 2020.

[7] Daniel Bolya, Cheng-Yang Fu, Xiaoliang Dai, Peizhao Zhang, Christoph Feichtenhofer, and Judy Hoffman. Token merging: Your vit but faster. *arXiv preprint arXiv:2210.09461*, 2022.

[8] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[9] Jianfei Chen, Lianmin Zheng, Zhewei Yao, Dequan Wang, Ion Stoica, Michael Mahoney, and Joseph Gonzalez. Actnn: Reducing training memory footprint via 2-bit activation compressed training. In *International Conference on Machine Learning*, pages 1803–1813. PMLR, 2021.

[10] Shouyuan Chen, Sherman Wong, Liangjian Chen, and Yuandong Tian. Extending context window of large language models via positional interpolation. *arXiv preprint arXiv:2306.15595*, 2023.

[11] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. Training deep nets with sublinear memory cost. *arXiv preprint arXiv:1604.06174*, 2016.

[12] Yukang Chen, Shengju Qian, Haotian Tang, Xin Lai, Zhijian Liu, Song Han, and Jiaya Jia. Longlora: Efficient fine-tuning of long-context large language models. *arXiv preprint arXiv:2309.12307*, 2023.

[13] Robert Chew, John Bollenbacher, Michael Wenger, Jessica Speer, and Annice Kim. Llm-assisted content analysis: Using large language models to support deductive coding. *arXiv preprint arXiv:2306.14924*, 2023.

[14] Together Computer. Redpajama: an open dataset for training large language models, 2023.

[15] Tri Dao. FlashAttention-2: Faster attention with better parallelism and work partitioning. In *International Conference on Learning Representations (ICLR)*, 2024.

[16] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. FlashAttention: Fast and memory-efficient exact attention with IO-awareness. In *Advances*

in Neural Information Processing Systems (NeurIPS), 2022.

[17] Yichuan Deng, Zhao Song, and Chiwun Yang. Attention is naturally sparse with gaussian distributed input. *arXiv preprint arXiv:2404.02690*, 2024.

[18] Yiran Ding, Li Lyna Zhang, Chengruidong Zhang, Yuanyuan Xu, Ning Shang, Jiahang Xu, Fan Yang, and Mao Yang. Longrope: Extending llm context window beyond 2 million tokens. *arXiv preprint arXiv:2402.13753*, 2024.

[19] Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

[20] Stefan Elfwing, Eiji Uchibe, and Kenji Doya. Sigmoid-weighted linear units for neural network function approximation in reinforcement learning. *Neural networks*, 107:3–11, 2018.

[21] R David Evans and Tor Aamodt. Ac-gc: Lossy activation compression with guaranteed convergence. *Advances in Neural Information Processing Systems*, 34:27434–27448, 2021.

[22] Yao Fu, Rameswar Panda, Xinyao Niu, Xiang Yue, Hannaneh Hajishirzi, Yoon Kim, and Hao Peng. Data engineering for scaling language models to 128k context. *arXiv preprint arXiv:2402.10171*, 2024.

[23] Tianyu Gao, Alexander Wettig, Luxi He, Yihe Dong, Sadhika Malladi, and Danqi Chen. Metadata conditioning accelerates language model pre-training. *arXiv preprint arXiv:2501.01956*, 2025.

[24] Yizhao Gao, Zhichen Zeng, Dayou Du, Shijie Cao, Hayden Kwok-Hay So, Ting Cao, Fan Yang, and Mao Yang. Seerattention: Learning intrinsic sparse attention in your llms. *arXiv preprint arXiv:2410.13276*, 2024.

[25] Saurabh Goyal, Anamitra Roy Choudhury, Saurabh Raje, Venkatesan Chakaravarthy, Yogish Sabharwal, and Ashish Verma. Power-bert: Accelerating bert inference via progressive word-vector elimination. In *International Conference on Machine Learning*, pages 3690–3699. PMLR, 2020.

[26] Wen Hao and He Qianru. Semantic comprehension method for chinese sentences based on minimal semantic structures and its application. *Chinese Journal of Electronics*, 32(3):613–624, 2023.

[27] Mark Hildebrand, Jawad Khan, Sanjeev Trika, Jason Lowe-Power, and Venkatesh Akella. Autotm: Automatic tensor movement in heterogeneous memory systems using integer linear programming. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 875–890, 2020.

[28] Neil Houlsby, Andrei Giurgiu, Stanislaw Jastrzebski, Bruna Morrone, Quentin De Laroussilhe, Andrea Gesmundo, Mona Attariyan, and Sylvain Gelly. Parameter-efficient transfer learning for nlp. In *International conference on machine learning*, pages 2790–2799. PMLR, 2019.

[29] Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. Lora: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685*, 2021.

[30] Chien-Chin Huang, Gu Jin, and Jinyang Li. Swapadvisor: Pushing deep learning beyond the gpu memory limit via smart swapping. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1341–1355, 2020.

[31] Albert Q Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, et al. Mistral 7b. *arXiv preprint arXiv:2310.06825*, 2023.

[32] Huiqiang Jiang, Qianhui Wu, Chin-Yew Lin, Yuqing Yang, and Lili Qiu. Llmlingua: Compressing prompts for accelerated inference of large language models. *arXiv preprint arXiv:2310.05736*, 2023.

[33] Gyuwan Kim and Kyunghyun Cho. Length-adaptive transformer: Train once with length drop, use anytime with search. *arXiv preprint arXiv:2010.07003*, 2020.

[34] Sehoon Kim, Sheng Shen, David Thorsley, Amir Gholami, Woosuk Kwon, Joseph Hassoun, and Kurt Keutzer. Learned token pruning for transformers. In *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 784–794, 2022.

[35] Vijay Anand Korthikanti, Jared Casper, Sangkug Lym, Lawrence McAfee, Michael Andersch, Mohammad Shoeybi, and Bryan Catanzaro. Reducing activation recomputation in large transformer models. *Proceedings of Machine Learning and Systems*, 5:341–353, 2023.

[36] Ravi Kumar, Manish Purohit, Zoya Svitkina, Erik Vee, and Joshua Wang. Efficient rematerialization for deep networks. *Advances in Neural Information Processing Systems*, 32, 2019.

[37] Mark Kurtz, Justin Kopinsky, Rati Gelashvili, Alexander Matveev, John Carr, Michael Goin, William Leiserson, Sage Moore, Nir Shavit, and Dan Alistarh. Inducing and exploiting activation sparsity for fast inference on deep neural networks. In *International Conference on Machine Learning*, pages 5533–5543. PMLR, 2020.

[38] Wonbeom Lee, Jungi Lee, Junghwan Seo, and Jaewoong Sim. {InfiniGen}: Efficient generative inference of large language models with dynamic {KV} cache management. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 155–172, 2024.

[39] Xiang Lisa Li and Percy Liang. Prefix-tuning: Optimizing continuous prompts for generation. *arXiv preprint arXiv:2101.00190*, 2021.

[40] Yucheng Li, Bo Dong, Chenghua Lin, and Frank Guerin. Compressing context to enhance inference efficiency of large language models. *arXiv preprint arXiv:2310.06201*, 2023.

[41] Zonglin Li, Chong You, Srinadh Bhojanapalli, Daliang Li, Ankit Singh Rawat, Sashank J Reddi, Ke Ye, Felix Chern, Felix Yu, Ruiqi Guo, et al. The lazy neuron phenomenon: On emergence of activation sparsity in transformers. *arXiv preprint arXiv:2210.06313*, 2022.

[42] Di Liu, Meng Chen, Baotong Lu, Huiqiang Jiang, Zhenhua Han, Qianxi Zhang, Qi Chen, Chengruidong Zhang, Bailu Ding, Kai Zhang, et al. Retrievalattention: Accelerating long-context llm inference via vector retrieval. *arXiv preprint arXiv:2409.10516*, 2024.

[43] Xiaoran Liu, Hang Yan, Shuo Zhang, Chenxin An, Xipeng Qiu, and Dahua Lin. Scaling laws of rope-based extrapolation. *arXiv preprint arXiv:2310.05209*, 2023.

[44] Xiaoxuan Liu, Lianmin Zheng, Dequan Wang, Yukuo Cen, Weize Chen, Xu Han, Jianfei Chen, Zhiyuan Liu, Jie Tang, Joey Gonzalez, et al. Gact: Activation compressed training for generic network architectures. In *International Conference on Machine Learning*, pages 14139–14152. PMLR, 2022.

[45] Zichang Liu, Jue Wang, Tri Dao, Tianyi Zhou, Binhang Yuan, Zhao Song, Anshumali Shrivastava, Ce Zhang, Yuandong Tian, Christopher Re, et al. Deja vu: Contextual sparsity for efficient llms at inference time. In *International Conference on Machine Learning*, pages 22137–22176. PMLR, 2023.

[46] I Loshchilov. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101*, 2017.

[47] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, et al. Mixed precision training. *arXiv preprint arXiv:1710.03740*, 2017.

[48] Iman Mirzadeh, Keivan Alizadeh, Sachin Mehta, Carlo C Del Mundo, Oncel Tuzel, Golnoosh Samei, Mohammad Rastegari, and Mehrdad Farajtabar. Relu strikes back: Exploiting activation sparsity in large language models. *arXiv preprint arXiv:2310.04564*, 2023.

[49] Ali Modarressi, Hosein Mohebbi, and Mohammad Taher Pilehvar. Adapler: Speeding up inference by adaptive length reduction. *arXiv preprint arXiv:2203.08991*, 2022.

[50] Tsendsuren Munkhdalai, Manaal Faruqui, and Siddharth Gopal. Leave no context behind: Efficient infinite context transformers with infini-attention. *arXiv preprint arXiv:2404.07143*, 2024.

[51] Daye Nam, Andrew Macvean, Vincent Hellendoorn, Bogdan Vasilescu, and Brad Myers. Using an llm to help with code understanding. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–13, 2024.

[52] OpenAI. ChatGPT: Get instant answers, find creative inspiration, learn something new. https://openai.com/chatgpt, 2022.

[53] Bowen Peng, Jeffrey Quesnelle, Honglu Fan, and Enrico Shippole. Yarn: Efficient context window extension of large language models. *arXiv preprint arXiv:2309.00071*, 2023.

[54] Jinxue Peng, Yong Wang, Jingfeng Xue, and Zhenyan Liu. Fast cross-platform binary code similarity detection framework based on cfgs taking advantage of nlp and inductive gnn. *Chinese Journal of Electronics*, 33(1):128–138, 2024.

[55] Xuan Peng, Xuanhua Shi, Hulin Dai, Hai Jin, Weiliang Ma, Qian Xiong, Fan Yang, and Xuehai Qian. Capuchin: Tensor-based gpu memory management for deep learning. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 891–905, 2020.

[56] Bharadwaj Pudipeddi, Maral Mesmakhosroshahi, Jinwen Xi, and Sujeeth Bharadwaj. Training large neural networks with constant memory using a new execution algorithm. *arXiv preprint arXiv:2002.05645*, 2020.

[57] Jack W Rae, Anna Potapenko, Siddhant M Jayakumar, and Timothy P Lillicrap. Compressive transformers for long-range sequence modelling. *arXiv preprint arXiv:1911.05507*, 2019.

[58] Yongming Rao, Wenliang Zhao, Benlin Liu, Jiwen Lu, Jie Zhou, and Cho-Jui Hsieh. Dynamicvit: Efficient vision transformers with dynamic token sparsification. *Advances in neural information processing systems*, 34:13937–13949, 2021.

[59] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, et al. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2023.

[60] Claude E Shannon. Prediction and entropy of printed english. *Bell system technical journal*, 30(1):50–64, 1951.

[61] Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Beidi Chen, Percy Liang, Christopher Ré, Ion Stoica, and Ce Zhang. Flexgen: High-throughput generative inference of large language models with a single gpu. In *International Conference on Machine Learning*, pages 31094–31116. PMLR, 2023.

[62] Yixin Song, Zeyu Mi, Haotong Xie, and Haibo Chen. Powerinfer: Fast large language model serving with a consumer-grade gpu. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, pages 590–606, 2024.

[63] Jiaming Tang, Yilong Zhao, Kan Zhu, Guangxuan Xiao, Baris Kasikci, and Song Han. Quest: Query-aware sparsity for efficient long-context llm inference. *arXiv preprint arXiv:2406.10774*, 2024.

[64] The Hugging Face Team. Transformers: Llama model implementation, line 735. https://github.com/huggingface/transformers/blob/08e3217bafddc5d11ce0e7369bcfaaabe5501ba5/src/transformers/models/llama/modeling_llama.py#L735, 2024. Accessed: 2025-05-13.

[65] The Hugging Face Team. Transformers: Tensor parallel integration, line 261. https://github.com/huggingface/transformers/blob/08e3217bafddc5d11ce0e7369bcfaaabe5501ba5/src/transformers/integrations/tensor_parallel.py#L261, 2024. Accessed: 2025-05-13.

[66] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.

[67] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023.

[68] Szymon Tworkowski, Konrad Staniszewski, Mikołaj Pacek, Yuhuai Wu, Henryk Michalewski, and Piotr Miłoś. Focused transformer: Contrastive training for context scaling. *Advances in Neural Information Processing Systems*, 36, 2024.

[69] Tuowei Wang, Ruwen Fan, Minxing Huang, Zixu Hao, Kun Li, Ting Cao, Youyou Lu, Yaoxue Zhang, and Ju Ren. Ripple: Accelerating llm inference on smartphones with correlation-aware neuron management, 2024.

[70] Tuowei Wang, Kun Li, Zixu Hao, Donglin Bai, Ju Ren, Yaoxue Zhang, Ting Cao, and Mao Yang. Long exposure: Accelerating parameter-efficient fine-tuning for llms under shadowy sparsity. In *2024 SC24: International Conference for High Performance Computing, Networking, Storage and Analysis SC*, pages 1176–1193. IEEE Computer Society, 2024.

[71] EC Wit and Marie Gillette. What is linguistic redundancy. *University of Chicago*, 1999.

[72] Chaojun Xiao, Pengle Zhang, Xu Han, Guangxuan Xiao, Yankai Lin, Zhengyan Zhang, Zhiyuan Liu, and Maosong Sun. Infllm: Training-free long-context extrapolation for llms with an efficient context memory. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024.

[73] Wenhan Xiong, Jingyu Liu, Igor Molybog, Hejia Zhang, Prajjwal Bhargava, Rui Hou, Louis Martin, Rashi Rungta, Karthik Abinav Sankararaman, Barlas Oguz, et al. Effective long-context scaling of foundation models. *arXiv preprint arXiv:2309.16039*, 2023.

[74] Wenhan Xiong, Jingyu Liu, Igor Molybog, Hejia Zhang, Prajjwal Bhargava, Rui Hou, Louis Martin, Rashi Rungta, Karthik Abinav Sankararaman, Barlas Oguz, et al. Effective long-context scaling of foundation models. *arXiv preprint arXiv:2309.16039*, 2023.

[75] Deming Ye, Yankai Lin, Yufei Huang, and Maosong Sun. Tr-bert: Dynamic token reduction for accelerating bert inference. *arXiv preprint arXiv:2105.11618*, 2021.

[76] Zihao Yi, Jiarui Ouyang, Yuwen Liu, Tianhao Liao, Zhe Xu, and Ying Shen. A survey on recent advances in llm-based multi-turn dialogue systems. *arXiv preprint arXiv:2402.18013*, 2024.

[77] Manzil Zaheer, Guru Guruganesh, Kumar Avinava Dubey, Joshua Ainslie, Chris Alberti, Santiago Ontanon, Philip Pham, Anirudh Ravula, Qifan Wang, Li Yang, et al. Big bird: Transformers for longer sequences. *Advances in neural information processing systems*, 33:17283–17297, 2020.

[78] Elad Ben Zaken, Shauli Ravfogel, and Yoav Goldberg. Bitfit: Simple parameter-efficient fine-tuning for transformer-based masked language-models. *arXiv preprint arXiv:2106.10199*, 2021.

[79] Longteng Zhang, Lin Zhang, Shaohuai Shi, Xiaowen Chu, and Bo Li. Lora-fa: Memory-efficient low-rank adaptation for large language models fine-tuning. *arXiv preprint arXiv:2308.03303*, 2023.

[80] Peitian Zhang, Zheng Liu, Shitao Xiao, Ninglu Shao, Qiwei Ye, and Zhicheng Dou. Long context compression with activation beacon. *arXiv preprint arXiv:2401.03462*, 2024.

[81] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, et al. Opt: Open pre-trained transformer language models. *arXiv preprint arXiv:2205.01068*, 2022.

[82] Zhenyu Zhang, Ying Sheng, Tianyi Zhou, Tianlong Chen, Lianmin Zheng, Ruisi Cai, Zhao Song, Yuandong Tian, Christopher Ré, Clark Barrett, et al. H2o: Heavy-hitter oracle for efficient generative inference of large language models. *Advances in Neural Information Processing Systems*, 36:34661–34710, 2023.

[83] Haoyi Zhou, Shanghang Zhang, Jieqi Peng, Shuai Zhang, Jianxin Li, Hui Xiong, and Wancai Zhang. Informer: Beyond efficient transformer for long sequence time-series forecasting. In *Proceedings of the AAAI conference on artificial intelligence*, volume 35, pages 11106–11115, 2021.

[84] Dawei Zhu, Nan Yang, Liang Wang, Yifan Song, Wenhao Wu, Furu Wei, and Sujian Li. Pose: Efficient context window extension of llms via positional skip-wise training. *arXiv preprint arXiv:2309.10400*, 2023.

ARTIFACT EVALUATED
usenix ASSOCIATION
AVAILABLE

FlexPipe: Maximizing Training Efficiency for Transformer-based Models with Variable-Length Inputs

Hairui Zhao[1,2+], Qi Tian[1+], Hongliang Li[1,3*], and Zizhong Chen[2]
[1] College of Computer Science and Technology, Jilin University, China.
[2] Department of Computer Science and Engineering, University of California, Riverside, USA.
[3] Key Laboratory of Symbolic Computation and Knowledge Engineering
of the Ministry of Education, China.

Abstract

Transformer achieves promising results among various deep learning architectures. Training transformer-based models (transformers) typically involves various parallelisms, such as data parallelism and pipeline parallelism (PP). Variable-length datasets have been adopted to facilitate multi-task training of transformers, which degrades training efficiency. Though many efforts have significantly improved the variable-length training, these efforts primarily focus on optimizations within a single iteration. However, substantial fluctuations of computation and memory requirements across iterations can also lead to inefficiency overall due to the static partitioning of distributed frameworks. Thus, this paper proposes FlexPipe from the perspective of a distributed system to enable high throughput variable-length training of transformers. To our knowledge, FlexPipe is the first flexible pipeline framework that dynamically adjusts PP by a live flexibility mechanism without training loss. We introduce a novel problem which aims at maximizing training throughput by adjusting the parallel configurations, along with an efficient heuristic algorithm to solve the problem. Extensive experiments show that FlexPipe achieves an average 1.25× training throughput compared to state-of-the-art methods.

1 Introduction

Recently, deep learning (DL) has shown tremendous success across a range of applications [22, 44, 52, 56]. Among various deep neural networks (DNNs), transformer-based models [43] (transformers) have emerged as the foundation for large language models (LLMs) due to their exceptional sequence modeling capabilities and scalability.

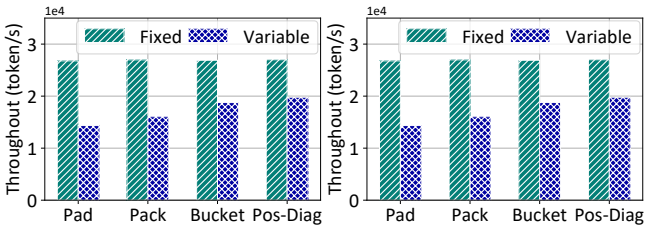
In pursuit of unprecedented accuracy, both data volume and model parameters of transformers are scaling rapidly, making it impossible to complete training on a single device. For example, GPT-4 [4], a 1.8 trillion parameters model, which required approximately 2 years to train on 1024 H100 GPUs. Thus, various parallelisms have been proposed, including data parallelism (DP) [25], tensor parallelism (TP) [17], pipeline parallelism (PP) [29, 41], and sequence parallelism (SP) [26].

Current training is mostly accommodated for traditional DNN models, presuming that they would have fixed-length inputs. However, it is not often the case for transformers, such as T5 [38], GPT [37] that are commonly trained on mixture datasets for effective multi-task learning [31, 40]. This inherently introduces variable-length inputs, e.g., the averaged 977.93 tokens per input for summarization [13], and 51.59 tokens for textual entailment [47]. The variable-length inputs significantly degrade the training efficiency of existing frameworks (i.e., padding) due to the considerable amount of wasted computations, which has attracted substantial attention [10, 16, 34, 38, 54]. Packing approach [8, 38] concatenates multiple shorter samples into a longer sample. Bucketing [34] sorts the dataset before constructing mini-batches. FlashAttention-2 [1] enables block diagonal attention by skipping irrelevant computations.

However, despite extensive efforts made by existing work, there still remains potential for further improving the variable-length training throughput. Fig. 1 shows the training throughput using different approaches, which highlights a significant performance reduction when training on a variable-length dataset compared to a fixed-length dataset. The main reason is that existing approaches focus on optimizing variable-length training within a single iteration. However, substantial fluctuations across iterations driven by variable-length inputs also lead to resource under-utilization of the overall system due to the static partitioning of existing distributed frameworks.

Existing distributed frameworks [14, 29, 32] typically launch a fixed number of devices with a static parallel strategy (i.e., combinations of different parallelism degrees) to maximize training throughput based on the memory requirements of the maximum sequence length. To alleviate resource under-utilization from fluctuations across iterations in variable-length training, the parallel strategy should be dynamically reconfigured based on sequence length. This enables

[+] Hairui Zhao and Qi Tian contributed equally to this work.
[*] Hongliang Li is the corresponding author.



(a) BERT-large on 8 GPUs (b) GPT (6.7B) on 16 GPUs

Figure 1: Throughput comparison between fixed-length and variable-length training.

the "redundant' devices (i.e., freed when handling shorter sequences) to be exploited for increasing the degree of DP, thereby improving overall efficiency.

Thus, this paper proposes FlexPipe from the perspective of a distributed system to enable high-throughput variable-length training of transformers. FlexPipe dynamically adjusts PP to meet both the computation and memory demands across iterations while maintaining the semantics of training. Then, FlexPipe utilizes "redundant" GPUs to enhance both resource utilization and training efficiency. Specifically, two main challenges are tackled when designing FlexPipe. The first challenge is how to achieve live (i.e., no termination) and efficient (i.e., minimize the GPU stalls) adjustment during the training. We introduce a live flexibility mechanism (LFM) in FlexPipe to minimize the overhead of the adjustment. The second challenge lies in determining when to perform the LFM and how to identify an appropriate new parallel strategy. We design a heuristic bound searching algorithm (HBSA) by efficiently balancing the trade-off between computational and memory usage by comprehensively considering the overhead of the LFM and other memory optimization approaches.

We make the following contributions in this paper:

- We first introduce a flexible PP framework, FlexPipe, that improves the efficiency of variable-length training by enabling the LFM of PP.
- We propose a novel flexible memory optimization problem aiming at maximizing the training efficiency by dynamically reconfiguring the parallel strategy and utilizing the "redundant" GPUs across different iterations.
- We design HBSA to solve the optimization problem while striking a good trade-off between computational efficiency and memory usage by comprehensively considering the overhead of the LFM and other memory optimization approaches.
- We implement FlexPipe [1] on Pytorch. Extensive experiments are conducted to evaluate the efficiency of FlexPipe. The results show that FlexPipe achieves an average 1.25× training throughput compared with state-of-the-art (SOTA) frameworks.

[1] https://github.com/Hairui-Zhao/FlexPipe

2 Background and Motivation

In this section, we first introduce the background of distributed transformer training, and then we illustrate this paper's motivation.

2.1 Background

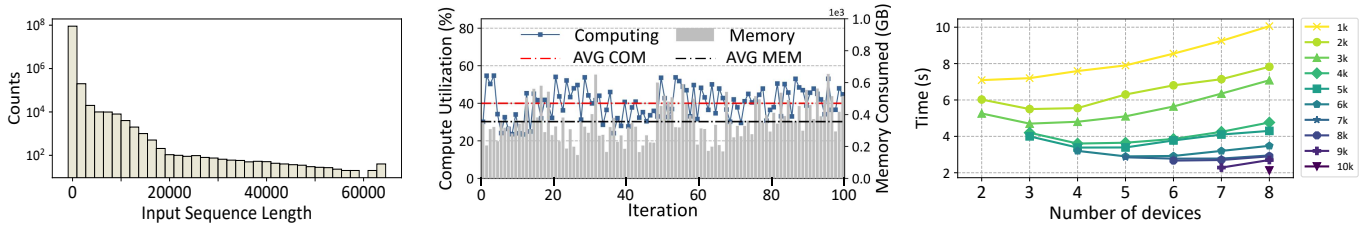
Transformers have achieved impressive performance in a variety of deep learning applications. Their architecture is typically based on encoders and decoders, which consists of multiple homogeneous encoder and decoder layers, such as BERT [7] (encoder-only) and GPT-3 [37] (decoder-only), where each layer has similar training resource consumption. As reported [16, 31], to perform multiple tasks, transformers enable training on a mixture of datasets with variable-length inputs. Self-attention is responsible for focusing on different positions of variable-length inputs. Fig. 2(a) shows the high variation in sequence lengths of a mixed dataset FLANv2 [30], the difference of sample lengths can be up to 60000 across different iterations when random sampling. Note that FlexPipe can be generalized to models beyond transformers. This paper focuses on the challenges posed by variable-length sequences, for which transformers are the most representative.

With the exponential growth of computation and memory demands, distributed training of transformers has become indispensable. Transformer training generally involves a combination of various parallelisms, including DP, TP, PP, and SP. This paper mainly focuses on the elasticity of PP, which is widely adopted in both commercial and academic DL clusters due to its considerably lower communication [12, 16, 54]. With PP, model parameter of DNN is partitioned into multiple stages at layer level. In each iteration, a mini-batch is fed into the first stage for forward propagation (FP), and the intermediate activations are transferred to the next stages. Then, during the backward propagation, the gradients are computed based on the FP activations and passed following the reverse order of FP. To increase training concurrency, PP splits the mini-batch into multiple micro-batches [9, 14, 16] or injects multiple mini-batches [32]. Finally, the model parameters are updated by the gradients synchronously or asynchronously with an optimizer, e.g., adaptive moment estimation (Adam) [19]. Note that while our optimization targets variable-length sequence training through a dynamic PP framework and leverages DP to enhance throughput, the resources freed by short sequences can likewise be repurposed for TP or even 3D parallelism to further improve scalability.

2.2 Motivation

We conduct two experiments to analyze the observations that motivate this paper.

Observation 1: Variable-length training leads to under-utilization of both computational and memory resources



(a) The sequence length distribution in FLANv2 [30].

(b) The computation efficiency and memory usage when trained on 8 GPUs.

(c) The training time under different sequence lengths when trained on different number of GPUs.

Figure 2: The observations when training GPT (3.35B) with variable-length inputs.

across different iterations. To accommodate the variable-length inputs into the traditional training frameworks [3, 35], the inputs are typically zero-padded to a fixed-length. However, this introduces extra memory consumption and wastes computation on zero tokens [16, 54]. To improve the training efficiency, the packing approach [8, 38] concatenates multiple shorter samples into longer ones, aligning its length with the maximum sequence length of input. However, it introduces computational and memory overhead, as self-attention scales quadratically with sequence length [26]. It also introduces truncation and cross-contamination, putting the training accuracy at risk. Bucketing [34] constructs mini-batches with similar sequence lengths while it destroys randomness of sample batches. Block diagonal attention [20] addresses this with position IDs and masks, and some works leverage fusion [6, 54] reduce memory usage.

Despite the forward strides made by existing work, they still suffer from significant fluctuations in both computational and memory requirements across different iterations. Fig. 2(b) illustrates the training throughput and memory usage of training GPT (3.35B) on variable-length inputs under the packing method. Such high variance in sample lengths incurs under-utilization of most iterations, their average computation throughput and memory utilizations are below 55% and 39%, respectively. Because most iterations have shorter samples but are processed using the devices designed for handling the longest samples.

The fundamental reason can be attributed to the fact that existing methods typically focus on the operator-level optimizations within a single iteration (e.g., kernel fusion, non-matmul FLOPs reduction). However, the substantial fluctuations in both computational and memory requirements across iterations also lead to under-utilization. This opens up an inter-iteration opportunity from the distributed systems perspective.

Observation 2: Static partitioning of PP leads to inefficient variable-length training. Though PP is widely used in training large-scale transformers [18, 29, 36, 41, 57], variable-length training still poses challenges to existing methods. To maximize memory usage and throughput for traditional DNN, PP frameworks (PPs) typically initialize the PP configurations (e.g., batch size, number of stages) to maximize the occupation of device memory. These configurations will not be adjusted during the entire training process. While for variable-length training, current PPs still rely on static partitioning based on the longest sequence length in the dataset to avoid out-of-memory (OOM) errors, which degrades the system efficiency.

Fig. 2(c) shows the training time of GPT (3.35B) on datasets with different fixed lengths using different numbers of GPUs. We observe that the number of devices required for efficient training varies with sample length. For instance, when training a dataset with a sample length of 3k, using 3 GPUs achieves 4.7s per iteration, the shortest training time. However, when utilizing 8 GPUs, the throughput decreases, resulting in a training time as high as 7.08s/iteration. This is because smaller samples lead to significant communication overhead and poor utilization of GPU resources due to overly fine-grained partitioning [14]. Such static strategies (i.e., based on the maximum device requirement) significantly degrades the overall training throughput. Moreover, in variable-length training, the majority of iterations are typically composed of short data samples. For example, when random sampling in FLANv2, 95% of iterations have a maximum sample length under 4k, which exacerbates the performance degradation. This motivates us to explore dynamically reconfiguring the parallel strategy across different iterations and utilizing the "redundant" GPUs to accelerate variable-length training.

Besides efficient variable-length training, there is also a growing demand for flexible training in large-scale distributed frameworks. This includes optimizing cluster utilization through dynamic scheduling [42] and meeting job-specific service level objectives [45]. In addition, flexible frameworks are crucial for addressing the challenges of heterogeneous cloud environments with fluctuating resource availability. They can also provide lower-cost fault recovery mechanisms [15, 48–51].

2.3 Challenges

Though the flexibility of PP shows its potential when dealing with variable-length and dynamic training, two challenges must be tackled to design an efficient flexible PP framework.

Firstly, dynamically adjusting PP introduces a huge overhead during training. Existing distributed frameworks utilize

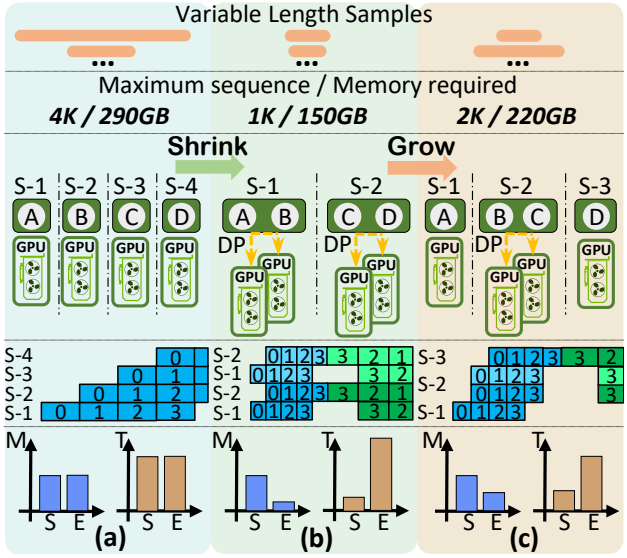


Figure 3: An example of FlexPipe when growing and shrinking. The bars (bottom of the figures) illustrate a comparison of memory usage (M) and training throughput (T). The label E represents training using FlexPipe, and S represents using the static partitioning based on a sequence length of 4K (i.e., Fig. 3(a)).

the suspend-resume or iteration stalling mechanism, which leads to a long stall due to initialization and communication overhead, usually longer than the duration of an iteration [57].

Although flexible PP could utilize "redundant" GPUs to improve the training throughput, frequent reconfiguration or an improper parallel strategy would still degrade its performance. This raises the second challenge about when to adjust PP and how to search for an appropriate parallel strategy to maximize throughput for variable-length training.

3 FlexPipe

3.1 Overview

The key insight of FlexPipe is to dynamically adjust the parallel strategy during training, adapting to the variable-length of transformers at each iteration. FlexPipe introduces a novel flexible PP framework, while previous PP frameworks rely on a fixed number of stages and static PP configurations during the whole training process, which misses opportunities to improve the variable-length training efficiency. Specifically, FlexPipe selects an appropriate number of devices used and configures PP, taking into account the computation and memory demands across iterations. Additionally, FlexPipe utilizes any idle GPUs to increase the degree of DP, enhancing both resource utilization and training efficiency.

Fig. 3 illustrates an example of flexible training with FlexPipe using four A100 GPUs to train GPT (3.35B) on a variable-length dataset with a maximum sample length of 4k. Fig. 3(a) shows that when processing a 4k sequence length mini-batch. To satisfy the memory requirement (290 GB), the model is partitioned into four stages, with each stage typically assigned to a single device (i.e., the middle of Fig. 3(a)). In this case, FlexPipe employs an identical parallel strategy and has a similar performance to the static partitioning (i.e., the bottom of Fig. 3(a), which shows the memory occupation and corresponding training throughput of GPipe and FlexPipe). This is because the maximum sequence length in the current mini-batch equals the dataset's overall maximum, leading to the same number of PP stages and configurations. In contrast, FlexPipe dynamically shrinks and grows PP stages as shown in Fig. 3(b) and (c) to improve throughput while meeting memory requirements. When training on a 1k sequence length mini-batch, FlexPipe adjusts the PP stages to 2 and leverages the remaining 2 GPUs for DP. For a 2k mini-batch, FlexPipe reconfigures the PP stages to 3 to satisfy the increased memory demands.

3.2 Design

FlexPipe is designed as an intermediary system between a typical training system (e.g., FlashAttention and Zero-Bubble) and its underlying execution engine (e.g., PyTorch [35]). FlexPipe dynamically adjusts the number of PP stages online and configurations for a given transformer-based model trained on a variable-length dataset. Fig. 4 shows the overview of FlexPipe architecture, which consists of three main modules.

Monitor profiles both the running system and devices to collect information, such as GPU states (e.g., TFlops and communication bandwidth), PP schemes (e.g., Zero-Bubble scheduler and partitioning), and DNN model states (e.g., hyperparameter and optimizer settings). Furthermore, it pre-fetches the future sequences from the data loader to minimize the profiling overhead. Based on the profiling results, it then estimates memory usage and throughput during training.

Planner leverages the collected data to decide whether to shrink or grow and generates the corresponding parallel strategy. Specifically, the Planner incorporates a trigger that first determines whether to apply a shrinking or growing algorithm to guarantee the memory requirements, based on the sequence length of the mini-batch in the next iteration. Then, the Planner calls the corresponding algorithm to evaluate the benefits of shrinking or growing (i.e., the trade-off between the throughput gains and the associated overhead) and makes a global decision, including a computation graph, adjustment decision, etc. As a way of alleviating memory pressure, PP may not always be the most cost-efficient approach. Other techniques such as recomputation [5, 14] or memory virtualization [39, 57], should also be considered. Further details on these algorithms are discussed in Section 5.

TwinLayer Manager receives global decisions from the Planner and refines them into detailed communication and computation operators. To enable an efficient and fast flexible PP mechanism, each server maintains a TwinLayer Manager.

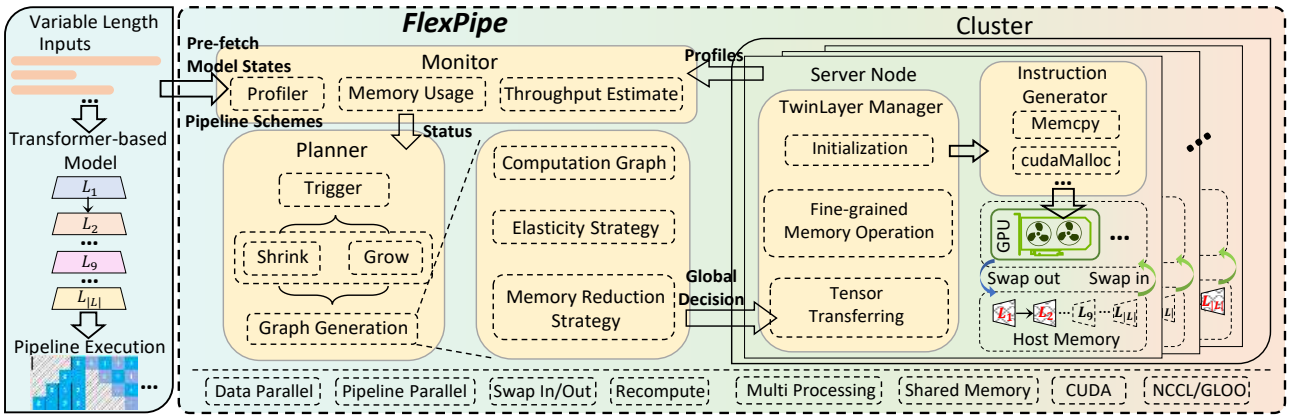


Figure 4: The overview of FlexPipe system architecture.

Additionally, all layers assigned to this node are duplicated and are stored in the host memory of each server. This design ensures flexibility and resilience and will be elaborated on in Section 4. The TwinLayer Manager is responsible for all operations (e.g., copying, swapping, and recomputation) related to the layers within each GPU, and the duplicated layers in the host memory. Then, the instruction generator converts the high-level operations from TwinLayer Manager into specific instructions, such as `cudaMalloc`, `cudaFree`, and communication primitives like `isend`.

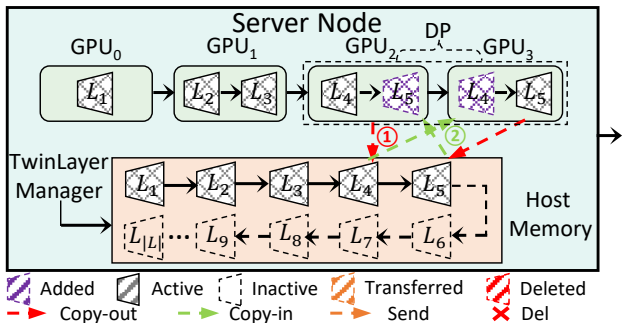
4 Live Flexibility Mechanism

To achieve high throughput with FlexPipe, the key challenge lies in implementing a fast and live flexibility mechanism. Existing PP frameworks [9, 29, 36] typically do not support an adjustable number of PP stages and configurations during runtime. Thus, resource managers [28, 59] in DL clusters that rely on these PP frameworks reconfigure the resource allocation of PP through a non-live suspend-resume mechanism. However, this incurs heavy overhead, including checkpointing, bootstrapping, and data loading. To address this challenge, a basic way to maintain live training without requiring a full resume is to initialize the stages on devices, transfer parameters, and modify dependencies between the iterations. However, the overhead of initialization and communication during the training of large-scale transformers cannot be ignored, which still leads to a long stall across iterations. In summary, both approaches incur immense overhead, which can severely degrade the training throughput when the adjustment is frequently triggered.

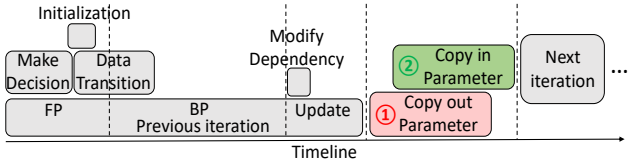
Thus, FlexPipe aims to design a *Live Flexibility Mechanism (LFM)* to transparently adjust the number of PP stages and configurations without stalling. The key idea behind this mechanism is to reduce the initialization overhead and overlap the communication and computation time. Specifically, FlexPipe introduces duplicated layers that are stored in the host memory of each node, containing all layers within that node and their corresponding optimizers, which brings two advantages. Firstly, TwinLayer supports a fine-grained layer granularity that reduces the overhead of initialization. Traditional PP frameworks operate at the stage of granularity during training, which prevents direct access to individual layers. When implementing flexibility, the parameters and optimizers of the stages must first be decomposed into individual layers, transmitted across devices, and then reassembled on the target devices. This process incurs significant overhead, e.g., 1.2s for GPT (3.35B). Secondly, by storing duplicated layers in the host memory, FlexPipe reduces the overhead of frequent memory allocations and deallocations.

Additionally, FlexPipe employs a TwinLayer Manager to efficiently orchestrate the communication and computation operations for each layer in the devices and host memory. To ease discussion, Figs. 5 and 6 illustrate examples of FlexPipe within a single node with four GPUs. Fig. 5(a) and Fig. 5(b) show the steps and their corresponding timeline when reducing a PP stage. The instructions are generated based on the global decision at the start of the previous iteration. To improve the training throughput, the freed GPUs will be utilized to increase the degree of DP. As shown in Fig. 5(a), a new DP instance is introduced between GPU_2 and GPU_3. Consequently, part of the data needs to be redistributed following the instructions for data transition. The initialization steps for PP and DP are executed following the decision-making step, which is overlapped with the data transition process (Fig. 5(b)). Then, the dependencies across different micro-batches (i.e., execution order) are modified after the BP step. Once the update of the previous iteration is complete, GPU_2 and GPU_3 send their updated parameters of layers L_4 and L_5 to the duplicated layers. Then, they respectively copy the parameters of layers L_5 and L_4 from the duplicated layers. Note that these copy steps are proceeded in a pipelined manner that copy-in process begins as soon as a portion of the parameters arrives. Then, the next iteration starts with a new configuration.

Conversely, to mitigate the memory pressure caused by training on temporary longer-length sequences, the degree of DP is decreased to add the PP stages. Fig. 6(a) and Fig. 6(b)

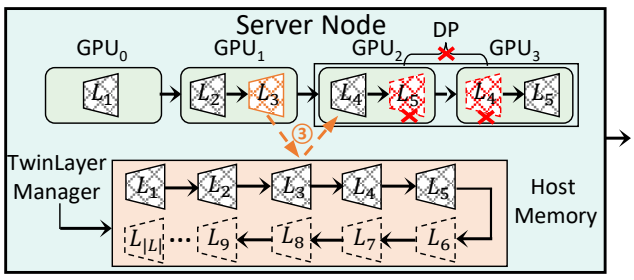


(a) Steps of adding a stage.

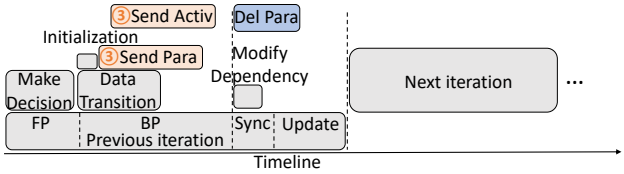


(b) Execution timeline of adding a stage.

Figure 5: Example of adding a stage with FlexPipe within a node with 4 GPUs.



(a) Steps of removing a stage.



(b) Execution timeline of removing a stage.

Figure 6: Example of removing a stage with FlexPipe within a node with 4 GPUs.

show the steps and their corresponding timeline when adding a PP stage. The same decision-making and initialization steps are first executed. Furthermore, the data redistribution and transition step is also needed to remove a worker of DP. Then, after the BP, the parameters of layers L_5 and L_4 can be directly deleted from devices GPU_2 and GPU_3, respectively. This is because all instances in DP maintain identical model parameters in each device. Once the gradients are computed during BP, the parameters can be updated after gradient synchronization without requiring the copies of parameters (Fig. 6(b)).

FlexPipe not only adjusts the number of stages but also enables flexible repartitioning by supporting layer migration between devices. As shown in Fig. 6, L_3 is migrated from GPU_1 to GPU_2 to satisfy memory requirements. By taking advantage of the characteristic of PP, where the activations generated by micro-batches in the FP are required during the BP, the transfer process can occur within the previous iteration when sufficient memory is available. After the decision-making step, the parameters are transferred immediately, and the activations of micro-batches already computed during the FP on GPU_1 are transferred to GPU_2. Then, the FP and BP for the remaining micro-batches will proceed on GPU_2.

In our evaluation, a non-live migration approach results in an average execution stall of 7.16s, while FlexPipe requires only 0.79s. The specific overhead will be discussed in Section 7. Due to space limitations, we omit the discussion of inter-server LFM, which is similar to the intra-server. The key difference lies in the overhead of transferring the TwinLayer across nodes due to lower inter-node bandwidth. This poses challenges for FlexPipe, particularly when migrating stages with large intermediate tensors. To address this, FlexPipe minimizes inter-node transfers by prioritizing intra-node remapping and pre-transferring critical data when necessary.

5 Efficient Variable-length Training

Although the LFM reduces the overhead of individual PP adjustments, frequent reconfigurations, and an improper parallel strategy can still degrade the training throughput. Therefore, an efficient algorithm is urgently needed to determine when to trigger the LFM and identify an appropriate parallel strategy. We formulate a new optimization problem, flexible memory optimization problem, which considers multiple memory optimization techniques and flexibility mechanisms to avoid unnecessary reconfigurations. To address this problem, we design a Heuristic Bound Search Algorithm (HBSA) that strikes a balance between flexibility and training efficiency.

5.1 Problem Formulation

We consider a transformer-based model trained on a mixture of datasets with variable-length samples. The model is deployed to a DL cluster and trained synchronously using PP across the device set.

System Model. With PP, the model that consists of $L = \{l_1, l_2, ..., l_{|L|}\}$ transformer layers is divided into multiple stages $S = \{s_1, s_2, ..., s_{|S|}\}$. Each stage consists of a consecutive set of layers (at least one layer, and $s_i \cap s_j \varnothing$), and the last layer of stage s_n is the predecessor of the first layer of stage s_{n+1}. For ease of formulation, homogeneous GPUs on several physical servers D are available for the training model, which is divided into multiple device groups $G = \{g_1, g_2, ..., g_{|G|}\}$ ($g_i \cap g_j = \varnothing$ and $||G|| = ||S||$). Note that $||S||$ does not necessarily have to be divisible by the number of total available GPUs $||D||$, i.e., $(||S||\%||D||) \in N$. Since "redundant" GPUs can be used to accelerate DP for part of the pipeline, as illustrated in Fig. 3(c) that has been widely adopted in hybrid

parallelisms [9].

Training Time Estimation. To estimate the training time of pipeline execution, we utilize an estimation function based on the existing work [36]. Given a mapping of stages to device groups ${}_S\Pi_G = \{s_1 \to g_1, ..., s_{|S|} \to g_{|G|}\}$, the training time $T({}_S\Pi_G, n_{micro})$ can be represented by,

$$T({}_S\Pi_G, n_{micro}) = \sum_{i=1}^{||S||} t_i + (n_{micro} - ||S|| + 1) \times \max_{1 \le j \le ||S||} t_j, \quad (1)$$

where n_{micro} is the number of micro-batches, t_i represents the time taken for a single operation of PP, including FP and BP computation, and the communication across devices (for PP and DP), which is proportional to μ.

Memory Usage. To efficiently leverage the "redundant" GPUs, we estimate the peak memory consumption of each iteration to calculate the maximum required device number for PP. Let B, μ, and M denote the global batch size, maximum sequence length, and memory capacity of a single GPU, respectively.

$$M_{peak}(L, B, \mu) = M_{p,g,o}(||L||, \mu) + M_{act}(||L||, B, \mu) + M_{buf}, \quad (2)$$

where $M_{p,g,o}$ represents the memory of parameters, gradients, and optimizer states, M_{act} and M_{buf} represent the memory of activation and other buffers.

The above runtime data (e.g., training time, communication time, batch size, and memory usage) is collected by FlexPipe's profiler during initial training iterations. Cost modeling for distributed training has been extensively studied in prior works [42], especially in DP [23] and PP [55]. Further modeling details were excluded for brevity.

Flexible Memory Optimization Problem. As a way of alleviating memory pressure, a flexibility mechanism may not always be the most cost-efficient approach. Other memory optimization approaches should also be comprehensively considered, such as recomputation [5, 14] or memory virtualization [39, 57]. Let O_{plan} represent the memory-optimized scheme utilized in the FlexPipe, including identifying layers for activation recomputation [41], portions of the optimizer to offload [27], and estimating memory savings for each action. $M_{Op}({}_S\Pi_G, O_{plan}, B, \mu)$ represents the memory consumption following optimized scheme O_{plan}. $T({}_{S'}\Pi_{G'} \to {}_S\Pi_G)$ represents the overhead of the LFM from ${}_{S'}\Pi_{G'}$ to ${}_S\Pi_G$. Let $T_{rc}(O_{plan})$ and $T_{swap}(O_{plan})$ represent the time of recomputing and offloading (swap-in, swap-out), respectively. Let M_G denote the total memory capacity of the current device groups G for PP and $M_G = M \times ||G||$, naively comparing the M_G and M_{peak} to decide whether to use the LFM may lead to inefficient adjustment, more details will be discussed in Sec. 5.2. Therefore, elaborate conditions are necessary to determine which mechanism to apply.

The memory of current ${}_{S'}\Pi_{G'}$ is less than or equal to the total memory capacity of the current device groups G', which is set as

$$condition\ \mathcal{A} : M_{peak}(L, B, \mu) \le M_{G'}. \quad (3)$$

The overhead of live flexibility from ${}_{S'}\Pi_{G'}$ to ${}_S\Pi_G$ and the computation time of ${}_S\Pi_G$ are greater than or equal the computation time of ${}_{S'}\Pi_{G'}$, which is set as

$$condition\ \mathcal{B} : T({}_S\Pi_G) + T({}_{S'}\Pi_{G'} \to {}_S\Pi_G) \ge T({}_{S'}\Pi_{G'}). \quad (4)$$

Under ${}_{S'}\Pi_{G'}$, the memory after using O_{plan} is less than or equal the total memory capacity of the current device groups G', which is set as

$$condition \quad \mathcal{C} : M_{Op}({}_{S'}\Pi_{G'}, O_{plan}, B, \mu) \le M_{G'}. \quad (5)$$

The overhead of the LFM from ${}_{S'}\Pi_{G'}$ to ${}_S\Pi_G$ and the computation time of ${}_S\Pi_G$ greater than or equal the sum of computation time of ${}_{S'}\Pi_{G'}$, the time of recomputing $T_{rc}(O_{plan})$ and the time of offloading $T_{swap}(O_{plan})$, which is set as

$$\begin{aligned} condition\ \mathcal{D} : T({}_S\Pi_G) + T({}_{S'}\Pi_{G'} \to {}_S\Pi_G) \ge T({}_{S'}\Pi_{G'}) \\ + T_{rc}(O_{plan}) + T_{swap}(O_{plan}). \end{aligned} \quad (6)$$

We use $\mathcal{T}$ to represent the iteration time of the model on the GPU cluster D,

$$\mathcal{T} = \begin{cases} T({}_{S'}\Pi_{G'}), & \mathcal{A} \wedge \mathcal{B} \\ T({}_{S'}\Pi_{G'}) + T_{rc}(O_{plan}) + T_{swap}(O_{plan}), & \neg\mathcal{A} \wedge \mathcal{C} \wedge \mathcal{D} \\ T({}_S\Pi_G) + T({}_{S'}\Pi_{G'} \to {}_S\Pi_G), & \mathcal{E} \end{cases} \quad (7)$$

where condition $\mathcal{E} = (\neg\mathcal{A} \wedge \neg\mathcal{D}) \vee (\mathcal{A} \wedge \neg\mathcal{B})$.

Definition1. Flexible Memory Optimization Problem (FMOP). *Given a transformer-base model L, a GPU cluster D, a current mapping of stages S' to device groups G' (${}_{S'}\Pi_{G'}$) and the maximum sequence length for the next iteration μ, FMOP seeks an appropriate the memory optimized scheme O_{plan} and mapping of stages to device groups ${}_S\Pi_G$ with minimum $\mathcal{T}$ considering variable-length data inputs, as:*

$$\begin{aligned} &\underset{O_{plan}, {}_S\Pi_G}{minimize}\ \mathcal{T} \\ &\text{s.t.} \quad (1), (2), (3), (4), (5), (6), (7) \end{aligned} \quad (8)$$

5.2 Algorithm

The FMOP can be formally shown as an NP-hard problem due to the extremely large search space. Moreover, frequent reconfiguration can still lead to significant performance degradation. To identify an appropriate parallel strategy and efficiently perform the LFM, we propose a Heuristic Bound Search Algorithm (HBSA), which consists of three parts: 1) computing the bounds of layer numbers in each stage and GPU numbers in each device group (Lines 3-11), 2) utilizing the bounds to navigate efficient mapping of stages and device

Algorithm 1 Heuristic Bound Search Algorithm (HBSA)

Input: Model L, GPU cluster D, time estimation functions. memory estimation functions, maximum sequence length μ. Current state: ${}_{S'}\Pi_{G'}, \mu'$.

Output: Memory optimization scheme O_{plan}, Mapping of stages to groups ${}_{S}\Pi_{G}$.

1: Initialize:$O_{plan} = \varnothing, S = \varnothing, G = \varnothing$.
2: // Exhaustive search.
3: **for** $i = 1$ *to* $||D||$ **do**
4: **if** $M_{peak}(B,\mu) < i \times M$ **then**
5: $N_{stage} = i$
6: *break*
7: // Computing the bounds.
8: **if** $N_{stage} \neq ||S'||$ **then**
9: $n^l_{low} = \lfloor ||L||/N_{stage} \rfloor, \quad n^g_{low} = \lfloor ||D||/N_{stage} \rfloor$
10: $n^l_{up} = min(\lfloor ||L||/N_{stage} \rfloor + (||L||\%N_{stage}), n^l_{max})$
11: $n^g_{up} = \lfloor ||D||/N_{stage} \rfloor + (||L||\%N_{stage})$
12: $Search(L, D, S, G, \nu)$
13: // Navigating efficient mapping.
14: **function** $Search(L, D, S_t, G_t, \nu)$
15: **for** $i = n^l_{low}$ *to* n^l_{up} **do**
16: **for** $j = n^g_{low}$ *to* n^g_{up} **do**
17: $s_t = Partition(L, i), \quad g_t = Partition(D, j)$
18: $L_t = L - s_t, \quad D_t = D - g_t$
19: **if** $||L_t|| > n^l_{up} \wedge ||D_t|| > n^g_{up}$ **then**
20: $Search(L_t, D_t, S_t \cup \{s_t\}, G_t \cup \{g_t\}, \nu)$
21: **if** $||L_t|| \in [n^l_{low}, n^l_{up}] \wedge ||D_t|| \in [n^g_{low}, n^g_{up}]$ **then**
22: $\hat{S} = S_t \cup \{s_t\} \cup \{L_t\}$
23: $\hat{G} = G_t \cup \{g_t\} \cup \{D_t\}$
24: **if** $T({}_{\hat{S}}\Pi_{\hat{G}}) + T({}_{S'}\Pi_{G'} \rightarrow {}_{\hat{S}}\Pi_{\hat{G}}) < \nu$ **then**
25: $S = \hat{S}, G = \hat{G}$
26: $\nu = T({}_{\hat{S}}\Pi_{\hat{G}}) + T({}_{S'}\Pi_{G'} \rightarrow {}_{\hat{S}}\Pi_{\hat{G}})$
27: // Determining the triggering timing.
28: **if** $\mathcal{A} \wedge \mathcal{B}$ **then**
29: **return** $O_{plan}, {}_{S'}\Pi_{G'}$ ▷①
30: **else if** $\mathcal{A} \wedge \neg\mathcal{B}$ **then**
31: **return** $O_{plan}, {}_{S}\Pi_{G}$ ▷④
32: **if** $\neg\mathcal{A} \wedge \neg\mathcal{B}$ **then**
33: $O_{plan} = init(L, {}_{S'}\Pi_{G'})$
34: **if** $\mathcal{C} \wedge \mathcal{D}$ **then**
35: **return** $O_{plan}, {}_{S'}\Pi_{G'}$ ▷②
36: **else if** $\neg\mathcal{D}$ **then**
37: **return** $O_{plan}, {}_{S}\Pi_{G}$ ▷③

groups (Lines 14-26), 3) exploiting the above conditions to determine when to trigger the LFM (Lines 28-37).

Bounds Computation. To complete the training of LLMs, it is crucial to ensure that PP can be executed without OOM errors. Therefore, the first step is to determine the minimum number of devices required to accommodate $M_{peak}(L,B,\mu)$, which corresponds to the number of stages. The HBSA can identify the minimum number of devices through an exhaustive search (Lines 3-6). To reduce the frequency and space of searching the mapping, we compare the number of stages in the next iteration N_{stage} with the current configuration $||S'||$. Based on this comparison, we compute the bounds for the number of layers in each stage (i.e., $||s_i|| \ (0 < i \leq ||S||)$) and the number of GPUs in each device group (i.e., $||g_i|| \ (0 < i \leq ||G||)$). HBSA leverages bounds to filter out a large number of invalid stage partitions and device combinations. The calculation of these bounds is based on the characteristics of transformer-based models and key observations.

The homogeneity of transformers allows us to easily predict the memory consumption of each layer, and stages with the same number of layers typically exhibit similar memory consumption. A key observation is that the variance in $||s_i|| \ (0 < i \leq ||S||)$ and $||g_i|| \ (0 < i \leq ||G||)$ is not significant in the optimal solution. Based on the heuristic, we should aim to keep the $||s_i||$ and the $||g_i||$ as consistent as possible. Therefore, we use $\lfloor ||L||/N_{stage} \rfloor$ and $\lfloor ||D||/N_{stage} \rfloor$ as the lower bounds for the $||s_i||$ and the $||g_i||$, respectively (Line 9).

Since the total number of model layers and GPUs cannot always be evenly divided by the number of stages N_{stage}, the upper bound for the $||g_i||$ is $n^g_{up} = \lfloor ||D||/N_{stage} \rfloor + (||L||\%N_{stage})$ (Line 11). Moreover, as the number of model layers per stage is also constrained by the GPU memory capacity M, where n^l_{max} represents the maximum number of layers that can be accommodated in a single GPU, the minimum value between $\lfloor ||L||/N_{stage} \rfloor + (||L||\%N_{stage})$ and n^l_{max} (Line 10) is the upper bound of layers.

Navigating mapping. To accelerate the search for efficient mapping of stages to device groups, HBSA uses the upper and lower bounds of layers and GPUs as the starting and stopping conditions for recursion. This effectively reduces the search cost by avoiding invalid and inefficient mappings caused by OOM or excessive overhead of the LFM.

In the *Search* function, we begin by using bounds to partition L and D (Lines 15-17). We then check if both remaining $||L_t||$ and $||D_t||$ exceed their upper bounds. If so, we recursively invoke the *Search* function with the remaining L_t and D_t (Lines 19-20). Conversely, if $||L_t||$ and $||D_t||$ are both within the defined bounds, we identify ${}_{\hat{S}}\Pi_{\hat{G}}$ as a valid mapping and assess whether it is the shortest in terms of time (Lines 21-26). Any other cases where $||L_t||$ and $||D_t||$ fall outside the bounds are deemed invalid mappings.

The introduction of the bound constraints effectively prevents extreme model-device partition (e.g., inefficient mappings with an excessively high degree of DP, causing communication overhead explosions). For candidate mappings within the bounds, we perform comprehensive cost modeling through Eq. 1 to estimate the training time, further searching for the most efficient mapping.

Determining triggering. Once an effective mapping is identified, it is essential to decide whether to trigger the LFM. Therefore, we use the conditions in Sec. 5 to guide this decision. As shown in Fig. 7, we summarize four types of so-

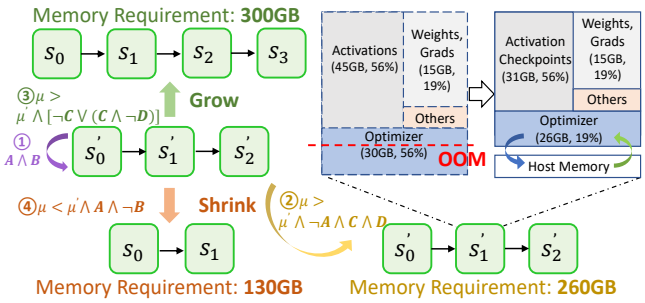


Figure 7: The trigger condition of flexibility by combining with recomputing and memory virtualization.

lutions in the HBSA, including ① maintaining the original mapping without memory optimization, ② applying current popular memory optimization techniques under the original mapping, ③ grow, which involves forming more stages and ④ shrink, which entails reducing the number of stages.

When both conditions $\mathcal{A}$ and $\mathcal{B}$ are satisfied, no action is necessary, meaning that neither memory optimization nor triggering flexibility is needed (①, Lines 28-29). If condition $\mathcal{A}$ is satisfied but condition $\mathcal{B}$ is not, then flexibility should be triggered (④, Lines30-31). When condition $\mathcal{A}$ is false, condition $\mathcal{B}$ must also be false. In such cases, it is necessary to employ memory optimization techniques (②, Lines 32-33). When both conditions $\mathcal{C}$ and $\mathcal{D}$ are true, triggering flexibility is unnecessary, and only memory optimization techniques should be utilized (Lines 34-35). If condition $\mathcal{D}$ is not met, flexibility should be activated (③, Lines 36-37). For example, as shown in Fig.7, increasing the maximum sequence length pushes GPU memory usage to 260GB, triggering an OOM error. Under condition $\mathcal{D}$, reconfiguration overhead exceeds that of memory optimization, thus FlexPipe prioritizes the memory optimization strategies (②). Specifically, in stage $S_1^{'}$, the recomputation reduces activation memory by 14GB, coupled with memory swapping that decreases optimizer memory by 4GB. Comprehensive reconfiguration details and examples are provided in Sec. 7, Tab. 2.

Here, reconfiguration is not required at every iteration during variable-length training. According to the input distribution, where the vast majority of sequences are short, only about 3% of the iterations contain long sequences. Thus, most training iterations fall within a relatively narrow resource range. Furthermore, HBSA provides a tolerant adjustment mechanism to avoid unnecessary reconfigurations by selectively combining reconfiguration with other lightweight memory optimization strategies. Empirical observations demonstrate that during training of the GPT-3.35B model on a 16-A100 GPU cluster, the system triggers grow (③) or shrink (④) at an average period of 37 iterations due to the symmetry. Memory optimization techniques (②) serve as the principal optimization methodology at an average period of 4 iterations.

6 System Implementation

FlexPipe is implemented using 8K LoC in Python and 2K LoC C++/CUDA code, a distributed DL runtime. Communication in PP and DP is implemented based on PyTorch's DDP with NCCL backend. We further describe the implementation of some of the unique aspects of FlexPipe.

Data preprocessing. To prevent delays between iterations and ensure efficient training, it is essential to preprocess the data before the training begins. Specifically, we implement a data prefetching strategy where the data loader not only fetches the current batch but also samples the next global batch in advance based on the training workflow feature. This advanced sampling process involves calculating the maximum sequence length, which ensures that the data is padded or truncated appropriately. The prepared data also prevents bottlenecks caused by input data stalls.

Model reconstruction. We designed a TwinLayer by reconstructing the original model layer. The TwinLayer minimizes the overhead of managing tensors by referencing their addresses directly rather than duplicating the tensors themselves, thereby reducing memory usage and improving computational efficiency. The TwinLayer Manager is implemented as an independent process on each server, where it runs multiple computational and communication threads using the C++ `thread` interface to avoid the Global Interpreter Lock limitations in Python's multithreading. Based on the timeline and current communication efficiency of the server, it generates effective orchestration strategies.

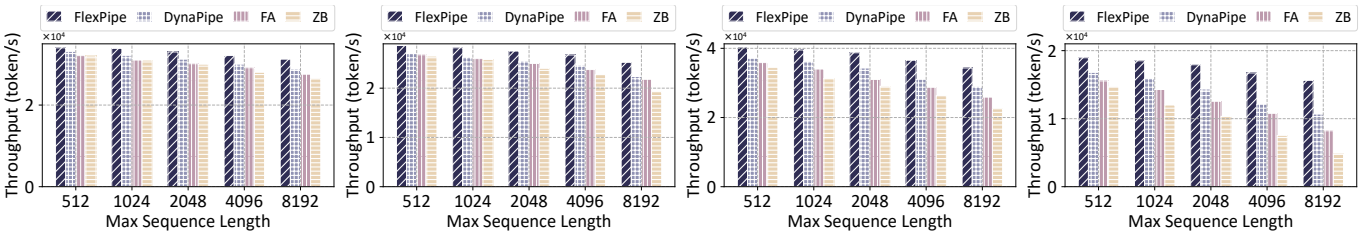
Communication transferring. We use DDP to transmit tensor between GPU-GPU and GPU-host. To leverage the duplicate channels between GPUs and the host, the weights and gradients are partitioned and transferred in a pipelined manner, which effectively reduces communication delays. To eliminate the potential disruption of training caused by live flexibility, we implemented asynchronous transfer operations using specific asynchronous interfaces, `cudaMemcpyAsync` and `isend`, which enable non-blocking data transmission.

7 Evaluation

We conduct comprehensive tests to evaluate the efficiency of FlexPipe. The results show that FlexPipe achieves 1.25× training throughput over SOTA methods on average.

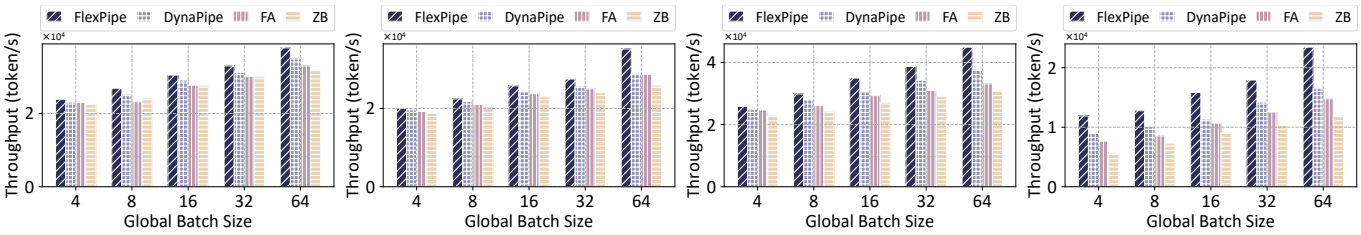
7.1 Experimental Setup

Testbed. We evaluate FlexPipe in an environment consisting of 8 NVIDIA SXM4 servers. Each node is equipped with 4 NVIDIA A100 GPUs (Ampere architecture) with 80GB of on-chip memory and is connected to a 64-core AMD EPYC 7763 CPU. The GPUs are interconnected using NVLink, providing a communication bandwidth of 300GB/s. GPU-to-CPU communication uses PCIe 4.0, offering a bandwidth



(a) BERT24 on 4 GPUs. (b) BERT96 on 8 GPUs. (c) GPT (3.35B) on 16 GPUs. (d) GPT (13B) on 32 GPUs.

Figure 8: The average training throughput under different maximum sequence lengths.



(a) BERT24 on 4 GPUs. (b) BERT96 on 8 GPUs. (c) GPT (3.35B) on 16 GPUs. (d) GPT (13B) on 32 GPUs.

Figure 9: The average training throughput under different global batch sizes.

Table 1: Transformer-based models for evaluation

Model	# Layers	# Heads	#Model Dim	# Params
BERT	24	16	1024	340M
BERT	96	16	1024	1.36B
GPT	16	32	4096	3.35B
GPT	40	40	5140	13B

of 64GB/s. These servers are connected by Infiniband with 50GB/s.The testbed runs on 64-bit SUSE Linux Enterprise Server 15 SP4 with CUDA 12.1 and PyTorch 2.1.1. As a middleware, FlexPipe is compatible with FlashAttention [6] and Zero-Bubble [36], which are incorporated to enhance the efficiency of the transformer-based model.

Baselines. FlexPipe is compared with three representative methods from different perspectives. (1) Zero-Bubble (ZB) [36] is currently the most effective PP training framework. We implement it by using padding to enable variable-length training. (2) FlashAttention (FA) [1] is widely used for optimizing LLMs from the kernel perspective, which reduces the computation time and memory footprint of attention. FlashAttention supports efficient variable-length training by packing methods and utilizes the default PP provided by PyTorch for distributed training. (3) DynaPipe [16] optimizes the variable-length training from the system perspective by supporting variable sizes of micro-batches to increase the padding efficiency.

Transformer-based models. Two transformer-based models are used in the experiments, including the variants of the GPT and BERT models. The details are shown in the Tab. 1. Both of them are trained on the FLANv2 dataset [30], one of the largest publicly available multi-task training datasets. The number of micro-batches for Zero-Bubble and FlashAttention is configured per model to maximize training throughput, while for FlexPipe and DynaPipe, it is dynamically adjusted.

Metrics. The average training throughput of actual tokens is used as the main metric, which is calculated by dividing the total number of tokens by the duration of one epoch. Moreover, we utilize the overhead of PP adjustment as a metric to evaluate the proposed flexibility mechanism.

7.2 Performance Comparison

7.2.1 Different Maximum Sequence Lengths

The average training throughput by using different methods under different maximum sequence lengths is shown in Fig. 8. We train BERT24, BERT96, GPT (3.35B), and GPT (13B) models with a global batch size of 4 on 1, 2, 4, and 8 nodes, respectively. The number of training devices is determined by the method with the highest peak memory requirement (i.e., ZA). For other methods with memory optimization techniques, the idle GPUs are utilized to accelerate training by launching DP, similar to FlexPipe. We could see that FlexPipe achieves 40.4% and 22.7% than Zero-Bubble and FlashAttention, respectively. Since Zero-Bubble and FlashAttention optimize variable-length training from a single perspective, either focusing solely on distributed systems or kernel optimization. In contrast, FlexPipe serves as a middleware system, enabling a holistic optimization approach to variable-length training. FlexPipe outperforms DynaPipe by 13.9%, as it takes accounts for fluctuations across iterations and utilizes "redundant" GPUs to further accelerate variable-length training, enhancing both performance and resource efficiency.

Moreover, as the maximum sequence length increases, throughput decreases due to a higher proportion of wasted tokens within each iteration. However, FlexPipe experiences a moderate decline in performance compared with other meth-

ods. This is because a larger maximum sequence length also amplifies the fluctuation between iterations, which FlexPipe is designed to handle more effectively.

7.2.2 Different Global Batch Sizes

Fig. 9 illustrates the average training throughput under different global batch sizes, where the maximum sequence length is set to 2048. The performance of all methods improves with the global batch size, as larger global batch sizes enable efficient parallelism in computations and reduce per-batch overhead (e.g., gradient aggregation). FlexPipe and DynaPipe increase faster than other methods because they both support dynamic distributed training according to the variable lengths across multiple iterations. The overhead of such adjustments becomes relatively smaller as the duration of each iteration increases (i.e., larger global batch size).

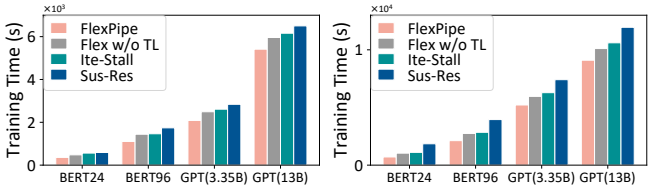
Furthermore, we observe that FlexPipe exhibits better performance as the model size increases. For instance, it improves an average performance by 9% over other algorithms with BERT24 and 57% performance improvement with GPT (13B). This can be attributed to the following two reasons: (1) larger models exhibit larger fluctuations due to variable-length inputs, providing FlexPipe with greater optimization opportunities. (2) more devices are required for training larger models, which allows FlexPipe to explore stage-to-device mappings more comprehensively. On the other hand, the flexibility mechanism heavily relies on communication, which intuitively suggests a declined performance for larger models. While FlexPipe mitigates this by overlapping the computation and communication. This optimization leverages a higher computation-to-communication ratio, enabling FlexPipe to achieve better performance with larger models.

Based on the above analysis, FlexPipe achieves an average throughput of 1.25 × than SOTA methods across different maximum sequence lengths and global batch sizes. Additionally, FlexPipe demonstrates better performance in the context of large models and large-scale setups.

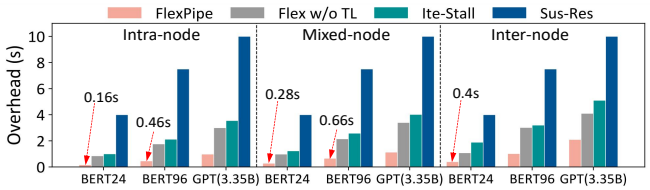
7.2.3 Overhead

We also evaluate the flexibility overhead of FlexPipe by comparing it against two common methods for adjusting PP: (1) Suspend-Resume is a non-live mechanism that suspends the training and saves the current status as a checkpoint, then restarts by loading the checkpoint with a new configuration. (2) Iteration Stalling supports live adjustment between iterations by transferring the model states without overlapping. Furthermore, to evaluate the effectiveness of TwinLayer, a degraded FlexPipe is implemented as Flex w/o TL by removing TwinLayer from the host memory.

Fig. 10(a) and 10(b) show the training time using different flexibility mechanisms under different maximum sequence lengths. We could see that FlexPipe significantly reduces



(a) Max_sequence length is 1024. (b) Max_sequence length is 4096.



(c) The detailed overhead under different envionments: intra-node consists of one node with 4 GPUs, inter-node consists of 4 nodes across 4 nodes, mixed-node consists of 2 nodes, each with 2 GPUs.

Figure 10: The flexibility overhead comparison with two common methods and a degraded FlexPipe.

35%, 23.1%, and 18.8% of the training time compared with Suspend-Resume, Iteration Stalling, and Flex w/o TL, respectively. This is because FlexPipe optimizes the initialization time required for dynamically adjusting pipelines by introducing the TwinLayer mechanism and designing elaborate fine-grained computation and communication overlap, enabling rapid flexibility and improving overall efficiency. As the maximum sequence length increases, FlexPipe demonstrates better performance compared to other methods, as discussed in Sec. 7.2.1 and 7.2.2, a larger maximum sequence length leads to more frequent fluctuations. Additionally, a larger maximum sequence length with a higher computation-to-communication ratio further reduces the flexibility overhead of FlexPipe.

Fig 10(c) further presents the detailed overhead of a single flexibility operation under different environments, including intra-node, mixed-node, and inter-node. Suspend-Resume has a stable overhead under different environments. The other methods perform better within a single node, but experience degraded performance across nodes due to their dependency on communication bandwidth. Notably, FlexPipe maintains a minimal increase in overhead across nodes, attributed to its elaborate overlap of computation and communication.

For cases where new stages are added under the GPT (3.35B), we estimate the communication overhead per reconfiguration to be approximately 0.95s, measured as the interval between completing one iteration's update and beginning the next. The performance of FlexPipe comes with the overhead of each component, typically involving a trade-off between memory, computation, and communication. However, as discussed in Sec. 5.2, reconfiguration is infrequent during variable-length training, and each adjustment yields performance benefits that persist over multiple iterations. Furthermore, FlexPipe's profiling overhead is primarily confined to early iterations, where extensive sampling is required. There-

Model	μ	Parallel Strategy	M_{peak}	O_{plan}
BERT96	512	(4,4)	140	(0,0)
	1024	(4,4)	175	(10,5)
	2048	(3,3,2)	238	(0,0)
	4096	(2,2,2,2)	352	(20,12)
	8192	(1,1,2,1,1,1,1)	596	(18,18)
GPT (3.35B)	512	(8,8)	185	(15,10)
	2048	(4,4,4,4)	315	(0,0)
	2560	(4,4,4,4)	358	(25,13)
	3072	(3,3,3,4,3)	397	(0,0))
	2560	(3,3,3,4,3)	358	(0,0)

Table 2: The configuration and training details of BERT96 and GPT (3.35B) by using FlexPipe, where μ and M_{peak} denote the sequence length and peak memory usage, respectively. The parallel strategy denotes the DP degree of each stage of PP, e.g., BERT96 is divided into 7 stages, and the degree of the third stage is 2. O_{plan} (r, s) denotes r GB activations recomputed, and s GB tensors offloaded during the training.

fore, as demonstrated by the above experiments, the cost of a single adjustment is justified when amortized across the subsequent gains in throughput.

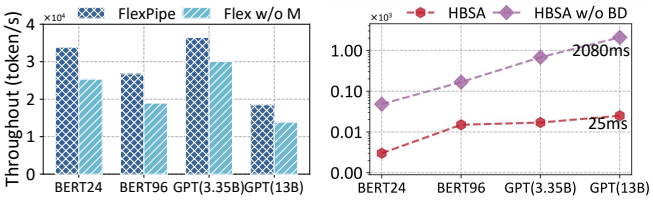
Furthermore, no significant performance degradation was observed from the compiler. According to our analysis, this is likely because FlexPipe's stage migration only remaps model stages across devices without altering the model structure or operator definitions, thus avoiding costly recompilation.

7.3 Discussion

7.3.1 Parallel Strategies of FlexPipe

To demonstrate how FlexPipe works, we train BERT96 and GPT (13B) on 2 and 8 nodes, respectively. Tab. 2 samples several representative iterations and illustrates their corresponding details. Unlike traditional training frameworks that rely on a fixed maximum number of GPUs (e.g., 16 GPUs for training GPT (3.35B)) to handle the variable sequence lengths across different iterations, FlexPipe dynamically adapts its parallel strategies according to the memory requirement to accommodate these fluctuations (e.g., 8 pipeline stages for a sequence length of 8192 and 4 stages for 512). Moreover, it effectively utilizes the "redundant" GPUs to boost the training throughput, leading to a significant performance improvement.

Note that the number of pipeline stages may vary even when the peak memory requirements are identical. For example, when training GPT (3.35B) with a sequence length of 2560, FlexPipe employed different parallel strategies across two iterations, which are influenced by the state of their previous iteration. This is because FlexPipe incorporates additional memory optimization techniques, such as recomputation and swapping, when adjusting its flexibility. FlexPipe balances the throughput improvements gained from these optimizations against their corresponding overheads.



(a) Throughput comparison. (b) Overhead comparision.

Figure 11: The ablation study of the algorithm.

Here, the number of iterations required for model convergence remains nearly identical with or without reconfiguration. FlexPipe is designed to accelerate each training iteration while maintaining strong training semantics. Therefore, even with long training runs, the reconfigurations in FlexPipe do not compromise training stability. With higher training throughput, the training tends to finish sooner. We omitted a detailed convergence analysis due to space constraints.

7.3.2 Effectiveness of Algorithm

To evaluate the effectiveness of HBSA, an ablation study is conducted by comparing FlexPipe with its modified versions. Fig. 11(a) shows the training throughput of different models using FlexPipe and Flex w/o M (only implements the flexibility mechanism, excluding recomputation and memory virtualization). We could see FlexPipe achieves an average 32.3% higher performance than Flex w/o M. When memory fluctuations are moderate, the overhead introduced by activation checkpointing and memory swapping is lower than the cost of flexibility adjustments. This mitigates the overhead caused by frequent adjustments. Furthermore, FlexPipe demonstrates even better performance advantages for larger models. This is because as the model size increases, FlexPipe w/o M requires more frequent PP adjustment, leading to greater overhead.

On the other hand, we compare the overhead of HBSA with HBSA w/o BD, which employs brute-force search based on the global solution space to determine the partitioning strategy. As shown in Fig. 11(b), HBSA achieves a significant reduction in overhead, with an average overhead of 15ms, compared to the 745ms for HBSA w/o BD. Although the overhead of HBSA increases with model size, the growth remains both reasonable and manageable.

As a heuristic algorithm, the computational overhead of HBSA is expected to grow linearly with the scale of the system. Compared to more complex methods, such as dynamic programming, HBSA remains relatively lightweight and incurs acceptable overhead. Moreover, FlexPipe's monitor performs sequence-length prefetching from the `dataloader`, allowing HBSA to be executed ahead of time and overlap with other computations. This further reduces the latency of reconfiguration decisions to the acceptable numbers of 15ms.

8 Related Work

Parallel Techniques. To deal with the explosive growth of both data volume and model complexity, a surge of studies have been proposed by researchers focused on various parallelisms [24,26,29,33,36]. With DP [24,25], multiple instances (i.e., worker) are launched by DL job, and each worker trains an identical DNN model with different parts of datasets. Thus, DP is commonly used to speed up large data training for the models that could fit on a single device. As the increasing of the model parameter, TP [17,46] splits individual layers (operation) into different devices to reduce the memory requirements. While each partition of operation requires two synchronous communication that slows down the training. Different from TP, PP [14,18,29,36] spits layers of the model into different stages, where each stage consists of a consecutive set of layers. Only transferring intermediate activations between the border layers of neighbor stages results in lower communication overhead. Recently, SP [26] aims at long sequence training by splitting the input into different devices and uses a ring-style communication, which is orthogonal to other parallelisms. This paper mainly focuses on the PP, which is indispensable in both commercial and academic DL clusters [29,33,53].

Pipeline Parallelism. Among the earliest PPs was GPipe [14], which reduces the bubble ratio by spitting a mini-batch into multiple micro-batches. Dapple [9] efficiently reduces this memory issue with a one-forward-one-backward scheduling policy, but it is not efficient enough. Hanayo [29] further improves the training throughput by utilizing a bidirectional pipeline without introducing additional memory copy. Zero-Bubble [36] nearly achieves zero bubbles by splitting the BP into two parts: computation for gradients and parameters. However, existing PPs either rely on static strategies (e.g., layer partitioning, micro-batch size) during training or reconfigure the parallel scheme based on suspend-resume [21, 42, 58] that brings huge overhead. vPipe [57] addresses the OOM issue when performing neural network search by dynamically partitioning layers. However, it cannot dynamically adjust the number of PP stages. PipeTransformer [12] improves the training efficiency of transformers by progressively freezing some layers during the training process while supporting elastic pipelines that dynamically allocate resources to the remaining active layers. However, PipeTransformer does not maintain the semantics of pipeline training, which is lossy. Though these works allow or support flexible pipelines, they do not take the variable-length training of transformers into account.

The optimizations of PPs often come at the expense of additional memory usage [18, 41, 57]. Therefore, they are typically combined with memory optimization techniques, which involve two widely used mechanisms. Memory virtualization [2,39] synchronously swaps in/out from GPU to host memory, which brings extra communication overhead. Chen et al. [5] propose recomputation that drops activations during the FP and later recomputes them when needed in BP, which reduces the memory footprint approximately to the square root of the total activations but incurs 33% extra computation.

Variable-length Training. The variable-length training of transformers has attracted lots of attention [8, 34, 38]. Traditionally, a straightforward solution is to pad all short samples with zeros to the length of the longest sequence. The naïve padding brings in redundant computations and wasted memory on padded tokens. Packing [8,38] can efficiently reduce the number of padding tokens by concatenating multiple short samples to form a long sample. However, packing is also memory inefficient because self-attention suffers from quadratic memory requirements with respect to the sequence length. Some libraries [34] sort the dataset so that each mini-batch contains samples with similar sequence lengths, which destroys the randomness and degrades the model performance (i.e., accuracy) [11].

Recently, a few DL frameworks have employed explicit designs for reducing the redundant computation caused by variable-length inputs [10,16,54]. ByteTransformer [54] proposes architecture-level optimization on multi-head attention and designs a padding-free algorithm for efficient variable-length training for transformers. DynaPipe [16] tackles the sequence length variation by advocating PP with variable-length micro-batches and optimizing micro-batch construction using dynamic programming. In contrast, FlexPipe mainly focuses on the trade-off between computational efficiency and memory usage.

9 Conclusion

This paper presents FlexPipe, a flexible PP framework that improves the efficiency of variable-length training by enabling the live flexibility mechanism (LFM) of PP. We propose a novel flexible memory optimization problem (FMOP) aiming at maximizing the training efficiency by dynamically reconfiguring parallel strategies and utilizing "redundant" GPUs across different iterations. We design an efficient heuristic bound searching algorithm to solve the optimization problem by comprehensively considering the overhead of the LFM and other memory optimization approaches. Experiments show that FlexPipe achieves an average 1.25× improvement in training throughput compared to existing SOTA frameworks.

Acknowledgment

This work is supported by the Natural Science Foundation of Jilin Province (Grant 20230101062JC), the National Key Research and Development Plan of China (Grant 2017YFC1502306), and the National Natural Science Foundation of China (Grant 62272190).

References

[1] FlashAttention Packing, Retrieved: 12/01/2024. https://huggingface.co/blog/packing-with-FA2/.

[2] NVIDIA Unified Memory, Retrieved: 10/08/2024. https://developer.nvidia.com/blog/unified-memory-cuda-beginners/.

[3] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek Gordon Murray, Benoit Steiner, Paul A. Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016*, pages 265–283, 2016.

[4] OpenAI Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, and Ilge Akkaya. Gpt-4 technical report. 2023.

[5] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. Training deep nets with sublinear memory cost. *CoRR*, abs/1604.06174, 2016.

[6] Tri Dao. Flashattention-2: Faster attention with better parallelism and work partitioning. *The Twelfth International Conference on Learning Representations, ICLR 2024*, 2024.

[7] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019*, pages 4171–4186, 2019.

[8] Hantian Ding, Zijian Wang, Giovanni Paolini, Varun Kumar, Anoop Deoras, Dan Roth, and Stefano Soatto. Fewer truncations improve language modeling. *Forty-first International Conference on Machine Learning, ICML 2024*, 2024.

[9] Shiqing Fan, Yi Rong, Chen Meng, Zongyan Cao, Siyu Wang, Zhen Zheng, Chuan Wu, Guoping Long, Jun Yang, Lixue Xia, Lansong Diao, Xiaoyong Liu, and Wei Lin. DAPPLE: a pipelined data parallel approach for training large models. *26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming,PPoPP 2021*, pages 431–445, 2021.

[10] Jiarui Fang, Yang Yu, Chengduo Zhao, and Jie Zhou. Turbotransformers: an efficient GPU serving system for transformer models. *26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2021*, pages 389–402, 2021.

[11] Ananth Gottumukkala, Dheeru Dua, Sameer Singh, and Matt Gardner. Dynamic sampling strategies for multi-task reading comprehension. *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, ACL 2020*, pages 920–924, 2020.

[12] Chaoyang He, Shen Li, Mahdi Soltanolkotabi, and Salman Avestimehr. Pipetransformer: Automated elastic pipelining for distributed training of large-scale models. *Proceedings of the 38th International Conference on Machine Learning, ICML 2021*, 139:4150–4159, 2021.

[13] Karl Moritz Hermann, Tomás Kociský, Edward Grefenstette, Lasse Espeholt, Will Kay, Mustafa Suleyman, and Phil Blunsom. Teaching machines to read and comprehend. *Advances in Neural Information Processing Systems 28: Annual Conference on Neural Information Processing Systems 2015,*, pages 1693–1701, 2015.

[14] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Xu Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V. Le, Yonghui Wu, and Zhifeng Chen. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019*, pages 103–112, 2019.

[15] Insu Jang, Zhenning Yang, Zhen Zhang, Xin Jin, and Mosharaf Chowdhury. Oobleck: Resilient distributed training of large models using pipeline templates. *Proceedings of the 29th Symposium on Operating Systems Principles*, 2023.

[16] Chenyu Jiang, Zhen Jia, Shuai Zheng, Yida Wang, and Chuan Wu. Dynapipe: Optimizing multi-task training through dynamic pipelines. *Proceedings of the Nineteenth European Conference on Computer Systems, EuroSys 2024*, pages 542–559, 2024.

[17] Ziheng Jiang, Haibin Lin, Yinmin Zhong, Qi Huang, Yangrui Chen, Zhi Zhang, Yanghua Peng, Xiang Li, Cong Xie, Shibiao Nong, Yulu Jia, Sun He, Hongmin Chen, Zhihao Bai, Qi Hou, Shipeng Yan, Ding Zhou, Yiyao Sheng, Zhuo Jiang, Haohan Xu, Haoran Wei, Zhang Zhang, Pengfei Nie, Leqi Zou, Sida Zhao, Liang Xiang, Zherui Liu, Zhe Li, Xiaoying Jia, Jianxi Ye, Xin Jin, and Xin Liu. Megascale: Scaling large language model training to more than 10, 000 gpus. *21st USENIX Symposium on Networked Systems Design and Implementation, NSDI 2024*, pages 745–760, 2024.

[18] Taebum Kim, Hyoungjoo Kim, Gyeong-In Yu, and Byung-Gon Chun. Bpipe: Memory-balanced pipeline parallelism for training large language models. *International Conference on Machine Learning, ICML 2023*, 202:16639–16653, 2023.

[19] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *3rd International Conference on Learning Representations, ICLR 2015*, 2015.

[20] Achintya Kundu, Rhui Dih Lee, Laura Wynter, Raghu Kiran Ganti, and Mayank Mishra. Enhancing training efficiency using packing with flash attention. *CoRR*, abs/2407.09105, 2024.

[21] Zhiquan Lai, Shengwei Li, Xudong Tang, Keshi Ge, Weijie Liu, Yabo Duan, Linbo Qiao, and Dongsheng Li. Merak: An efficient distributed DNN training framework with automated 3d parallelism for giant foundation models. *IEEE Trans. Parallel Distributed Syst.*, 34(5):1466–1478, 2023.

[22] Haoran Li, Zhanming Jie, and Wei Lu. Non-autoregressive machine translation as constrained HMM. *Findings of the Association for Computational Linguistics, ACL 2024,*, pages 12361–12372, 2024.

[23] Hongliang Li, Hairui Zhao, Ting Sun, Xiang Li, Haixiao Xu, and Keqin Li. Interference-aware opportunistic job placement for shared distributed deep learning clusters. *Journal of Parallel and Distributed Computing*, 183:104776, 2024.

[24] Hongliang Li, Hairui Zhao, Zhewen Xu, Xiang Li, and Haixiao Xu. Explsched: Maximizing deep learning cluster efficiency for exploratory jobs. *IEEE International Conference on Cluster Computing, CLUSTER 2023,*, pages 173–184, 2023.

[25] Mingzhen Li, Wencong Xiao, Hailong Yang, Biao Sun, Hanyu Zhao, Shiru Ren, Zhongzhi Luan, Xianyan Jia, Yi Liu, Yong Li, Wei Lin, and Depei Qian. Easyscale: Elastic training with consistent accuracy and improved utilization on gpus. *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2023*, pages 55:1–55:14, 2023.

[26] Shenggui Li, Fuzhao Xue, Chaitanya Baranwal, Yongbin Li, and Yang You. Sequence parallelism: Long sequence training from system perspective. *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2023*, pages 2391–2404, 2023.

[27] Youjie Li, Amar Phanishayee, Derek Gordon Murray, Jakub Tarnawski, and Nam Sung Kim. Harmony: Overcoming the hurdles of gpu memory capacity to train massive dnn models on commodity servers. *Proc. VLDB Endow.*, 15:2747–2760, 2022.

[28] Guodong Liu, Youshan Miao, Zhiqi Lin, Xiaoxiang Shi, Saeed Maleki, Fan Yang, Yungang Bao, and Sa Wang. Aceso: Efficient parallel DNN training through iterative bottleneck alleviation. *Proceedings of the Nineteenth European Conference on Computer Systems, EuroSys 2024*, pages 163–181, 2024.

[29] Ziming Liu, Shenggan Cheng, Haotian Zhou, and Yang You. Hanayo: Harnessing wave-like pipeline parallelism for enhanced large model training efficiency. *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2023*, pages 56:1–56:13, 2023.

[30] Shayne Longpre, Le Hou, Tu Vu, Albert Webson, Hyung Won Chung, Yi Tay, Denny Zhou, Quoc V. Le, Barret Zoph, Jason Wei, and Adam Roberts. The flan collection: Designing data and methods for effective instruction tuning. *International Conference on Machine Learning, ICML 2023*, 202:22631–22648, 2023.

[31] Swaroop Mishra, Daniel Khashabi, Chitta Baral, and Hannaneh Hajishirzi. Cross-task generalization via natural language crowdsourcing instructions. *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics , ACL 2022*, pages 3470–3487, 2022.

[32] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, Phillip B. Gibbons, and Matei Zaharia. Pipedream: generalized pipeline parallelism for DNN training. *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019*, pages 1–15, 2019.

[33] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, Amar Phanishayee, and Matei Zaharia. Efficient large-scale language model training on GPU clusters using megatron-lm. *International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2021*, page 58, 2021.

[34] Myle Ott, Sergey Edunov, Alexei Baevski, Angela Fan, Sam Gross, Nathan Ng, David Grangier, and Michael Auli. fairseq: A fast, extensible toolkit for sequence modeling. *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019*, pages 48–53, 2019.

[35] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen,

Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Z. Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019,*, pages 8024–8035, 2019.

[36] Penghui Qi, Xinyi Wan, Guangxing Huang, and Min Lin. Zero bubble (almost) pipeline parallelism. *The Twelfth International Conference on Learning Representations, ICLR 2024*, 2024.

[37] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. *OpenAI blog*, 2019.

[38] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *J. Mach. Learn. Res.*, 21:140:1–140:67, 2020.

[39] Minsoo Rhu, Natalia Gimelshein, Jason Clemons, Arslan Zulfiqar, and Stephen W. Keckler. vdnn: Virtualized deep neural networks for scalable, memory-efficient neural network design. *49th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2016*, pages 18:1–18:13, 2016.

[40] Victor Sanh, Albert Webson, Colin Raffel, Stephen H. Bach, Lintang Sutawika, Zaid Alyafeai, Antoine Chaffin, Arnaud Stiegler, Arun Raja, Manan Dey, M Saiful Bari, Canwen Xu, Urmish Thakker, Shanya Sharma Sharma, Eliza Szczechla, Taewoon Kim, Gunjan Chhablani, Nihal V. Nayak, Debajyoti Datta, Jonathan Chang, Mike Tian-Jian Jiang, Han Wang, Matteo Manica, Sheng Shen, Zheng Xin Yong, Harshit Pandey, Rachel Bawden, Thomas Wang, Trishala Neeraj, Jos Rozen, Abheesht Sharma, Andrea Santilli, Thibault Févry, Jason Alan Fries, Ryan Teehan, Teven Le Scao, Stella Biderman, Leo Gao, Thomas Wolf, and Alexander M. Rush. Multitask prompted training enables zero-shot task generalization. *The Tenth International Conference on Learning Representations, ICLR 2022*, 2022.

[41] Zhenbo Sun, Huanqi Cao, Yuanwei Wang, Guanyu Feng, Shengqi Chen, Haojie Wang, and Wenguang Chen. Adapipe: Optimizing pipeline parallelism with adaptive recomputation and partitioning. *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3, ASPLOS 2024*, pages 86–100, 2024.

[42] Taegeon Um, Byungsoo Oh, Minyoung Kang, Woo-Yeon Lee, Goeun Kim, Dongseob Kim, Youngtaek Kim, Mohd Muzzammil, and Myeongjae Jeon. Metis: Fast automatic distributed training on heterogeneous gpus. *Proceedings of the 2024 USENIX Annual Technical Conference, USENIX ATC 2024*, pages 563–578, 2024.

[43] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017*, pages 5998–6008, 2017.

[44] Chen Wang, Ziwei Fan, Liangwei Yang, Mingdai Yang, Xiaolong Liu, Zhiwei Liu, and Philip S. Yu. Pre-training with transferable attention for addressing market shifts in cross-market sequential recommendation. *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, KDD 2024*, pages 2970–2979, 2024.

[45] Haoyu Wang, Zetian Liu, and Haiying Shen. Machine learning feature based job scheduling for distributed machine learning clusters. *IEEE/ACM Transactions on Networking*, 31:58–73, 2023.

[46] Minjie Wang, Chien-Chin Huang, and Jinyang Li. Supporting very large models using automatic dataflow graph partitioning. *Proceedings of the Fourteenth EuroSys Conference 2019, Dresden*, pages 26:1–26:17, 2019.

[47] Adina Williams, Nikita Nangia, and Samuel R. Bowman. A broad-coverage challenge corpus for sentence understanding through inference. *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2018*, pages 1112–1122, 2018.

[48] Shixun Wu, Yitong Ding, Yujia Zhai, Jinyang Liu, Jiajun Huang, Zizhe Jian, Huangliang Dai, Sheng Di, Bryan M Wong, Zizhong Chen, et al. Ft k-means: A high-performance k-means on gpu with fault tolerance. In *2024 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 322–334. IEEE, 2024.

[49] Shixun Wu, Yujia Zhai, Jiajun Huang, Zizhe Jian, and Zizhong Chen. Ft-gemm: A fault tolerant high performance gemm implementation on x86 cpus. In *Proceedings of the 32nd International Symposium on High-Performance Parallel and Distributed Computing*, pages 323–324, 2023.

[50] Shixun Wu, Yujia Zhai, Jinyang Liu, Jiajun Huang, Zizhe Jian, Huangliang Dai, Sheng Di, Franck Cap-

pello, and Zizhong Chen. Turbofft: Co-designed high-performance and fault-tolerant fast fourier transform on gpus. In *Proceedings of the 30th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, pages 70–84, 2025.

[51] Shixun Wu, Yujia Zhai, Jinyang Liu, Jiajun Huang, Zizhe Jian, Bryan Wong, and Zizhong Chen. Anatomy of high-performance gemm with online fault tolerance on gpus. In *Proceedings of the 37th International Conference on Supercomputing*, pages 360–372, 2023.

[52] Yisheng Xiao, Lijun Wu, Junliang Guo, Juntao Li, M. Zhang, Tao Qin, and Tie-Yan Liu. A survey on non-autoregressive generation for neural machine translation and beyond. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, pages 11407–11427, 2022.

[53] Tailing Yuan, Yuliang Liu, Xucheng Ye, Shenglong Zhang, Jianchao Tan, Bin Chen, Chengru Song, and Di Zhang. Accelerating the training of large language models using efficient activation rematerialization and optimal hybrid parallelism. *Proceedings of the 2024 USENIX Annual Technical Conference, USENIX ATC 2024*, pages 545–561, 2024.

[54] Yujia Zhai, Chengquan Jiang, Leyuan Wang, Xiaoying Jia, Shang Zhang, Zizhong Chen, Xin Liu, and Yibo Zhu. Bytetransformer: A high-performance transformer boosted for variable-length inputs. *IEEE International Parallel and Distributed Processing Symposium, IPDPS 2023*, pages 344–355, 2023.

[55] Hairui Zhao, Hongliang Li, Qi Tian, Jie Wu, Meng Zhang, Z Xu, X Li, and H Xu. Arraypipe: Introducing job-array pipeline parallelism for high throughput model exploration. In *Proc. of the IEEE International Conference on Computer Communications (INFOCOM)*, 2025.

[56] Hairui Zhao, Xinyu Li, and Hongliang Li. Visage: Visual-aware generation of adversarial examples in black-box for text classification. In *CCF International Conference on Natural Language Processing and Chinese Computing*, pages 440–453. Springer, 2024.

[57] Shixiong Zhao, Fanxin Li, Xusheng Chen, Xiuxian Guan, Jianyu Jiang, Dong Huang, Yuhao Qing, Sen Wang, Peng Wang, Gong Zhang, Cheng Li, Ping Luo, and Heming Cui. vpipe: A virtualized acceleration system for achieving efficient and scalable pipeline parallel DNN training. *IEEE Trans. Parallel Distributed Syst.*, 33(3):489–506, 2022.

[58] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P. Xing, Joseph E. Gonzalez, and Ion Stoica. Alpa: Automating inter- and intra-operator parallelism for distributed deep learning. *16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2022*, pages 559–578, 2022.

[59] Quan Zhou, Haiquan Wang, Xiaoyan Yu, Cheng Li, Youhui Bai, Feng Yan, and Yinlong Xu. Mpress: Democratizing billion-scale model training on multi-gpu servers via memory-saving inter-operator parallelism. *IEEE International Symposium on High-Performance Computer Architecture, HPCA 2023*, pages 556–569, 2023.

Optimus: Accelerating Large-Scale Multi-Modal LLM Training by Bubble Exploitation

Weiqi Feng[1]*, Yangrui Chen[2], Shaoyu Wang[3]*, Yanghua Peng[2], Haibin Lin[2] and Minlan Yu[1]
[1]*Harvard University,* [2]*Bytedance,* [3]*University of Southern California*

Abstract

Multimodal large language models (MLLMs) have extended the success of large language models (LLMs) to multiple data types, such as image, text and audio, achieving significant performance in various domains, including multimodal translation, visual question answering and content generation. Nonetheless, existing systems are inefficient to train MLLMs due to substantial GPU bubbles caused by the heterogeneous modality models and complex data dependencies in 3D parallelism. This paper proposes Optimus, a distributed MLLM training system that reduces end-to-end MLLM training time. Optimus is based on our principled analysis that scheduling the encoder computation within the LLM bubbles can reduce bubbles in MLLM training. To enable scheduling encoder computation for all GPUs, Optimus searches for separate parallel plans for the encoder and LLM, and adopts a bubble scheduling algorithm to exploit LLM bubbles without breaking the original data dependencies in the MLLM model architecture. We further decompose the encoder layer computation into a series of kernels and analyze the common bubble pattern of 3D parallelism to carefully optimize the sub-millisecond bubble scheduling, minimizing the overall training time. Our experiments in a production cluster show that Optimus accelerates MLLM training by 20.5%-21.3% with ViT-22B and GPT-175B model over 3072 GPUs compared to baselines.

1 Introduction

Multimodal Large Language Models (MLLMs) build upon the advancements of Large Language Models (LLMs) by extending their capabilities to comprehend and generate content across multiple modalities, such as text, images, and audio. Notable MLLMs, including GPT-4V [21], Google Gemini [28], Grok-1.5 Vision [33] and LLava [18], have achieved remarkable progress in domains like visual question answering [2, 19], multimodal translation [27, 34], and content generation and understanding [21, 28, 40]. The substantial computational demands of MLLMs underscore the urgent need to enhance training performance to fully harness their capabilities.

Multimodal large language models (MLLMs) typically integrate multiple encoders, each specialized for processing a specific modality, alongside a substantial language model component. Multimodal data are input into their respective encoders, and the resulting outputs are combined to form the input for the language model.

The multimodal encoders and the language model exhibit significant differences in functionality, architecture, and input data sizes, leading to varied resource demands. However, existing distributed training systems are mainly designed for sequential unimodal (e.g., MegaScale [13], Megatron-LM [20], Chimera [16]), and fall short in training MLLMs. For example, when training a large MLLM containing several hundred billion parameters using Megatron-LM on over 3,000 GPUs, more than 40% of GPU cycles remain idle. Upon analyzing typical MLLM training tasks, we identified two critical issues: (1) Communication overhead in 3D parallelism is extensive and frequent, resulting in significant GPU idle times; (2) The pipeline stages of MLLM are imbalanced, and the data dependency between adjacent pipeline stages results in considerable data waiting time. Existing solutions can be categorized into two groups: (1) optimizing LLM, e.g.Megatron-LM and Zero-bubble pipeline [23]; (2) optimizing multimodal encoders, e.g.DistMM [12]. Nonetheless, none of the existing works consider LLM and encoders simultaneously. As demonstrated in Section 2.2, around 48% of GPU cycles are wasted in our internal large-scale MLLM training task.

In this paper, we introduce Optimus, a distributed MLLM training system that enables scheduling encoder computations within idle periods — referred to as "LLM bubbles" — to achieve efficient 3D parallelism. Scheduling encoder computations within these LLM bubbles using existing training frameworks is challenging for three reasons.

First, existing training frameworks, e.g., Megatron-LM [20],

*Work done during internship at Bytedance.

MegaScale [13], and zero-bubble pipeline [23], employ unified parallel strategies to MLLM models, distributing encoder and LLM layers across different GPUs. As a result, most GPUs contain only LLM model states and are incapable of performing encoder computations during LLM bubbles. In contrast, our approach utilizes separate parallelization plans for encoders and LLMs to colocate both encoder and LLM model states on each GPU. We systematically enumerate potential 3D parallelism plans for the encoder and eliminate those that violate GPU memory constraints.

Second, complex data dependencies inherent in MLLM constrain the scheduling of encoder computations within LLM bubbles. These dependencies include those related to synchronous training iterations and internal dependencies within the encoder itself, as discussed in Section 2.3. The most intricate dependency is the microbatch-level data dependency between the encoder and LLM, which requires that the encoder completes its forward pass before the LLM begins its forward pass for each microbatch, and that the encoder's backward pass commences only after the LLM has completed its backward pass for each microbatch. To manage these dependencies, we employ a two-stage dependency management strategy: local scheduling to address the first two types of dependencies and global ordering to handle the encoder-LLM microbatch-level dependencies.

Third, the LLM bubble duration varies significantly, ranging from sub-millisecond intervals to several hundred milliseconds, posing a considerable challenge for bubble reduction. Existing frameworks [16, 20, 23] schedule computations at the layer level, and sub-millisecond bubbles are too brief to complete even a single encoder layer forward or backward. To address this, we decompose encoder layer computations into sequences of kernels, enabling the effective utilization of these brief bubbles. Furthermore, we analyze common patterns in LLM bubbles and optimize the scheduling by interleaving encoder kernel computations with LLM computations, thereby minimizing the overall training time.

Optimus was implemented based on Megatron-LM, incorporating the aforementioned design principles. Comprehensive evaluations were conducted using several representative MLLM models. The results are promising - Optimus outperforms state-of-the-art baselines by 20.3% on average and Optimus also scales well with the size of models and GPUs. Our experiments in a production cluster show that Optimus accelerates MLLM training by 20.5%-21.3% with ViT-22B and GPT-175B models over 3072 GPUs compared to baselines.

2 Background

2.1 Multimodal LLM Characteristics

Multimodal LLMs are increasingly important. These models build upon the foundational principles of LLMs by integrating advanced natural language processing methodologies while extending their scope to encompass diverse data modalities. GPT-4 [21] exemplifies such an advancement, enhancing the capabilities of its predecessors to include multimodal understanding and generation. It demonstrates human-level performance across various benchmark tests involving both image and text inputs.

A multimodal large language model (MLLM) typically comprises three primary components: one or more modality encoders, input projectors, and a large language model backbone [36]. The Modality Encoders process inputs from non-textual modalities into their respective feature representations, while the input projector aligns these features with the text feature space. Subsequently, the LLM backbone utilizes the aligned multimodal and textual features as input. Figure 1 illustrates the architecture of the MLLM. The input projector is excluded from further discussion due to its relatively minimal computational demands compared to the encoder and the LLM backbone, as detailed in Llava [18]. For this analysis, the input projector is considered the final layer of the modality encoder.

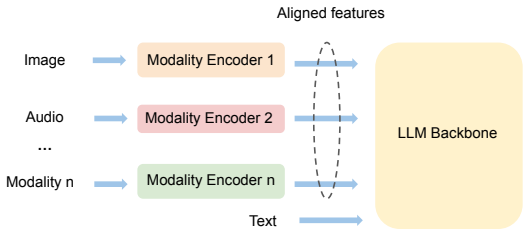


Figure 1: Multimodal model architecture.

Different from homogeneous LLM architecture, multimodal LLM has the following unique characteristics.

Dominant Model Size of LLM Backbone: In multimodal LLMs, the LLM backbone has a significantly larger number of parameters compared to other components such as encoders and projectors. For instance, Flamingo [3] boasts a total of 80 billion parameters, with its LLM backbone alone comprising 70 billion parameters.

Dependency between Encoders and LLM Backbone: In MLLM training, there are two types of data dependencies between encoders and LLM. During the forward pass, encoders must complete the generation of encoded features before the LLM backbone can proceed with forwarding. Conversely, in the backward pass, the LLM backbone calculates gradients before the encoders initiate the backward pass.

2.2 Bubbles in MLLM Training

Existing LLM pipeline optimizations lack model-agnostic properties and are inadequate for MLLM training tasks. In our internal large-scale MLLM training tasks with ViT encoder and GPT backbone (over 100B parameters), we employed

Megatron-LM across more than 3,000 NVIDIA GPUs and observed over 48% GPU cycle idleness despite utilizing multiple SOTA techniques, including MegaScale [13], Zero Bubble Pipeline [23], fine-grained communication-computation overlapping [32]. We analyzed the profiled timeline to identify and investigate the occurrences of GPU idleness (i.e., bubbles). Table 1 shows the total duration and percentage of average training step time (5.12s) occupied by different types of bubbles.

Bubble types	Percentage	Total time (s)
DP bubble (all-gather)	3.3%	0.167
DP bubble (reduce-scatter)	8.9%	0.458
PP bubbles (warmup)	5.0%	0.291
PP bubbles (cooldown)	9.2%	0.471
PP bubbles (other)	8.7%	0.445
TP bubble	11.2%	0.585

Table 1: Total time and percentage of average training step time (5.12s) occupied by different types of bubbles.

These bubbles can be classified into three categories based on their underlying causes.

(1) Communication in Data Parallelism (DP). Data parallelism requires communication to aggregate gradients, leading to GPU idle time during these communications. Specifically, MegaScale [13] and Megatron-LM [25] employ a distributed optimizer, similar to P_{os+g} in ZeRO [24], to save memory for large model training, which performs two collective communications: *all-gather* and *reduce-scatter*. At the start of each training step, an all-gather operation collects updated parameters from all DP ranks, resulting in a DP all-gather bubble that occupies 3.3% of the training time. At the end of the training step, reduce-scatter is performed to aggregate gradients, leading to a DP reduce-scatter bubble that consumes 8.9% of the training time[1]. It should be noted that overlapping optimization in data parallelism proposed in Megascale [13] has already been applied; however, the aforementioned DP communications for the first model chunk can not be hidden due to the nature of synchronous training [13]. Figure 2 illustrates DP bubbles arising from all-gather and reduce-scatter operations at the start and conclusion of each training step.

(2) Dependency in Pipeline Parallelism (PP). Despite applying pipeline send-receive overlap optimization from Megascale [13], pipeline bubbles persist due to the inherent data dependencies between stages during the forward and backward passes. Importantly, the Zero Bubble Pipeline method cannot eliminate pipeline bubbles in MLLM training, owing to the required changes in the optimizer [23] (see discussions in §7). Figure 2 illustrates the MLLM training pipeline schedule, comprising three phases: warm-up (forward only), steady (one forward and one backward), and cool-down (backward only). Throughout pipeline training, three types of bubbles arise:

- PP warm-up bubbles occur at all stages except the first one due to the forward dependency of the first forward pass, averaging 5.0% of the training time.
- PP cool-down bubbles occur at all stages except the first one due to the backward dependency of the final backward pass, averaging 9.2% of the training time.
- Other PP bubbles manifest in all stages except the last one due to dependencies of other forward and backward passes, occupying 8.7% of training time. For instance, PP bubbles emerge immediately after the PP warm-up phase due to the backward dependency of the initial backward pass. Additionally, in cases where pipeline stages are imbalanced due to the heterogeneity of MLLM models, extra pipeline bubbles not depicted in Figure 2 occur.

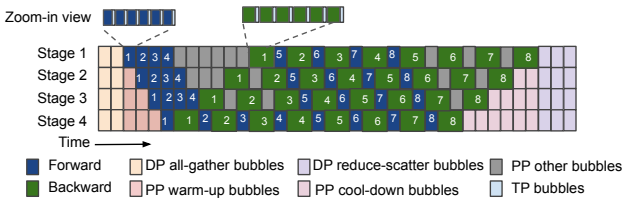


Figure 2: Timeline illustration of MLLM training showing different categories of bubbles (simplified based on the Megatron-LM 1F1B schedule [20]).

(3) Communications in Tensor Parallelism (TP). Tensor parallelism involves partitioning individual layers across multiple GPUs, necessitating communication during forward and backward passes to synchronize between GPUs. In Megatron-LM, each forward or backward pass of a transformer layer requires two all-gather and two reduce-scatter kernels [14]. Figure 3 provides a detailed view of CUDA computation and communication kernels during two GPT-175B [5] layer forward passes. Green kernels represent all-gather communications in the CUDA communication stream, while blue kernels denote reduce-scatter communications. The compute stream idles during these communications. Typically, these TP bubbles last for sub-millisecond durations, averaging around 300 μs. However, during MLLM training, there are thousands of TP bubbles, collectively accounting for 11.2% of the training time.

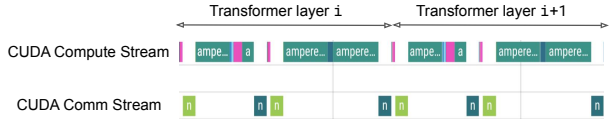


Figure 3: Zoom-in view of TP bubbles during two LLM layers forward.

[1]The reduce-scatter bubble is larger than the all-gather bubble because it involves higher communication volume (FP32 gradients vs. BF16 parameters) and suffers from synchronization delays due to straggling GPUs at the end of the training step.

2.3 Challenges

To minimize bubbles in MLLM training, we aim to exploit the distinct dual-component structure of MLLM, which consists of encoders and the LLM backbone. We make two key observations. First, the majority of bubbles during MLLM training occur during the forward and backward passes of the LLM backbone, with around 90% arising from LLM communication based on production data. Second, the encoders require fewer computational operations (FLOPs) than the LLM backbone due to their smaller parameter size [4, 7, 10, 17, 18].

Based on these insights, we propose **scheduling encoder computations in LLM bubbles** which occur during LLM communication to minimize bubbles throughout the MLLM training process. We identify three primary challenges of scheduling encoder computation to LLM bubbles.

Challenge 1: Only a limited number of GPUs have both encoder and LLM model states. Current training systems [20, 39] employ pipeline parallelism to distribute the MLLM as a single pipeline across multiple GPUs. Due to the dependency between the encoder and LLM, encoder layers are assigned to initial pipeline stages, while LLM layers are assigned to later pipeline stages. Consequently, only one pipeline stage typically contains both the encoder and LLM layers. To illustrate, Figure 4 demonstrates the use of 3D parallelism (DP=1, PP=4, TP=2) to parallelize MLLM across 8 GPUs, where only 2 GPUs in pipeline stage 1 hold both the encoder and LLM model states. The remaining 6 GPUs cannot execute encoder computations during LLM bubbles because they lack encoder model states.

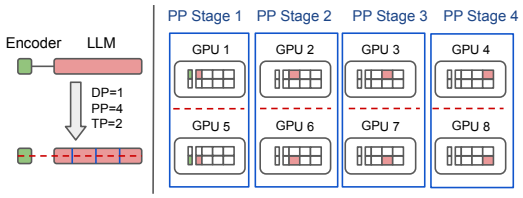


Figure 4: Only GPUs in pipeline stage 1 have both encoder and LLM model states.

Challenge 2: Complex Dependencies in MLLM Training. The complex dependencies inherent in MLLM training pose significant challenges when scheduling encoder computation within LLM bubbles. First, in synchronous training, the utilization of LLM bubbles is restricted to performing encoder computations solely within the current training iteration (*iteration dependency*). Second, dependencies within the encoder pipeline require that the forward computation of the current encoder pipeline stage i is scheduled only after the completion of the previous encoder stages, and the backward computation can be scheduled only after the subsequent encoder stage concludes. Lastly, the encoder-LLM dependency imposes microbatch-level constraints: the encoder must complete the forward pass of microbatch i before the LLM pipeline can initiate the forward pass of the same microbatch. Similarly, the encoder can begin the backward pass of microbatch i only after the LLM pipeline has completed its backward pass of microbatch i.

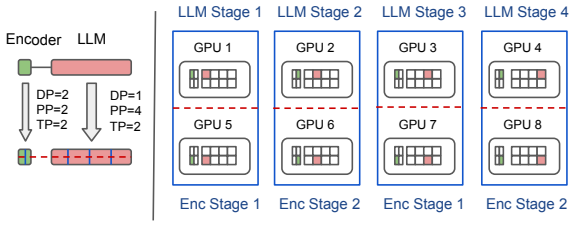


Figure 5: All GPUs both hold encoder and LLM model states when giving encoder and LLM separate parallel plans.

Challenge 3: Sub-millisecond LLM bubbles. Existing frameworks like MegaScale [13] and Megatron-LM [20] typically schedule at layer level. However, bubbles in the LLM exhibit a wide range of durations, spanning from sub-milliseconds (TP bubbles) to hundreds of milliseconds (DP bubbles). For instance, TP bubbles shown in Figure 3 average around 300μs across different LLM layers during forward and backward passes. This duration is insufficient to complete even a single encoder layer forward or backward. To illustrate, a single ViT-22B layer typically requires around 1.4 milliseconds to complete forward propagation and 2.0 milliseconds to complete backward propagation.

3 Design Decisions and System Overview

We discuss the core design decisions that drive Optimus design and provide an overview of Optimus. The next section discusses the detailed design.

3.1 Design Decisions

Design decision 1: Colocate encoders and LLM with separate parallelism. To ensure that each GPU possesses both encoder and LLM model states, we propose assigning separate parallel plans to encoders and LLMs across all GPUs. This strategy is illustrated in Figure 5, where using parallel plan (DP=2, PP=2, TP=2) for encoders and (DP=1, PP=4, TP=2) for LLM. Each GPU retains both encoder and LLM model states, and then it becomes feasible for all GPUs to execute encoder computations during LLM bubbles. Note that colocating both the encoder and LLM states may require more GPU memory, and we analyze the memory overhead in Section 4.5.

Design decision 2: Dual-Stage Dependency Management. We use two stages to handle complex dependencies in MLLM training: local scheduling and global ordering. Each encoder pipeline undergoes local scheduling, which schedules encoder computations with available LLM bubbles, adhering

to the iteration-dependency and encoder-internal dependencies. Global ordering ensures microbatch-level dependency between encoders and LLM by sequencing the encoder's ending times forward and the encoder's starting times backward across microbatches. This involves comparing timestamps to verify encoder-LLM dependency compliance. As shown in Figure 6, local scheduling is applied independently to two encoder pipelines, maintaining iteration dependency and encoder-internal dependency. In global ordering, timestamps across all microbatches (totaling 8) are checked to confirm that encoder-LLM dependencies are met.

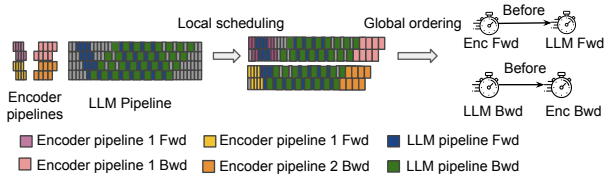


Figure 6: Solve complex dependencies in MLLM training through local scheduling and global ordering.

Design Decision 3: Schedule encoder computation at Kernel Level. Decomposing the encoder layer into kernels enables efficient utilization of sub-millisecond bubbles. However, TP communication kernels in the encoder layer compete for link bandwidth during LLM TP bubbles, causing longer time per iteration. To resolve this, we must additionally schedule encoder communication kernels during LLM compute (see Figure 7).

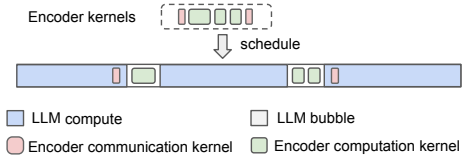


Figure 7: Schedule encoder computation kernels within LLM bubbles and encoder communication kernels within LLM compute.

3.2 Optimus Overview

Optimus is a distributed training system designed for MLLM, enabling the scheduling of encoder computation within LLM bubbles to improve end-to-end training latency. To tackle the challenges outlined in Section 3.1, Optimus comprises two components: the model planner, which addresses Challenge 1 by ensuring all GPUs hold both encoder and LLM model states, and the bubble scheduler, which addresses Challenge 2 (complex dependencies in MLLM training) and Challenge 3 (sub-millisecond LLM bubbles).

Model Planner. The model planner partitions encoders and the LLM backbone separately to all given GPUs (addressing Challenge 1 in §3.1). It selects a 3D parallelism plan for the LLM backbone and explores possible 3D parallelism plans for the encoders, considering the available GPU memory after the deployment of the LLM. With the model planner, each GPU holds both LLM and encoder model states, enabling encoder computation during LLM bubbles. The encoder and LLM model parallel plans are provided as input to the bubble scheduler, where Optimus selects parallel plans based on the output schedule with the shortest execution time.

Algorithm 1: Optimus workflow

```
1  Function Optimus(mllm):
2      encPlans, llmPlan = ModelPlanner(mllm)
3      bestLat, bestSchedule = +∞, None
4      for encPlan in encPlans do
5          schedule = BubbleScheduler(encPlan, llmPlan)
6          if schedule.lat < bestLat then
7              bestSchedule = schedule
8              bestLat = schedule.lat
9          end
10     end
11     return bestSchedule
```

Bubble Scheduler. Bubble scheduler is responsible for scheduling encoder computation into LLM bubbles. Given that the LLM training pipeline divides data into multiple microbatches, the scheduler schedules encoder computations on a per-microbatch basis and satisfies encoder-LLM data dependency at the microbatch level (addressing Challenge 2 in §3.1). In addition, the scheduler breaks down encoder computation into kernel granularity, to enable the utilization of sub-millisecond bubbles (TP bubbles) during LLM training (addressing Challenge 3 in §3.1). The current design is based on the 1F1B-interleaved pipeline schedule [20], but it is not tightly coupled to it and can be adapted to support other pipeline schedule strategies (as discussed in Section 6).

Optimus uses the model planner to devise parallel plans for encoders and LLMs. Subsequently, for each encoder parallel plan, Optimus utilizes the bubble scheduler to generate a schedule and estimate the latency. The latency estimation is based on offline profiling conducted during the model planner step, where we use the offline profiled encoder execution time and the LLM pipeline training timeline. Ultimately, Optimus selects the schedule with the shortest training time to schedule encoder computation into LLM bubbles. The workflow of Optimus is outlined in Algorithm 1.

4 Optimus Design

Section 4.1 describes how the model planner searches the parallel plans for the encoder, Section 4.2 details how the bubble scheduler exploits the coarse-grained and fine-grained bubbles through local scheduling, Section 4.3 discusses how

the bubble scheduler handles encoder-LLM data dependencies through global ordering, Section 4.4 designs the bubble scheduling in multi-branch encoder models, and Section 4.5 analyzes the memory consumption of the bubble scheduling algorithm.

4.1 Model Planner

The workflow of the model planner consists of searching encoders and LLM parallelism plans, colocating encoders and LLMs, ensuring memory constraints are met, and constructing separate microbatches for encoder and LLM pipelines.

Searching separate parallel plans. Initially, the planner determines the 3D parallelism plan $(DP_{llm}, PP_{llm}, TP_{llm})$ for the LLM backbone based on insights in Megatron-LM [20]. Subsequently, the planner enumerates potential 3D parallelism plans $(DP_{enc}, PP_{enc}, TP_{enc})$. To guarantee that multiple encoder models can be colocated with each LLM model, we enforce that PP_{enc} is a factor of PP_{llm} and TP_{enc} is a factor of TP_{llm}.

Colocating encoders and LLM. To guarantee that each GPU can perform encoder computations during LLM downtime, the model planner assigns encoder and LLM model states to every GPU. As illustrated in Figure 5, all GPUs contain model states for both the encoder (depicted in green) and the LLM (shown in red). Without such colocation, many GPUs would lack the necessary encoder model states to execute encoder computations.

Prune parallel plans based on memory constraint. Since the encoder and LLM stages are colocated on GPUs, we estimate the memory requirements for encoder and LLM model states, as well as LLM activations, based on the selected parallelism plan—drawing on the memory analysis in [14]. We omit encoder activations from the estimation due to their negligible memory footprint. Any plans that exceed GPU memory capacity are pruned early.

Constructing separate microbatches. Due to the different parallel plans for encoders and LLMs, there are $m = \frac{DPenc}{DPllm}$ times more encoder pipelines than LLM pipelines for a given set of GPUs (e.g. $m = 2$ in Figure 5). For GPUs belonging to the same LLM pipeline, there are m encoder pipelines colocated. Depending on the number of microbatches N_{mb} utilized in LLM pipeline training, the data from these N_{mb} microbatches needs to be distributed among these m encoder pipelines, where each encoder pipeline i handles forward and backward computations for N_{enc_i} microbatch data. The model planner enumerates possible ways to partition these N_{mb} microbatches among the m encoder pipelines. For instance, if there are 8 microbatches in the LLM training and $m = 2$ encoder pipelines, there are a total of 7 possible partitioning options, such as $[1,7]$, $[2,6]$, ..., $[7,1]$.

Algorithm 2: BubbleScheduler

```
1  Function BubbleScheduler(encPlan, llmPlan):
2      schedules = InitSchedule(encPlan, llmPlan)
3      dep = GetEncLLMDep(llmPlan)
4      bestLat, bestSchedule = +∞, None
5      for schedule in schedules do
6          schedule = OptimizeSchedule(schedule, dep,
             FWD)
7          schedule = OptimizeSchedule(schedule, dep,
             BWD)
8          if schedule.lat < bestLat then
9              bestSchedule = schedule
10             bestLat = schedule.lat
11         end
12     end
13     return bestSchedule
14 Function OptimizeSchedule(schedule, dep, mode):
15     while True do
16         encPPID = findCritical(schedule, mode)
17         newSchedule, success =
             ScheduleKernels(encPPID, schedule, mode)
18         if success and checkEncLLMDep(schedule, dep)
             then
19             schedule = newSchedule
20         else
21             return schedule
22         end
23     end
```

4.2 Bubble Scheduling

Although LLM bubbles in different GPUs have different start times and durations, there is one common pattern of LLM bubbles as shown in Figure 8. There is one single big bubble (the sum of DP all-gather bubble and PP-warm bubble) before any LLM computation starts, and one single big bubble (the sum of PP-cooldown bubble and reduce-scatter bubble) after all LLM computation finishes. And there are many small bubbles (PP bubbles and TP bubbles) [14, 20, 25] interleaved with LLM computation.

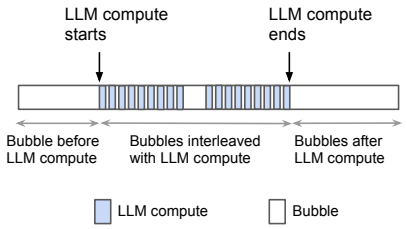


Figure 8: Bubble pattern of 3D parallelism.

The bubble scheduler, as described in Algorithm 2, initially engages in **coarse-grained bubble exploitation** by creating initial schedules that incorporate encoder computations within

the bubbles positioned before and after LLM computations (line 2). However, it's possible that these two bubbles may not allow sufficient time to complete all encoder computations, leading to some encoder computations being unscheduled within bubbles. To reduce the total training time, the bubble scheduler then executes **fine-grained bubble exploitation**. This involves refining the schedule by allocating encoder forward computations to the bubbles that alternate with LLM computations (line 7), followed by scheduling encoder backward computations to these same bubbles (line 8). The final output of the bubble scheduler is the schedule that achieves the shortest possible runtime.

Coarse-grained bubble exploitation. For each potential data partitioning approach, the bubble scheduler initializes the schedule by scheduling encoder forward operations to occur before LLM computations and encoder backward operations to occur after LLM computations. Figure 9 illustrates the initialized schedule when there are $m = 2$ encoder pipelines and the data partitioning approach is $[3,5]$, i.e., 3 microbatches are allocated to the first encoder pipeline and 5 to the second encoder pipeline.

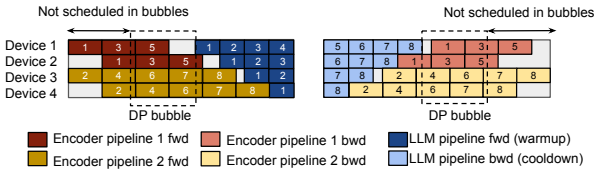


Figure 9: Bubble schedule initializes the schedule when the first encoder pipeline is allocated 3 microbatches and the second encoder pipeline is allocated 5 microbatches ($N_{mb} = 8$).

Fine-grained bubble exploitation. The `OptimizeSchedule` function (line 15 at Algorithm 2) refines the initial schedule through an iterative approach. Initially, the bubble scheduler employs `findCritical` to identify the encoder pipeline whose computation is on the critical path of the end-to-end MLLM training (line 17). Subsequently, it utilizes `ScheduleKernels` to allocate one microbatch of this encoder computation to bubbles interleaved with LLM computations (line 18). If enough bubbles are available to accommodate encoder computation and encoder-LLM data dependencies are satisfied (as discussed in §4.3), the bubble scheduler continues scheduling. Otherwise, it terminates and returns the best schedule found so far.

When optimizing the schedule for encoder forward computation (line 7 in Algorithm 2), `findCritical` identifies the encoder pipeline whose forward computation is critical. As shown in the left portion of Figure 10, encoder pipeline 2's forward computation (microbatch 8 forward) is initially on the critical path in the initial schedule. After successfully scheduling that microbatch forward to later bubbles, encoder pipeline 1 assumes the critical path position. This iterative process leads to a reduction in the end-to-end MLLM training time after each step. Similarly, encoder pipelines whose backward computation is critical are illustrated in the right portion of Figure 10. After each step, the bubble scheduler must verify if it still satisfies the encoder-LLM data dependency before proceeding with the next steps.

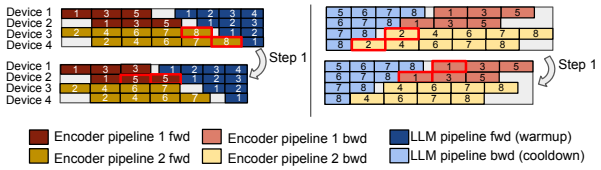


Figure 10: Find encoder pipeline that is on the critical path of end-to-end MLLM training (left: encoder forward on the critical path, right: encoder backward on the critical path).

When scheduling encoder computation to bubbles interleaved with LLM compute (`ScheduleKernels` at line 17), the bubble scheduler decomposes the encoder computation into kernel granularity and schedules these kernels based on the duration of the bubble. For each bubble, the bubble scheduler schedules multiple kernels while ensuring that the total execution time of these kernels is within the bubble duration. Additionally, the bubble scheduler must satisfy the encoder's internal data dependencies. As illustrated in Figure 11, device 1 holds the first two layers of the encoder, while device 2 holds the next two layers. When scheduling kernels during the forward pass, device 2 can only utilize bubbles that occur after device 1 completes its forward pass to execute encoder computation. For the forward computation, the bubble scheduler schedules encoder computation from upstream encoder pipeline stages to downstream encoder pipeline stages. Conversely, for backward computation, the bubble scheduler schedules encoder computation in the reverse order. While each encoder layer also includes communication kernels, the scheduler ensures that these kernels are not assigned to TP bubbles that occur during LLM communication. Instead, the scheduler identifies long-duration computation kernels within the LLM layers and overlaps them with encoder communication kernels. As the LLM and encoder layers alternately perform computation and communication tasks, they make efficient use of GPU bandwidth and Streaming Multiprocessors (SMs). This design strategy helps to minimize resource contention and improves overall GPU utilization [15].

Complexity. Our bubble scheduling algorithm has low complexity. Given n GPUs and the number of prime factors of n is n_p, the search space of parallel plans is $C^2_{n_p+1}$. The number of microbatch partitioning is $O(N_{mb}^{m-1})$. Hence, the complexity for scheduling bubbles is $O(C^2_{n_p+1} * N^m_{mb} * (F+B))$. In our experiments, it usually takes around several minutes to calculate the optimal schedule (see §5.3.2), which is also a one-time cost.

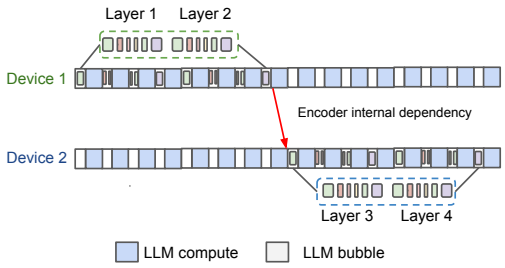


Figure 11: Scheduling encoder computation kernels needs to satisfy encoder internal dependencies.

4.3 Address Encoder-LLM dependency

Table 2: The list of symbols frequently used in the paper.

Symbol	Description
DP_{llm}	LLM Data Parallel Size
DP_{enc}	Encoder Data Parallel Size
N_{mb}	Number of microbatches in LLM training
MB_i	Encoder input data microbatch
A_i	LLM input activations for microbatch i
G_i	LLM output gradients for microbatch i
F_i	Forward dependency point for microbatch i
B_i	Backward dependency point for microbatch i

The model planner provides different parallel strategies for encoders and LLM backbone, including the number of microbatches, resulting in complex data dependencies both between and within the encoder and LLM. Also, the communication and computation of the encoder and LLM are executed by interleaving, and this may introduce additional pipeline bubbles, if not orchestrated effectively, intensifying the complexity of dependencies in the system.

The bubble scheduler addresses encoder-LLM dependencies at the microbatch level by examining the encoder-LLM forward and backward dependency points for each microbatch i. These dependency points, denoted as F_i and B_i respectively, represent the time when the LLM requires the corresponding activations A_i (output by the encoder) for forward propagation, and when the LLM generates the corresponding gradients G_i (input for the encoder) during backward propagation. To ensure the satisfaction of encoder-LLM dependencies, the bubble scheduler employs two functions: `GetEncLLMDep` (line 3 at Algorithm 2) and `CheckEncLLMDep` (line 19 at Algorithm 2), as described below.

`GetEncLLMDep` gets encoder-LLM forward and backward dependency points. Given that the interleaved 1F1B schedule [20] stands out as one of the most efficient pipeline schedules for LLM training, we delve into the specifics of the data dependency points F_i and B_i within this schedule. The top illustration in Figure 12 depicts an instance of the interleaved 1F1B schedule featuring two model chunks. Here, the forward dependency points denote the instances when the first pipeline stage (device 1) commences forward execution for the first model chunk (depicted in dark blue), while the backward dependency points signify the moments when the first pipeline stages (device 1) complete backward execution for the first model chunk (depicted in dark green).
We observe that deferring forward data dependency points for the last four microbatches ($F5$ through $F8$) is feasible without exerting any adverse effects on the overall pipeline latency. To accomplish this, we can adjust the number of warmup microbatches at each pipeline stage, as illustrated in the bottom portion of Figure 12. This adjustment enables the bubble scheduler to leverage bubbles during the phase transition from the warmup phase to the 1F1B-steady phase for scheduling encoder forward computation when optimizing initial schedules. `GetEncLLMDep` yields the adjusted forward and backward data dependency points for 1F1B interleave schedules.

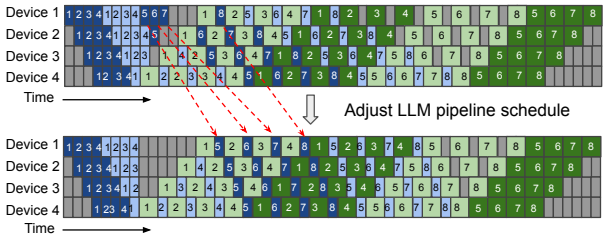


Figure 12: Interleaved 1F1B pipeline schedules before and after adjustment. The top figure shows the default interleaved 1F1B schedule in the Megatron-LM paper [20]. The bottom figure shows the interleaved 1F1B schedule after adjustment. In both schedules, each device is assigned 2 chunks. Dark colors show the first chunk, and light colors show the second chunk.

`CheckEncLLMDep` verifies the satisfaction of microbatch-level encoder-LLM dependencies. By considering the scheduled encoder computation into bubbles, the bubble scheduler estimates when the encoder finishes the forward pass for microbatches distributed over different encoder pipelines. The bubble scheduler sorts these finishing times in ascending order as EF_i (global ordering), representing when the encoder forward operation ends for microbatch i involved in LLM pipeline training. The forward dependency for encoder-LLM is considered met if the encoder completes its forward operation before the specified F_i timepoint ($EF_i \leq F_i$) for all microbatches ($i = 1...N_{mb}$). Similarly, the backward dependency is satisfied if the encoder's backward operation begins no earlier than the B_i timepoint ($EB_i \geq B_i$) for each microbatch ($i = 1...N_{mb}$). `CheckEncLLMDep` returns true when it confirms that both the forward and backward dependencies are successfully met. To illustrate this, Figure 13 provides an example of evaluating encoder-LLM dependency with two encoder pipelines, each handling four microbatches. The order in which the encoder completes its forward pass dictates

how the activations are used in the LLM pipeline: activations from encoder pipeline 1 are designated as the 1st, 3rd, 7th, and 8th microbatches, while activations from encoder pipeline 2 are used as the 2nd, 4th, 5th, and 6th microbatches. The bubble scheduler then verifies microbatch-level dependency by ensuring that each encoder's forward operation concludes before the start of the corresponding LLM forward pass and that each encoder's backward operation does not commence until after the LLM has ended, for each microbatch.

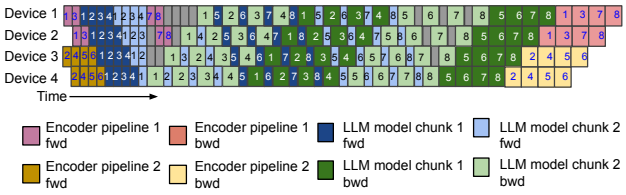


Figure 13: Illustration example of `checkEncLLMDep`.

When dependencies are satisfied, the bubble scheduler integrates necessary peer-to-peer (P2P) communications into the training schedule between the last stage of the encoder pipeline and the first stage of the LLM pipeline. For instance, if encoder pipeline j completes the forward pass for microbatch i, the scheduler will insert a P2P send (sending activations) at the last stage of encoder pipeline j and a P2P receive (receiving activations) at the first stage of the LLM pipeline. Similarly, when the LLM pipeline completes the backward pass for microbatch i, the scheduler adds a P2P send (sending gradients) at the first stage of the LLM pipeline and a P2P receive (receiving gradients) at the last stage of encoder pipeline j. In the scenario illustrated in Figure 13, where the training pipeline processes 8 microbatches, the scheduler inserts 8 pairs of P2P send-receive operations between devices 1 and 2 to manage the dependencies between encoder pipeline 1 and the LLM pipeline. These include 4 pairs for forward activation, send/receive, and 4 pairs for backward gradient send/receive. Similarly, another 8 pairs of P2P send-receive operations are inserted between devices 3 and 4 to handle the dependencies between encoder pipeline 2 and the LLM pipeline.

4.4 Multi-Branch Encoder Scheduling

To support MLLM with multiple encoders [6, 35], the model planner applies an encoder parallelism plan $(DP_{enc}, PP_{enc}, TP_{enc})$. Independently for each encoder. For pipeline parallelism, layers within each encoder are divided into PP_{enc} stages (as illustrated in Figure 14). Each layer of every encoder is then parallelized according to TP_{enc}. The bubble scheduler breaks down the layers of distinct encoders into kernel-level granularity and arranges their scheduling as if these kernels were part of a single encoder. This is because the encoders within MLLM operate independently, without any data dependencies between them.

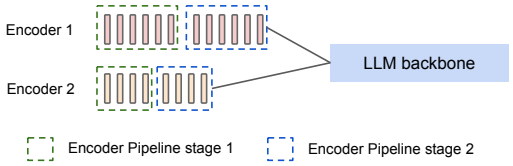


Figure 14: An example of model planner applying $PP_{enc} = 2$ to MLLM with two encoders.

4.5 Memory Analysis

When utilizing n_{gpu} GPUs for MLLM training, the model planner requires DP_{enc} replicated encoder model states and DP_{llm} replicated LLM model states based on parallel plans. Suppose the number of parameters in the encoder is ϕ_{enc} and the number of parameters in the LLM is ϕ_{llm}, with each parameter requiring k bytes of memory. The average GPU memory usage MEM_{model} for storing model states is calculated as follows:

$$MEM_{model} = \frac{k \cdot (DP_{enc}\phi_{enc} + DP_{llm}\phi_{llm})}{n_{gpu}}$$

In comparison to existing 3D parallel training solutions, where $DP_{enc} = DP_{llm}$, the estimated memory overhead $MEM_{overhead}$ can be expressed as:

$$MEM_{overhead} = \frac{k(DP_{enc} - DP_{llm})\phi_{enc}}{n_{gpu}}$$

With a larger value of DP_{enc}, there is a higher memory overhead due to more replicated encoder model states. However, this results in less complex encoder internal dependencies during scheduling (indicated by a smaller PP_{enc}). Model planner filters the encoder parallel plans based on the estimated memory usage MEM_{model}, ensuring adherence to GPU memory constraints. In practice, the memory overhead typically amounts to less than 12% in our evaluation (§5.3.1) because ϕ_{enc} is small (e.g., the largest vision encoder has 22 billion parameters [9]) and k is small (e.g., $k = 6$ when using $bf16$ parameters and $fp32$ gradients with distributed optimizer [1]).

5 Evaluation

We have developed Optimus based on the open-source Megatron-LM framework [1] and evaluated Optimus on training large-scale multimodal LLMs.

5.1 Methodology

Testbed. We conduct our experiments in a production training cluster with thousands of NVIDIA Hopper GPUs. Each GPU has 80GB of memory and 989 TFLOPS of computing performance. The intra-server connection is NVLink, and the inter-server connection is a high-bandwidth RDMA network.

MLLM models. We examine the performance of Optimus using various sizes of image encoders and LLM backbones.

The image encoders include three sizes: ViT-22B [9], ViT-11B, and ViT-5B, which are scaled-down versions of ViT-22B with smaller hidden sizes. For the language models, we employ two sizes: LLAMA-70B [31] and GPT-175B [5]. Appendix A includes detailed model configurations.

Baselines. We compare Optimus against three open-source MLLM training systems and one strawman baseline. DiffusionPipe [30] and DistTrain [37] are excluded from baseline comparison; the rationale is detailed in Section 6.

• PyTorch FSDP [38]: FSDP is a distributed data-parallel training module designed to scale PyTorch models across multiple GPUs with minimal code changes. It shards the model across GPUs, runs *All_Gather* to collect all shards from all ranks to recover the full parameter for forward and backward computation, and runs *Reduce_Scatter* to synchronize gradients.

• Alpa [39]: Alpa is a compiler system for distributed DL training that automatically generates parallel execution plans covering 3D parallelisms.

• Megatron-LM [20]: Megatron-LM is a state-of-the-art LLM training framework that integrates 3D parallelism techniques. Megatron-LM is designed for symmetric transformer models, and we place multimodal encoders in the pre-process in the first pipeline stage to adapt to MLLM training.

• Megatron-LM balanced: In this strawman method, we balance the layer partitioning among different pipeline stages with an interleaved 1F1B pipeline schedule. Considering the heterogeneity in MLLM submodules, we use a dynamic programming algorithm to assign different layers of submodules to pipeline stages and achieve approximately the same computation amount. The DP algorithm is a simplified version of Alpa's inter-operator DP algorithm and is included in Appendix B.

We use iteration time and Model Flops Utilization (MFU) [8] as the performance metrics. The reported performance numbers are averaged over 300 training iterations after a warm-up of 10 iterations. The detailed Megatron-LM configurations across experiments are included in Appendix D.

5.2 End-to-End Performance

5.2.1 Weak-Scaling Experiment

Experiment Setup. To study the ability to train large models, we follow common ML practice to scale the model size along with the number of GPUs. We evaluate the weak-scaling training performance of Optimus and baselines based on model configurations in Table 3.

Results. Figure 15 presents a comparison between Optimus and baseline methods across various sizes of MLLM. Optimus achieves a speedup of up to 1.22× compared to Megatron-LM and 1.18× compared to the Megatron-LM balanced. Alpa and FSDP face GPU out-of-memory (OOM) issues with these models.

Name	Encoder	LLM	#GPUs	Batch Size
Model A	ViT-11B	LLAMA-70B	64	32
Model B	ViT-22B	LLAMA-70B	128	64
Model C	ViT-11B	GPT-175B	256	128
Model D	ViT-22B	GPT-175B	512	256

Table 3: Weak-scaling MLLM configurations.

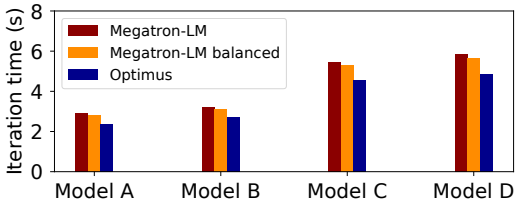


Figure 15: Weak-scaling experiment results (Alpa and FSDP are not shown in the figure because of OOM).

For our comparison with Alpa and FSDP, we crafted a modest MLLM that includes ViT-3B and GPT-11B, where Optimus demonstrates a 3.09× speedup compared to Alpa and a 15.1% improvement over FSDP, as detailed in Table 4. Further setup details can be found in Appendix C.

	Alpa	FSDP	Megatron-LM	Megatron-LM balanced	Optimus
Time (s)	8.61	3.20	3.42	3.04	**2.78**

Table 4: Training performance comparison with Alpa and FSDP.

5.2.2 Strong-Scaling Experiment

Experiment setup. We assess the strong-scaling training performance of Optimus and Megatron-based baselines using the ViT-22B+GPT-175B model. Following Megascale [13], we progressively increase the number of GPUs used (1536, 2048, and 3172) while keeping the batch size constant at 1536.

Results. Table 5 compares training performance between Optimus and Megatron-LM based baselines with an increasing number of GPUs. Optimus reduces iteration time by up to 21.3% compared to Megatron-LM, and by up to 20.5% compared to the Megatron-LM balanced. As the GPU count increases, Optimus demonstrates a greater speedup over baseline solutions. While Optimus maintains a stable MFU, baseline MFU drops at larger scales. This is expected—keeping the batch size constant while scaling up the GPU count increases the bubble ratio, allowing Optimus to schedule more encoder computation into LLM bubbles.

Batch Size	Method	GPUs	Iteration Time (s)	MFU	Aggregate PFlops/s
1536	Megatron-LM	1536	10.65	31.6%	480.7
		2048	8.26	30.6%	619.8
		3072	5.91	28.5%	866.3
	Megatron-LM balanced	1536	10.43	32.3%	490.9
		2048	8.06	31.3%	635.2
		3072	5.87	28.7%	872.2
	Optimus	1536	9.80	34.4%(**1.06**×)	522.4
		2048	7.29	34.6%(**1.11**×)	702.3
		3072	4.87	34.6%(**1.21**×)	1051.3

Table 5: Strong-scaling training performance of Optimus and baselines. The number in parentheses in the MFU column represents the speedup of Optimus compared to Megatron-LM balanced.

5.2.3 Multi-Encoder MLLM Experiment

Experiment setup. We assess the training performance of Optimus and Megatron-LM on multi-encoder MLLMs on 512 GPUs with batch size 256 (refer to Table 6). The Megatron-LM balanced baseline was excluded from this evaluation since its dynamic programming algorithm is designed to partition layers solely in MLLMs with a single encoder (linear model configuration).

Name	Encoder-1	Encoder-2	LLM
DualEnc(11B, 5B)	ViT-11B	ViT-5B	GPT-175B
DualEnc(22B, 5B)	ViT-22B	ViT-5B	GPT-175B
DualEnc(22B, 11B)	ViT-22B	ViT-11B	GPT-175B

Table 6: Multi-encoder MLLM configurations.

Results. Figure 16 illustrates the average iteration times of Optimus compared to the Megatron-LM. Optimus achieves a speedup of up to 1.25×, 1.26× and 1.27× on these MLLMs. This increased speedup by Optimus can be attributed to the Megatron-LM's approach of placing all encoders in the first pipeline stage, which leads to a more severe pipeline imbalance due to the larger total parameter count of the encoders.

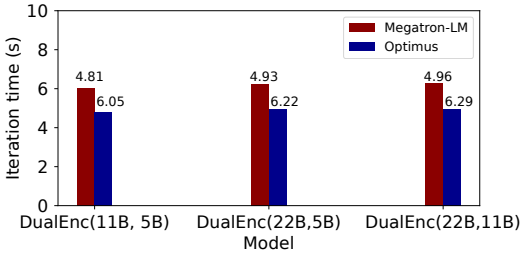


Figure 16: Training performance of Optimus and Megatron-LM on multi-encoder MLLMs.

5.3 Microbenchmarks

5.3.1 Optimus Memory

Experiment setup. We measure the GPU memory consumption of Optimus and baselines during the training of MLLMs of different sizes (listed in Table 3).

Results. As shown in Figure 17, Optimus presents a maximum GPU memory overhead of 12% when compared to the most memory-efficient baseline across various models. It is noted that Optimus uses less GPU memory than both baselines for model C and Megatron-LM balanced for model D. This discrepancy stems from the baseline's strategy of distributing computational loads across different pipeline stages, which can lead to memory imbalances due to varying hidden sizes in the encoder and LLM layers.

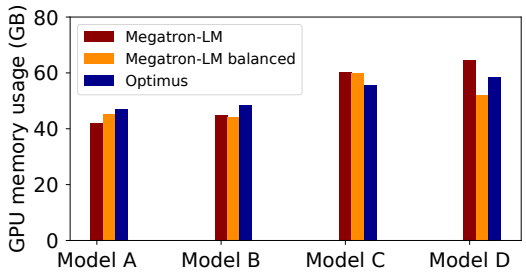


Figure 17: GPU memory usage of Optimus and Megatron-based baselines for MLLMs shown in Table 3.

5.3.2 Bubble Scheduler Algorithm

Experiment Setup. We executed the bubble scheduler algorithm on a single CPU core to compute the bubble schedule for training the ViT-22B+GPT-175B model with a global batch size of 1536 across an increasing number of GPUs (1536, 2048, and 3172), the same as the setting described in the strong-scaling experiment (Section 5.2.2). To evaluate the efficacy of the bubble scheduler algorithm, we developed a metric called *scheduling efficiency*, which quantifies the percentage of encoder computations that can be effectively scheduled within the LLM bubble. We report two efficiency metrics derived from simulations: Eff_{coarse}, observed when utilizing only coarse-grained bubble exploitation, and Eff_{fine}, observed when both coarse-grained and fine-grained bubble exploitations are activated (see §4.2). Additionally, we report

the runtime of the bubble scheduler algorithm.

Results. Table 7 illustrates that the bubble scheduler achieves higher scheduling efficiencies, Eff_{coarse} and Eff_{fine}, when operating with an increased number of GPUs for MLLM training. This improvement is attributed to the constant batch size of 1536, where the number of microbatches allocated to each LLM pipeline is reduced (32, 24, 16) as the number of GPUs increases (1536, 2048, 3172). Consequently, the LLM pipeline exhibits a higher bubble ratio due to the fixed durations of DP bubble and PP-warmup/PP-cooldown bubbles, while the total time for the end-to-end LLM pipeline decreases. Moreover, enabling fine-grained bubble exploitation can yield up to a 1.67× increase in efficiency compared to Eff_{coarse}. It is noted that the runtime of the bubble scheduler algorithm tends to decrease as the number of microbatches in the LLM pipeline reduces, due to fewer microbatch partitioning options (see algorithm complexity analysis in 4.2).

Settings	#Microbatch	Eff_{coarse}	Eff_{fine}	Runtime (s)
1536-GPU	32	34.3%	57.5%	322.2
2048-GPU	24	45.8%	69.3%	89.6
3172-GPU	16	68.7%	85.0%	15.1

Table 7: Scheduling efficiency and algorithm runtime of the bubble scheduler algorithm.

6 Discussion

MLLM training with frozen parameters. Optimus readily supports multi-stage training workflows commonly used in practice, such as those adopted by LLaVA [18]. While our paper focuses on the general setting where all parameters—including those of the encoder and LLM—are updated, Optimus can naturally adapt to stages where only adapters are trained. In such cases, Optimus schedules the encoder + adapter forward pass and the adapter's backward pass into the LLM pipeline bubbles, while skipping the encoder's backward computation due to frozen parameters. This maintains correct data dependencies and continues to utilize bubbles effectively.

Complex computation graph. Optimus focuses on the bubble scheduling on a typical MLLM model architecture, which consists of multimodal encoders followed by one LLM. We may further explore the bubble scheduling for complex MLLM computation graphs. A new partitioning algorithm is required to divide the computation graph into the backbone pipeline schedule and the bubble-filling workload. And the bubble scheduling algorithm of Optimus can be easily extended to the partitioned computation graph.

Other pipeline schedules. We use a widely used Megatron-LM interleaved 1F1B pipeline schedule for MLLM training. However, there exist other pipeline schedules (e.g., Chimera [16] and zero-bubble pipeline [23]) that may have superior performance in certain scenarios. The bubble scheduling of Optimus is orthogonal to these pipeline schedule optimizations, and Optimus can be applied to other pipeline schedules when the specific encoder-LLM dependency is analyzed and addressed.

Online scheduling. Our bubble scheduling algorithm simplifies execution by omitting the consideration of fluctuations in CUDA kernel runtime. We collect performance statistics, such as kernel execution time, to detect bubbles during a training step, assuming consistent behavior in future steps. However, deviations from predicted execution times can lead to suboptimal scheduling, with larger or altered pipeline bubbles. A possible solution is real-time performance monitoring to dynamically adjust the schedule.

Exclusion of DiffusionPipe and DistTrain as Baselines. We explicitly exclude DiffusionPipe [30] and DistTrain [37] from our set of baselines for the following reasons. DiffusionPipe is specifically designed for diffusion models and has been evaluated only on small-scale clusters ($\leq$64 GPUs). Its focus on a different model family and scale makes it ill-suited for comparison with Optimus, which targets large-scale multimodal LLM (MLLM) training across thousands of GPUs. DistTrain relies on a simplified model partitioning strategy that can result in significant pipeline imbalance when applied to the heterogeneous structures of MLLMs. Furthermore, DistTrain is not open source, preventing direct empirical comparisons.

7 Related works

Multi-modal training. Pytorch FSDP training [38] supports only data parallelism and is less efficient than hybrid parallel strategies. Alpa [39] automates parallelism for various models but falls short by not supporting state-of-the-art 1F1B-interleave pipeline parallelism [20] and requiring more memory than the optimized Megatron-LM framework [25], also missing opportunities in pipeline optimization due to its unified view of encoders and decoders. DistMM [12] provides solutions to orchestrating multiple parallel encoders, but it is designed for contrastive learning and overlooks the decoder, leaving a gap in comprehensive training efficiency. DiffusionPipe [30] and DistTrain [37] are two additional works on multi-modal training, each with limitations outlined in the previous section.

Bubble reducing. Previous efforts in reducing "bubbles" have approached the problem from various angles. The 1F1B-interleave pipeline [20] technique minimizes bubbles by chunking the model and alternating these chunks across different stages, whereas the Zero bubble pipeline [23] approach further granulates backward pass computations to eliminate bubbles. However, in practice, the Zero bubble pipeline schedule cannot completely remove all pipeline bubbles because it requires changes to the optimizer, which raises concerns about end-to-end model convergence. On the other hand, asynchronous tensor parallelism [26] and Google's overlapping

technique [32] aim to overlap tensor parallelism communication with computation, but are limited by specific hardware configurations and struggle to maintain full overlap as computing capabilities advance.

Bubble exploiting. Pipefisher [22] leverages pipeline bubbles across multiple training steps to complete the K-FAC, whereas our method operates within a single synchronized training step, focusing on immediate optimization. Hydro's Bubble Squeezer [11] utilizes GPT model bubbles for independent tasks like hyperparameter tuning, which can not enhance the performance of the training steps themselves. Bamboo [29] employs pipeline bubbles for redundant computations to mitigate the impact of preemption in training on volatile instances, based on the assumption that later pipeline stages host more layers, which often does not hold in large language model (LLM) training scenarios.

8 Conclusion

We present Optimus, a distributed MLLM training system that enables the scheduling of encoder computation within LLM bubbles to reduce end-to-end MLLM training time. To reduce GPU bubbles during MLLM training, Optimus partitions multimodal encoders and the LLM backbone, and schedules encoder computation in LLM bubbles. We search for the optimal parallelism plan for the encoders with the consideration of memory and computation resource constraints, which balances the encoder computation among GPUs for bubble filling. Optimus further employs a bubble scheduling algorithm to address encoder-LLM dependency and select the optimal schedule for filling kernel-level encoder computation into sub-millisecond LLM bubbles. Our extensive experiments demonstrate that Optimus can accelerate MLLM training by 20.5%-21.3% with ViT-22B and GPT-175B models over 3072 GPUs compared to baselines and significantly outperforms existing MLLM training systems by 20.3% on average.

References

[1] GitHub - NVIDIA/Megatron-LM: Ongoing research training transformer models at scale — github.com. https://github.com/NVIDIA/Megatron-LM. [Accessed 07-05-2024].

[2] Aishwarya Agrawal, Jiasen Lu, Stanislaw Antol, Margaret Mitchell, C. Lawrence Zitnick, Dhruv Batra, and Devi Parikh. Vqa: Visual question answering, 2016.

[3] Jean-Baptiste Alayrac, Jeff Donahue, Pauline Luc, Antoine Miech, Iain Barr, Yana Hasson, Karel Lenc, Arthur Mensch, Katherine Millican, Malcolm Reynolds, et al. Flamingo: a visual language model for few-shot learning. *Advances in neural information processing systems*, 35:23716–23736, 2022.

[4] Jinze Bai, Shuai Bai, Shusheng Yang, Shijie Wang, Sinan Tan, Peng Wang, Junyang Lin, Chang Zhou, and Jingren Zhou. Qwen-vl: A versatile vision-language model for understanding, localization, text reading, and beyond, 2023.

[5] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[6] Jiawei Chen and Chiu Man Ho. Mm-vit: Multi-modal video transformer for compressed video action recognition, 2021.

[7] Jun Chen, Deyao Zhu, Xiaoqian Shen, Xiang Li, Zechun Liu, Pengchuan Zhang, Raghuraman Krishnamoorthi, Vikas Chandra, Yunyang Xiong, and Mohamed Elhoseiny. Minigpt-v2: large language model as a unified interface for vision-language multi-task learning, 2023.

[8] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. Palm: Scaling language modeling with pathways. *Journal of Machine Learning Research*, 24(240):1–113, 2023.

[9] Mostafa Dehghani, Josip Djolonga, Basil Mustafa, Piotr Padlewski, Jonathan Heek, Justin Gilmer, Andreas Peter Steiner, Mathilde Caron, Robert Geirhos, Ibrahim Alabdulmohsin, et al. Scaling vision transformers to 22 billion parameters. In *International Conference on Machine Learning*, pages 7480–7512. PMLR, 2023.

[10] Danny Driess, Fei Xia, Mehdi S. M. Sajjadi, Corey Lynch, Aakanksha Chowdhery, Brian Ichter, Ayzaan Wahid, Jonathan Tompson, Quan Vuong, Tianhe Yu, Wenlong Huang, Yevgen Chebotar, Pierre Sermanet, Daniel Duckworth, Sergey Levine, Vincent Vanhoucke, Karol Hausman, Marc Toussaint, Klaus Greff, Andy Zeng, Igor Mordatch, and Pete Florence. Palm-e: An embodied multimodal language model, 2023.

[11] Qinghao Hu, Zhisheng Ye, Meng Zhang, Qiaoling Chen, Peng Sun, Yonggang Wen, and Tianwei Zhang. Hydro:{Surrogate-Based} hyperparameter tuning service in datacenters. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 757–777, 2023.

[12] Jun Huang, Zhen Zhang, Shuai Zheng, Feng Qin, and Yida Wang. Distmm: Accelerating distributed multimodal model training. In *NSDI 2024: 21st USENIX Symposium on Networked Systems Design and Implementation*, 2024.

[13] Ziheng Jiang, Haibin Lin, Yinmin Zhong, Qi Huang, Yangrui Chen, Zhi Zhang, Yanghua Peng, Xiang Li, Cong Xie, Shibiao Nong, et al. Megascale: Scaling large language model training to more than 10,000 gpus. *arXiv preprint arXiv:2402.15627*, 2024.

[14] Vijay Anand Korthikanti, Jared Casper, Sangkug Lym, Lawrence McAfee, Michael Andersch, Mohammad Shoeybi, and Bryan Catanzaro. Reducing activation recomputation in large transformer models. *Proceedings of Machine Learning and Systems*, 5, 2023.

[15] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, and Soumith Chintala. Pytorch distributed: Experiences on accelerating data parallel training, 2020.

[16] Shigang Li and Torsten Hoefler. Chimera: efficiently training large-scale neural networks with bidirectional pipelines. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14, 2021.

[17] Haotian Liu, Chunyuan Li, Yuheng Li, Bo Li, Yuanhan Zhang, Sheng Shen, and Yong Jae Lee. Llava-next: Improved reasoning, ocr, and world knowledge, January 2024.

[18] Haotian Liu, Chunyuan Li, Qingyang Wu, and Yong Jae Lee. Visual instruction tuning, 2023.

[19] Kenneth Marino, Mohammad Rastegari, Ali Farhadi, and Roozbeh Mottaghi. Ok-vqa: A visual question answering benchmark requiring external knowledge. In *Proceedings of the IEEE/cvf conference on computer vision and pattern recognition*, pages 3195–3204, 2019.

[20] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, et al. Efficient large-scale language model training on gpu clusters using megatron-lm. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15, 2021.

[21] OpenAI(2023). Gpt-4v(ision) system card, 2023.

[22] Kazuki Osawa, Shigang Li, and Torsten Hoefler. Pipefisher: Efficient training of large language models using pipelining and fisher information matrices. *Proceedings of Machine Learning and Systems*, 5, 2023.

[23] Penghui Qi, Xinyi Wan, Guangxing Huang, and Min Lin. Zero bubble pipeline parallelism, 2023.

[24] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. Zero: Memory optimizations toward training trillion parameter models. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–16. IEEE, 2020.

[25] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *CoRR*, abs/1909.08053, 2019.

[26] Siddharth Singh, Zack Sating, and Abhinav Bhatele. Communication-minimizing asynchronous tensor parallelism, 2023.

[27] Umut Sulubacak, Ozan Caglayan, Stig-Arne Grönroos, Aku Rouhe, Desmond Elliott, Lucia Specia, and Jörg Tiedemann. Multimodal machine translation through visuals and speech. *Machine Translation*, 34:97–147, 2020.

[28] Gemini Team, Rohan Anil, Sebastian Borgeaud, Yonghui Wu, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, Johan Schalkwyk, Andrew M Dai, Anja Hauth, et al. Gemini: a family of highly capable multimodal models. *arXiv preprint arXiv:2312.11805*, 2023.

[29] John Thorpe, Pengzhan Zhao, Jonathan Eyolfson, Yifan Qiao, Zhihao Jia, Minjia Zhang, Ravi Netravali, and Guoqing Harry Xu. Bamboo: Making preemptible instances resilient for affordable training of large {DNNs}. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 497–513, 2023.

[30] Ye Tian, Zhen Jia, Ziyue Luo, Yida Wang, and Chuan Wu. Diffusionpipe: Training large diffusion models with efficient pipelines. *Proceedings of Machine Learning and Systems*, 6:101–113, 2024.

[31] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023.

[32] Shibo Wang, Jinliang Wei, Amit Sabne, Andy Davis, Berkin Ilbeyi, Blake Hechtman, Dehao Chen, Karthik Srinivasa Murthy, Marcello Maggioni, Qiao Zhang, et al. Overlap communication with dependent computation via decomposition in large deep learning models. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, pages 93–106, 2022.

[33] xAI. Grok-1.5 vision preview, 2024.

[34] Shaowei Yao and Xiaojun Wan. Multimodal transformer for multimodal machine translation. In *Proceedings of*

the 58th annual meeting of the association for computational linguistics, pages 4346–4350, 2020.

[35] Zhewen Yu, Jin Wang, Liang-Chih Yu, and Xuejie Zhang. Dual-encoder transformers with cross-modal alignment for multimodal aspect-based sentiment analysis. In *Proceedings of the 2nd Conference of the Asia-Pacific Chapter of the Association for Computational Linguistics and the 12th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 414–423, 2022.

[36] Duzhen Zhang, Yahan Yu, Chenxing Li, Jiahua Dong, Dan Su, Chenhui Chu, and Dong Yu. Mm-llms: Recent advances in multimodal large language models. *arXiv preprint arXiv:2401.13601*, 2024.

[37] Zili Zhang, Yinmin Zhong, Ranchen Ming, Hanpeng Hu, Jianjian Sun, Zheng Ge, Yibo Zhu, and Xin Jin. Disttrain: Addressing model and data heterogeneity with disaggregated training for multimodal large language models. *arXiv preprint arXiv:2408.04275*, 2024.

[38] Yanli Zhao, Andrew Gu, Rohan Varma, Liang Luo, Chien-Chin Huang, Min Xu, Less Wright, Hamid Shojanazeri, Myle Ott, Sam Shleifer, et al. Pytorch fsdp: experiences on scaling fully sharded data parallel. *arXiv preprint arXiv:2304.11277*, 2023.

[39] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P Xing, et al. Alpa: Automating inter-and {Intra-Operator} parallelism for distributed deep learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 559–578, 2022.

[40] Deyao Zhu, Jun Chen, Xiaoqian Shen, Xiang Li, and Mohamed Elhoseiny. Minigpt-4: Enhancing vision-language understanding with advanced large language models. *arXiv preprint arXiv:2304.10592*, 2023.

A MLLM model configurations

Here we list all the the MLLM configurations used in the evaluation experiments of Optimus. ViT encoder configurations can be found in Table 8. LLM backbone configuration can be found in Table 9. In all experiments, we use sequence length 2048.

Table 8: Model configurations for ViT.

Models	Width	Depth	MLP dimension	Heads	Attention head dimension	Params
ViT-3B	2304	48	9216	18	128	3B
ViT-5B	3072	48	12288	24	128	5.5B
ViT-10B	4096	48	16384	32	128	10B
ViT-22B	6144	48	24576	48	128	22B

Table 9: Model configurations for LLM.

Models	Width	Depth	Heads	Attention-head dimension	Params
GPT-11B	3072	80	24	128	11B
LLAMA-70B	8192	80	64	128	70B
GPT-175B	12288	96	96	128	175B

B Megatron-LM balanced DP algorithm

We employ a dynamic programming (DP) algorithm to assign layers to different virtual stages for the Megatron 1F1B-interleaved schedule [20]. Following Alpa [39], the DP algorithm aims to minimize the latency of the slowest stage to reduce the end-to-end latency of the pipeline schedule. Given a pipeline parallel size of PP and V model chunks configured, the DP algorithm seeks to minimize the latency of the slowest virtual stage. It determines the optimal layer partition strategy that distributes layers across these $V \times PP$ virtual stages.

We define the function $F(l,m)$ to represent the maximum latency of a single virtual stage when the first m virtual stages. The computation begins with $F(l,1) = \sum_{i=1}^{i \le l} t_i$, where t_i denotes the execution time of the i-th layer (estimated based on FLOPs). The optimal structure of F is:

$$F(l,m) = \min_{j<l}(\max(F(j,m-1), \sum_{i=j+1}^{i \le l} t_i))$$

For a MLLM model with L layers, the layer partition strategy is determined by calculating $F(L, V \times PP)$ and recording the partitioning results to find the optimal solution. This ensures that the latency of the longest virtual stage, $F(L, V \times PP)$, is minimized across all virtual stages in a 1F1B-interleaved pipeline schedule. The dynamic programming algorithm described above is suitable for MLLM configurations with a single encoder, where encoder layers and LLM layers follow a linear structure. However, this DP algorithm does not apply to MLLM models that feature multiple encoders, as these encoders do not have data dependencies among each other.

C Comparison of Training Performance between Optimus, Alpa, and FSDP.

Experiment setup. To facilitate a comparison with Alpa and FSDP, we constructed a modest MLLM consisting of ViT-3B and GPT-11B, with specific configurations provided in Appendix A. We assessed the training performance using 8 NVIDIA A100 GPUs, as we encountered issues with the CUDA library when attempting to run Alpa on NVIDIA Hopper GPUs. The global batch size was set at 16, and the sequence length was 2048.

Results: According to Table 10, Optimus achieves a 3.09× speedup over Alpa and a 15.1% improvement over FSDP.

	Alpa	FSDP	Megatron-LM	Megatron-LM balanced	Optimus
Time (s)	8.61	3.20	3.42	3.04	**2.78**

Table 10: Training performance comparison with Alpa and FSDP

D Detailed configurations for Megatron-LM based baselines

D.1 Weak-scaling experiment

Table 11 shows detailed configurations for Megatron-LM based baselines in the weak scaling experiment.

Model	Method	GPUs	Microbatch size	Parallel configurations
Model A	Megatron-LM	64	2	(DP=2, PP=4, TP=8)
	Megatron-LM balanced			(DP=2, PP=4, TP=8, V=6)
Model B	Megatron-LM	128		(DP=4, PP=4, TP=8)
	Megatron-LM balanced			(DP=4, PP=4, TP=8, V=6)
Model C	Megatron-LM	256		(DP=4, PP=8, TP=8)
	Megatron-LM balanced			(DP=4, PP=8, TP=8, V=12)
Model D	Megatron-LM	512		(DP=8, PP=8, TP=8)
	Megatron-LM balanced			(DP=8, PP=8, TP=8, V=12)

Table 11: Megatron-LM based baseline configurations in the weak-scaling experiment

D.2 Strong-scaling experiment

Table 12 shows detailed configurations for Megatron-LM based baselines in the strong scaling experiment.

Model	Method	GPUs	Microbatch size	Parallel configurations
Model D	Megatron-LM	1536	2	(DP=24, PP=8, TP=8)
	Megatron-LM balanced			(DP=24, PP=8, TP=8, V=12)
	Megatron-LM	2048		(DP=32, PP=8, TP=8)
	Megatron-LM balanced			(DP=32, PP=8, TP=8, V=12)
	Megatron-LM	3072		(DP=48, PP=8, TP=8)
	Megatron-LM balanced			(DP=48, PP=8, TP=8, V=12)

Table 12: Megatron-LM based baseline configurations in the strong-scaling experiment

D.3 Multi-encoder MLLM experiment

In multi-encoder MLLM experiment, we use (DP=8, TP=8, PP=8) and configure microbatch size as 2 for Megatron-LM for all MLLM models.

Towards Optimal Rack-scale µs-level CPU Scheduling through In-Network Workload Shaping

Xudong Liao[1] Han Tian[2] Xinchen Wan[1] Chaoliang Zeng[3] Hao Wang[1] Junxue Zhang[2]
Mengyu Ma[4] Guyue Liu[5] Kai Chen[1]

[1]*iSING Lab, Hong Kong University of Science and Technology*
[2]*University of Science and Technology of China* [3]*BitIntelligence* [4]*Inspur* [5]*Peking University*

Abstract

Rack-scale CPU scheduling has emerged as a promising direction to accommodate the increasing demands for microsecond-level services. However, prior work suffers from both inaccurate load balancing in the network and complex yet sub-optimal scheduling within each server due primarily to its application-agnosticism. This paper presents Pallas, an application-aware rack-scale CPU scheduling solution for microsecond-level services with near-optimal performance. At the heart of Pallas is an *in-network workload shaping* to partition the workload into different shards, each of them preserving high homogeneity regarding the CPU demands. With the shaped workloads, Pallas then performs simple yet near-optimal inter-server load balancing and intra-server scheduling. We have fully implemented Pallas and our extensive experiments across various synthetic workloads and real-world applications demonstrate that Pallas significantly outperforms the state-of-the-art solution RackSched by delivering stably low tail latency and high throughput, reducing tail latency by 8.5× at medium load and as much as two orders of magnitude at high load, while gracefully handling long-term workload shifts and short-term transient bursts.

1 Introduction

Modern datacenters have deployed many user-facing applications for online services, such as key-value stores [9, 12, 13], interactive data analytics [55], search ranking & sorting [16], and function-as-a-service [19]. These services typically have strict service level objectives (SLOs) that require the applications to provide high throughput with *low tail latency* in the range of tens to hundreds of *microseconds* [15, 25]. To serve the increasing application demands, rack-scale CPU scheduling [53, 56, 85] have been proposed to scale beyond a single server to multiple servers within a rack. For example, as shown in Figure 1a, previous solution RackSched [85] leverages a programmable switch to perform inter-server load balancing and utilizes existing dataplane operating system, Shinjuku [45], for intra-server scheduling.

While RackSched enables cross-server scheduling, we find that it fails to provide consistently low tail latency for diverse workloads due to two key reasons. First, its Join-the-Shortest-Queue (JSQ) based inter-server load balancing method operates in an *application-agnostic* manner. Without the knowledge of the load each request generates at each server, it may cause *load imbalance* between servers. Second, this application-agnostic method also leads to assigning each server a mix of long and short requests, leaving the challenging *Head-of-line (HoL) blocking* problem to each intra-server scheduler. Unfortunately, our experiment in §2 shows that existing intra-server scheduling algorithms cannot provide near-optimal or even satisfactory low tail latency over heterogeneous workloads, resulting in ∼50× slowdown compared to the ideal.

This paper asks: *Can we design a rack-scale scheduler for microsecond-level services that can deliver consistently low tail latency and high throughput over diverse workloads?* To answer this question, we present Pallas, a novel approach for rack-scale CPU scheduling with near-optimal low tail latency and high throughput.

To design Pallas, we make an important observation: while it is challenging for each server to schedule mixed workloads, it is easier to schedule a *homogeneous* workload with near-optimal tail latency. Based on this observation, we propose to resolve both load imbalancing and HoL problem at network level through *workload shaping*. Our key idea is to proactively transform mixed workloads into groups of uniform ones, reducing the workload variance for servers. Then each server only needs to handle a shaped and uniform workload, which can be easily processed by simple scheduling algorithms to achieve optimal tail latency. Realizing this idea requires accurately estimating server load generated by each request and network devices capable of categorizing requests based on the estimation. Prior works have shown that existing data center application characteristics [29] and programmable switches [8] can achieve both requirements (§3). For instance, the service types can be inferred from the packet headers in key-value stores [9, 12] or in-memory databases [69, 75], and the CPU demands can be monitored and correlated with the

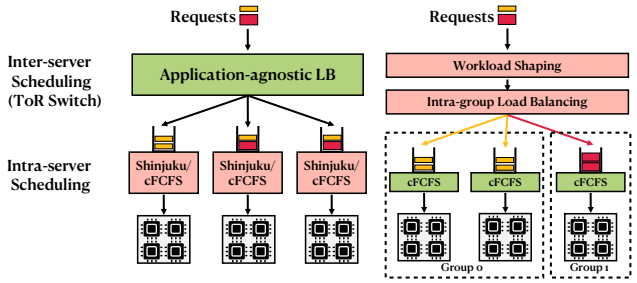
service types.

We fully explore *workload shaping*, aided by the above techniques, to design Pallas (Figure 1b), which decomposes the rack-scale scheduling into three levels : i) *workload-level shaping* runs at the ToR switch and partitions the entire workloads into different serving groups according to the estimated request CPU execution duration, with the goal to preserve group-level workload homogeneity; ii) group-level scheduling that also runs at the ToR switch and dispatches requests between servers within a group, ensuring equal load distribution among servers; and iii) server-level scheduling that runs at each server and steers uniform requests to CPU workers using the simple yet effective centralized First Come First Serve (cFCFS) policy. These three scheduling levels work in concert for Pallas to achieve the near-optimal tail latency (explained in §3).

Specifically, to translate the above 3-level architecture into a practical end-to-end system, Pallas addresses the following three challenges raised by real application characteristics and existing programmable switches:

- *Generating an effective scheduling policy that balances performance and server utilization* (§4.3): The scheduling policy, which decides how to divide workloads into groups, significantly impacts the ultimate performance. The key challenge lies in balancing workload homogeneity with server utilization: more groups might improve tail latency but lower server utilization, while fewer groups could optimize resource usage but potentially compromise tail latency. We address this challenge by designing a performance-oriented methodology to generate policy candidates with a tailored clustering algorithm and obtain the best performant workload shaping policy via micro-benchmarks.
- *Handling workload change gracefully via request bouncing* (§4.4): As the workload shifts over time, a static scheduling policy may become suboptimal. Simply replacing the policy is not sufficient and may result in temporal performance degradation during the transition. To provide consistently low tail latency, we dynamically adapt scheduling policies to runtime workloads and propose a request bouncing mechanism to resolve the temporal HoL blocking during the policy transition by prioritizing the precedence of short requests over long requests.
- *Mitigating transient workload burst via request cloning* (§4.5): To mitigate temporal server overload caused by bursty traffic, which happens in workload due to inherent variances, and avoid performance degradation, we propose a novel burst-aware request cloning which masks the tail latency inflation by cloning the overloaded requests to the under-utilized servers. Compared to the idea of traditional cycle stealing often used to mitigate intra-server burst [29], our approach offers a no-regret control and ensures the burst handling does not result in worse tail latency.

We have implemented the fully functional Pallas (§5) with commodity programmable switch, and evaluate it extensively



(a) Arch. of RackSched. (b) Arch. of Pallas.

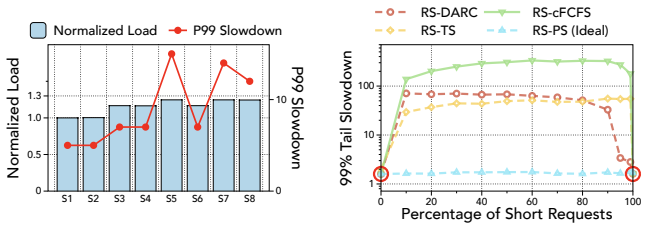
Figure 1: Rack-scale scheduling architectures: RackSched vs Pallas.

using a combination of synthetic workloads and real applications. Our results demonstrate that Pallas significantly outperforms RackSched [85] in terms of stably lower tail latency and higher throughput across various workloads (§6). For instance, compared to RackSched, Pallas reduces the tail latency by 8.5× and 5.5× in synthetic bimodal workload and real workload application RocksDB [13] at medium load, respectively. Besides, Pallas achieves 2.3× higher throughput than RackSched in the trimodal workload for the same tail latency objective, and reduces the tail latency by two orders of magnitude at high load in RocksDB. In addition, we show that Pallas can gracefully adapt to workload shifts and efficiently handle microsecond-level workload bursts. Our experiments further show that Pallas can preserve the performance superiority over other related solutions [56, 76, 82] (§6.5).

Similar to existing literature [53, 56, 76, 82, 85], Pallas focuses on stateless or replicated stateful services, including microservices, stream processing and replicated caches and storage. Supporting stateful services is beyond the scope of this paper. Besides, Pallas expects that the workloads' CPU demands can be statistically estimated based on historical data (see more details at §7).

Comparison with prior work. Compared to the DARC algorithm in Perséphone [29], which focus on intra-server isolation through CPU core partitioning, Pallas operates at rack scale, leveraging programmable switches to partition heterogeneous workloads before they reach servers. This architecture enables simplified and near-optimal scheduling both across and within servers. To the best of our knowledge, Pallas is the first to introduce application-aware workload shaping directly within the network switch, which differs from RackSched's application-agnostic load balancing, proactively preventing server-side HoL blocking rather than merely reacting to it after the damage is done. Moreover, while Pallas adopts request cloning to handle short-term workload bursts—a technique used by NetClone [53]—Pallas performs this selectively, within already-shaped workload groups, avoiding the excessive overhead of general-purpose cloning.

We open-source Pallas at `https://github.com/HKUST-SING/pallas`.



(a) Inter-server load imbalance. (b) Intra-server HoL blocking.

Figure 2: Existing rack-scale scheduler is inefficient.

2 Rack-scale Scheduling and Status Quo

Serving µs-level services at the rack scale requires the scheduler to provide both low-tail latency to serve requests and high throughput to utilize hundreds to thousands of cores in the rack. Recent work RackSched [85] is the representative rack-scale µs-level scheduler. As shown in Figure 1a, RackSched employs a two-level scheduling hierarchy: an inter-server scheduler in the top-of-rack (ToR) switch and an intra-server scheduler within each server. The inter-server scheduler tracks real-time server loads and selects a server for each request based on its current load. Then the intra-server scheduler assigns the request to its workers, leveraging an existing intra-server scheduling mechanism like Shinjuku [45] for high-dispersion workloads and centralized First-Come-First-Serve (cFCFS) for low-dispersion workloads.

Despite being promising, we have identified two main issues with RackSched that result in long tail latency: load imbalance between servers and head-of-line blocking at each server.

Load imbalance between servers. RackSched employs a Join-the-Shortest-Queue (JSQ)-based load balancing algorithm, directing requests to servers based solely on their queue lengths by having servers relay their load information to the inter-server scheduler. This *application-agnostic* method, lacking insight into the specific load each request imposes on servers, can inadvertently cause load imbalances between servers. This is particularly evident when different types of requests demand varying server resources. For instance, in Figure 1a, the first two servers have the same queue lengths. However, they manifest distinct loads due to the variability in the execution times of their respective requests.

Besides, it faces challenges in accurately capturing µs-level load dynamics. The inherent delay, approximately one RTT (around 10µs), required to obtain queue length data means that by the time the scheduler is updated, the server's actual load might have shifted. This results in the inter-server scheduler operating on potentially outdated and inaccurate load information. To understand this problem experimentally, we run RackSched to schedule requests among eight servers[1] and report their normalized perceived load and 99% tail slowdown in Figure 2a. We report the slowdown instead of latency as it better reflects the impact of long requests on shorter ones. The observation is that the load imbalance between servers can lead to a significant discrepancy in tail slowdown (∼3×).

Head-of-line blocking at each server. Given its application-agnostic load balancing strategy, the RachSched inter-server scheduler may distribute both long and short request types to each server. As a result, each server needs to address the challenges of head-of-line blocking, ensuring short requests are not blocked by longer ones. Unfortunately, even advanced intra-server schedulers [29, 45, 72] struggle to consistently offer near-optimal tail latency across diverse workloads.

To demonstrate this issue, we evaluated RackSched's performance using a Bimodal workload (short=1µs,long=100µs) with varying short-to-long request ratios. We integrated RackSched with four representative intra-sever scheduling solutions: i) Centralized first-come-first-serve (cFCFS) from ZygOS [72] and Shenango [68] which serves requests based on their arriving order; ii) Time Sharing (TS) from Shinjuku [45] which uses *preemption* to prioritize short requests [2]; iii) DARC from Perséphone [29] which reserves *dedicated cores* for short requests; and iv) Processor Sharing (PS) which is an *ideal yet impractical* one for comparison. We simulate it with 0.1µs preemption slice without *any* overhead. The simulation is based on ten servers, each equipped with twelve cores. All schemes are executed under a 95% load to highlight the performance discrepancy distinctly[3].

Figure 2b presents the results, with RS-X denoting RackSched combined with intra-server scheduling algorithm X. We find that all practical algorithms (cFCFS, TS, DARC) exhibit a notable performance gap compared to the *ideal yet impractical* algorithm RS-PS when the workload has a mix of short and long requests. This disparity becomes especially evident when the proportion of short requests ranges between 10% and 90%. By analyzing these solutions, we can see that cFCFS fails to address the HoL problem due to its inherent scheduling approach, processing earlier arriving long requests ahead of shorter ones. Despite TS and DARC using preemption or core reservation to mitigate the HoL problem, they also cannot fundamentally resolve it in practice and result in long tail latency.

To summarize, current rack-scale scheduling employs application-agnostic load balancing between servers, assigning a mix of short and long requests to each server. This approach essentially leaves the challenging HoL blocking problem to each intra-server scheduler, which unfortunately cannot be optimally resolved by any of the existing solutions.

[1] Refer to the `high` load case in §6.4.1 for the detailed experimental setting.

[2] RackSched uses Shinjuku (TS) with a 250µs preemption time slice by default. After adjustments, a 50µs time slice yielded satisfactory results for our setting. We simulate its context switching overhead as 1µs.

[3] Under light load, these schemes exhibit comparable performance, which is consistent with the results presented in [45].

3 Key Idea: In-Network Workload Shaping

Despite it is hard to optimally schedule mixed workloads within each server, we observed that it is easy to schedule *homogeneous* workloads. As shown in Figure 2b, when the proportion of short requests is 0% (purely long) or 100% (purely short), the three scheduling solutions—RS-cFCFS (theoretically optimal in these cases), RS-DARC, and RS-TS—achieve tail latencies comparable to the ideal RS-PS.

This observation led us to a novel approach: *in-network workload shaping*. By transforming mixed workloads into groups of homogeneous workloads at the network level, we can address the scheduling complexity within the network rather than leaving it to each server. This allows each server only needs to handle one type of request, either short or long, making it easy to achieve optimal performance.

There are two opportunities that support realizing the in-network shaping approach. i) As underscored by [29], cloud applications have the capability to transparently display request types within message headers. For example, Memcached [9] exposes request types at packet headers; Redis [12] specifies commands with a serialization protocol; Remote Procedure Calls (RPCs) with protobuf [2] can define message types. Notably, requests sharing the same type frequently demonstrate analogous processing behaviors and hence similar execution durations. Based on this observation, we can monitor the execution patterns of each type of request, enabling estimations of request execution durations based on these information. ii) Commodity datacenters are equipped with programmable switches, which provide programmable dataplane that can be leveraged to perform the workload estimation by extracting information from packet headers and hence implement advanced scheduling algorithms based on the estimated workload.

Our high-level architecture is shown Figure 1b. The programmable Top-of-Rack (ToR) switch stands central to this design, proactively partitioning mixed workloads into groups of uniform workloads based on their estimated running time. Each designated subset of servers then focuses on a specific workload group, with an intra-group load balancer ensuring even load distribution between servers. Then at each server, the intra-server scheduler handles requests using the simple and optimal cFCFS algorithm.

This architecture can effectively address the two limitations of existing rack-scale scheduling approaches to achieve near-optimal performance. i) *Accurate load balancing*: Within a given group, the actual load is proportional to the cumulative request numbers, enabling a more precise load balancing between servers. ii) *Eliminating HoL blocking*: After workload shaping, each server only needs to handle a specific type of request. This significantly eliminates HoL blocking problem for most servers. Only a few servers need to handle it when the workload cannot be entirely partitioned. By mitigating the HoL blocking, a simple cFCFS intra-server scheduler can be used to achieve optimal tail latencies.

Why this is theoretically near optimal. The CPU scheduling problem can be formalized as the classical `M/G/K` queuing problem [52]. While it can be theoretically resolved by JSQ/cFCFS and JSQ/PS for inter-server and intra-server scheduling, respectively, our findings in §2 have revealed that these approaches are impractical or inefficient for µs-level latency. Fortunately, `M/G/K` can be decomposed into multiple `M/D/K′` subproblems, each exhibiting deterministic workload. Wierman et al. [80] shows that FCFS is tail-optimal for deterministically light-tailed workload within each server, while round-robin is near-optimal for balancing the load between servers within each `M/D/K′` model. This convinces us that a near-optimal framework can be achieved through in-network workload shaping, which distributes homogeneous and deterministic workload for each server.

4 Pallas Design

In this section, we first discuss the design challenges of Pallas, then overview its workflow, and finally describe its design details for addressing these challenges.

4.1 Design Challenges

Realizing the idea of *in-network workload shaping* into a practical system presents three outstanding technical challenges:

- **How to generate an effective workload shaping policy?** Creating an effective scheduling policy requires i) accurately estimating the workload, ii) partitioning the mixed workload into groups based on estimated server loads, and iii) mapping these groups to available servers. The key challenge lies in balancing workload homogeneity with server utilization: more groups might improve tail latency but lower server utilization, while fewer groups could optimize resource usage but potentially compromise tail latency.
- **How to gracefully handle long-term workload changes?** A scheduling policy may become suboptimal as the workload distribution shifts over time. Simply replacing the scheduling policy to accommodate workload dynamics is not sufficient and may result in temporal tail latency degradation. For instance, new scheduling policy that transitions servers from groups serving long requests to the groups for short requests may incur HoL blocking.
- **How to efficiently handle short-term transient bursts?** Pallas encounters limitations in capturing microsecond-level dynamics due to the coarse-grained control granularity of long-term workload changes handling, especially in the face of transient partial workload surges due to its inherent variances. As a result, it is imperative for Pallas to incorporate a mechanism to mitigate the impact of these transient spikes.

4.2 Overview

Figure 3 shows the high-level workflow of Pallas. Pallas runs the inter-server scheduler as the ToR switch and the intra-

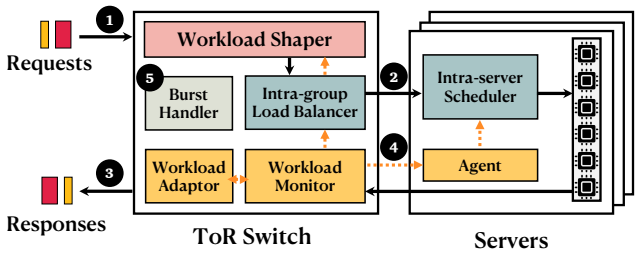


Figure 3: Pallas overview.

	Request Type	Time(μs)
RocksDB	GET(10)	12
	SCAN(5000)	650
TPC-C	Payment	5
	OrderStatus	6
	NewOrder	16
	Delivery	62
	StockLevel	74

Table 1: Workload examples of RocksDB and TPC-C with request types and corresponding execution times. GET(10) and SCAN(5000) denote retrievals of 10 values and scanning 5000 keys-value pairs, respectively.

server scheduler at each server. Central to the key workload shaper, Pallas composes two other components: intra-group load balancer and intra-server scheduler. Each component implements a portion of the scheduling policy, which is initially generated offline (§4.3).

Workload Shaper. The workload shaper proactively transforms mixed workloads into groups of homogeneous workloads at the ToR switch based on the request types shown in the packets (step ❶). This stems from the observation that the same type of requests have similar execution times on servers. For example, Table 1 shows the execution times of RocksDB and TPC-C workloads we profiled (details in §6.1), which are also used in [29, 45]. Clearly, the GET(10) and SCAN(5000) requests of RocksDB can be explicitly clustered into two groups. On the other hand, TPC-C necessitates a more nuanced grouping approach (e.g., NewOrder could be its own group or merged with Payment and OrderStatus requests).

Based on the well-shaped workloads, Pallas is able to perform *simple yet near-optimal* intra-group load balancing and intra-server scheduling according to the initially generated policies.

- **Intra-group load balancer.** After determining the group for each workload request, the load balancer is responsible for distributing the requests to servers within the group. To minimize the impact of load imbalance between servers, it leverages the weighted-round-robin (WRR) algorithm to distribute request to each server based on their provisioned computation capacity for the group (step ❷). WRR optimally weighted-balances the perceived number of requests for servers within a group, aligning with their provisioned computation capacity. We note that it is less practical for the intra-group load balancer to directly determine the CPU core for each request. The reason stems from the challenge of accurately tracking the states of hundreds to thousands of CPU cores within a rack at microsecond-scale [53, 85]. The core state requires one RTT to update in the ToR switch, which may introduce stale information and lead to inefficient scheduling decisions.
- **Intra-server scheduler.** After group-level load balancing, servers dispatch the well-shaped, load-balanced request streams to different CPU cores with optimal scheduling policy. Specifically, they use the simple and optimal cFCFS algorithm to serve the uniform workload. If the workload cannot be fully partitioned, a few servers that are allocated to handle multiple groups use the DARC algorithm [29]. Note that the workload monitoring and estimation in the Pallas framework are conducted at ToR switch. Therefore, the server employing the DARC algorithm is exempted from redundant workload estimations.

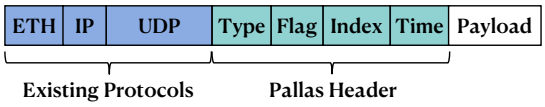


Figure 4: Packet format of Pallas.

Workload monitoring and change adaptation. To facilitate workload estimation, Pallas server embeds the actual execution duration of each request in the header of its corresponding response packet. The *workload monitor* extracts this information and updates the dataplane record registers when processing the response packets (step ❸). The control plane of the switch executes a *workload adaptor* to periodically read information from workload monitor, craft the appropriate scheduling policies (step ❹), and provision resources if it deduces significant long-term workload shifts (§4.4). When Pallas detects a bursty arrival of requests, the dataplane *burst handler* implements a no-regret burst reaction mechanism (step ❺) to mitigate the burst on time (§4.5).

Packet format. Figure 4 depicts the packet header format of Pallas's request and response. It consists of four major fields: 8-bit `Type`, 8-bit `Flag`, 32-bit `Index` and 32-bit `Time`, which are used for request grouping in workload shaping, request cloning in burst handling (§4.5), redundant replies filtering, and actual execution time of request recording in workload monitoring, respectively. Note that currently Pallas assumes each request is contained in one packet. Supporting requests with multiple packets is not the primary objective of this paper. However, this can be realized by incorporating RackSched's request affinity mechanism [85] into Pallas, which is feasible given Pallas's low hardware footprint as shown in §5.

4.3 Scheduling Policy Generation

In this subsection, we first describe how Pallas generates workload shaping policy, which is the core of Pallas to shape

uniform workload, and then elaborate how Pallas leverages it to provision resources and formalize the scheduling policies.

Balancing homogeneity and server utilization. Pallas should derive an effective request group mapping policy that preserves high group-level workload homogeneity with high system utilization. Drawing from the TPC-C workloads example[4] (Table 1), we observe that request types such as *Payment*, *OrderStatus*, and *NewOrder* exhibit comparable CPU demands. If one were to prioritize sheer request homogeneity, these types should be allocated to disjoint serving groups. However, such a configuration fragments system resources and may inadvertently result in more system idling, inducing system inefficiencies. Conversely, incorporating these request types within a single serving group could compromise group-level homogeneity, which may potentially result in a discernible performance deviation from the optimal. In summary, more groups improve homogeneity (potentially lowering latency) but risk lower utilization; fewer groups improve utilization but risk HoL blocking. Therefore, how to find a group mapping policy that achieves high intra-group homogeneity with high system efficiency remains a problem.

Since it is intractable to find the optimal policy analytically, we resort to a performance-oriented greedy mechanism to derive a solution inspired by [30]. This mechanism encompasses two primary stages: i) generation of group mapping and ii) selection of candidates. We describe each stage below by visualizing policy generation with TPC-C workload in Figure 5.

Group mapping candidates generation (step ⓐ). We employ a clustering algorithm to generate several group mapping candidates. With the monitored workload information, the algorithm derives group mapping policies by aggregating similar request types considering both the monitored CPU demands and their proportions across all requests with customized k-means algorithm.

Optimal candidate selection (step ⓑ). Upon obtaining all candidates from the preceding procedures, we aim to identify a policy that optimizes tail latency without compromising system utilization. To this end, we employ a performance-oriented strategy to obtain the most appropriate group mapping policy, with the focus on an empirical metric, i.e., 99th percentile latency, that matches our performance objective. Specifically, we compare the group mapping policies through offline simulations and evaluate their 99th percentile latencies. Consequently, the policy with the best objective value is selected (e.g., *Final Mapping* Table in Figure 5). We summarize these two steps in Algorithm 1 in Appendix §A.

Resource provisioning (step ⓒ). Upon determining the definitive request grouping policy, we compute the cumulative CPU demands for each group and allocate the requisite CPU resources to each serving group according to the reported computing capacity of each server (e.g., `server indices` in *Intra-group LB Policy* Table). We denote r as a class of requests with the same type, e.g., GET, E_r as the average execution time of r, ϕ_r as the ratio of r among all requests, $\mathcal{D}_r$ and $\mathcal{D}_g$ as the CPU demand of r and serving group g, respectively. We compute the CPU demand of each serving group and allocate resources as follows:

$$\begin{aligned} \mathcal{D}_r &= \frac{E_r \times \phi_r}{\sum_i E_i \times \phi_i} \\ \mathcal{D}_g &= \sum_{r \in g} \mathcal{D}_r \\ \mathcal{R}_g &= \frac{\mathcal{D}_g}{\sum_{g \in G} \mathcal{D}_g} \times \mathcal{R}_{total}, \end{aligned} \tag{1}$$

where G indicates total serving groups, $\mathcal{R}_g$ denotes the CPU resources allocated to serving group g, and $\mathcal{R}_{total}$ represents the total CPU resources provisioned to the application. It is noteworthy that Pallas allocates resources at fine-grained core-level. Hence, certain groups might be allocated with fractional server numbers, for instance, 12 cores in 1.2 servers for group 0 in the example. In such scenarios, a few servers will cater to multiple groups. To temper the scheduling intricacies throughout the system, our resource provisioning strategy guarantees that only a small number of servers serve multiple groups via gathering them to the same servers as much as possible.

Scheduling policy generation (step ⓒ and ⓓ). Upon finalizing the request grouping policy and resource allocation for each serving group, Pallas configures the policies in switch and each server, thereby establishing the system's request scheduling policy. When serving groups occupy multiple servers, the workload monitor formulates an intra-group load balancing policy and updates the load balancing table with weight values which correlate to the provisioned cores for each server within that group (e.g., `Round-Robin Weights` in *Intra-group LB Policy* Table). Concurrently, Pallas sends the intra-server scheduling policies (e.g., *Intra-server Sched. Policy* Table) to the agents located on each server. These transmitted policies encapsulate the scheduling algorithms and the relevant per-serving group CPU cores reservations, if any, especially in scenarios where a server accommodates multiple serving groups. In this way, the intra-server scheduler is able to conduct scheduling following the relayed scheduling policy from the agent.

Workload variance handling. We note that Pallas is resilient to the execution time dispersion as found in real workloads via the use of Exponential Weighted Moving Average (EWMA) for tracking execution times. EWMA inherently tolerates dispersion by capturing the central tendency, providing a stable average for grouping decisions even with variance in individual requests. As long as the EWMA-derived average execution times between different types of requests are sufficiently distinguishable, Pallas can effectively separate them into appropriate groups. Furthermore, the policy generation

[4] We use the TPC-C workload to examplify a workload with n-modal distribution.

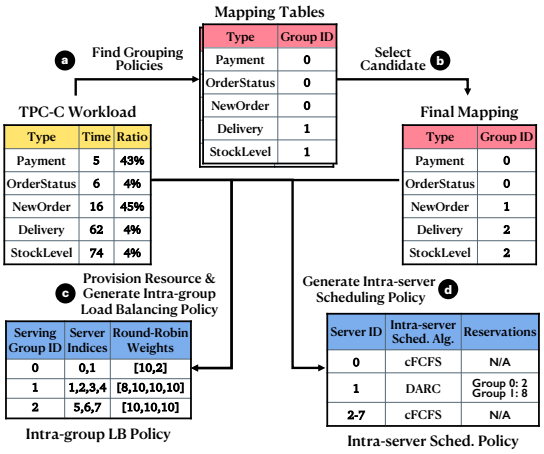


Figure 5: Example of generating a scheduling policy for the TPC-C workload across eight servers, each equipped with 10 cores.

process itself evaluates grouping candidates based on overall empirical performance evaluated by simulation. This process selects groupings that perform well despite the expected level of intra-group variance inherent in the EWMA-based assignment.

4.4 Long-term Workload Change Adaptation

As workloads may vary over time, Pallas is tailored to accommodate workload changes and recalibrate its generated scheduling policies dynamically. It monitors real-time workload metrics in the switch control plane by accessing the state registers within the switch dataplane.

Workload change detection. Leveraging the detailed per-request metrics, i.e., average request execution time and total number of requests within a monitor interval, Pallas workload adaptor periodically assesses the new aggregated demand of each serving group in comparison to its allocated CPU resources. In particular, it examines the discrepancy between the allocated CPU resources and the real-time demand as inferred from the latest monitored workload data for each serving group, with the resource allocation methodology in §4.3. It then computes the cumulative demand discrepancy across all serving groups as the resource provision process in §4.3. If this discrepancy exceeds a predetermined threshold δ, Pallas activates a workload change reaction mechanism. Currently, the workload adaptor works at a 10ms granularity to capture relatively long-term workload changes. In our evaluation, we set δ to 10 and study its sensitivity in the Appendix §D.3.

Workload change reaction. In response to the detected workload changes, Pallas adapts its scheduling policies swiftly. Specifically, Pallas re-invokes the workload estimation methods outlined in §4.3 to formulate new policies for both inter-server and intra-server scheduling. Subsequent to this, the related tables within the switch dataplane are updated. It is

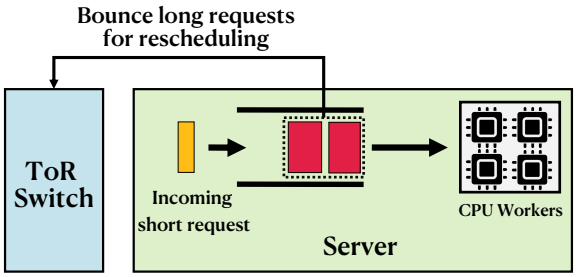


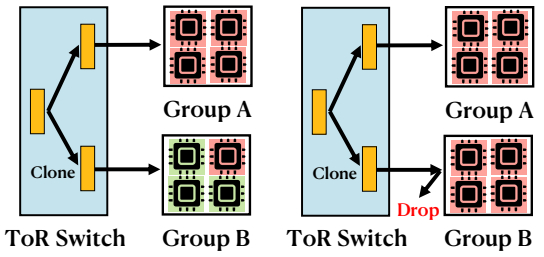
Figure 6: Pallas request bouncing.

worthy to highlight that, in order to temper potential disruptions during the policy reconfiguration phase which makes the system unstable and hence hurts the tail latency, Pallas adopts an *incremental update approach*. This includes caching the previously enforced scheduling policies in workload monitor and determines their deviation from the newly devised policies. Pallas then selectively updates only those segments of the policy tables that should be changed, thereby curtailing the reconfiguration duration. Additionally, Pallas restricts the number of servers impacted by the reconfiguration, via preserving the established roles of servers in relation to their serving groups and making adjustments only when necessary.

Request bouncing. Pallas aims to eliminate HoL blocking through in-network workload shaping. To accommodate workload dynamics, Pallas periodically revises its scheduling polices. This periodic adjustment leads to the reallocation of end-host servers across various serving groups. Consequently, during such reconfiguration phases, short requests within Pallas might experience temporary HoL blocking. This scenario arises especially when new servers, which previously catered to long requests, are allocated to serve short requests. As such, the HoL dilemma can resurface during switch reconfiguration phases. To address this, we introduce a novel *request bouncing* mechanism, ensuring Pallas navigates reconfiguration phases seamlessly. When a Pallas server receives new scheduling decisions, such as serving groups from long requests to short requests, we prioritize the processing of incoming short requests, thereby proactively emptying existing queues of long requests that might induce HoL blocking. To achieve this, the Pallas server compares its current serving groups against newly arriving ones. As shown in Figure 6, if the new request should take precedence by exhibiting relatively shorter CPU demands, Pallas purges the current queue, bouncing the requests back to the switch for re-scheduling. This approach stems from the rationale that the network transmission time introduces less slowdown penalty on long requests, especially when compared with the performance degradation by HoL blocking that would impose on shorter requests.

4.5 Short-term Transient Burst Handling

Although workload estimation captures long-term workload shifts, it faces challenges in handling sub-millisecond-level transient surges [60], as it might make the system temporarily



(a) Successful cloning. (b) Failed cloning.

Figure 7: Pallas request cloning. The yellow box represents the request of group A. The red core and green core represent the busy CPU and idle CPU, respectively.

overloaded and hence hurt tail latency. To remedy this, we introduce a mechanism that detects bursts and an innovative burst reaction method that mitigates such transient fluctuations. We note that Pallas specifically tailors this mechanism to manage bursts associated with particular request types or serving groups, attributing to their inherent variances[5].

Burst detection. Formally, we define a burst inspired by the Heavy Change Detection mechanisms [74, 81]. Consider a fixed time interval τ, we denote the number of requests for group g received by the switch during the one time interval τ_t to be $\mathbb{C}_{g,t}$. Pallas stipulates that a burst occurs when the real-time observed $\mathbb{C}_{g,t}$ exceeds a threshold $\mathbb{C}_{thres}$. Once a burst is detected, Pallas enters the *reaction mode* to react to the burst. The burst detection threshold $\mathbb{C}_{thres}$ represents the upper limit of requests from group g that can be processed by the entire allocated computational resources without inducing queuing. Let N denote the number of group g's requests a single core can process within the τ duration. The threshold $\mathbb{C}_{thres}$ can be formulated as $\sum_{i \in \mathcal{R}_g} N_i$. The value of N_i can either be determined through offline profiling or dynamically estimated by the *agent* resided within each server. To achieve prompt burst detection, we set the time interval τ at 20µs.

Why not server-level cycle stealing? While cycle stealing [37] proves effective for handling burst within an individual server in Perséphone [29], resulting from its ability to determine that the stolen cores are idle, this effectiveness diminishes at the rack-scale. The challenge emerges from the difficulties of accurately determining a server's idle state at scale for microservices [53]. In this case, the request that is redirected to other group servers via cycle stealing may even deliver worse tail latency when the destination server is busy. Consequently, we aim at a no-regret burst reaction mechanism. It should ensure that the ultimate performance never deteriorates compared to scenarios without any burst reaction, a guarantee that cycle stealing cannot provide.

Burst reaction. Upon detecting a burst at the serving group level in terms of request rates, Pallas employs a best-effort request cloning technique to handle transient spikes. Pallas assumes that a burst typically does not occur across the entire workload. Thus, when a specific serving group grapples with a burst, other groups temporarily experience underutilization. As a result, Pallas strategically clones [53] the incoming requests to idle groups for burst handling. The total cloning number hinges on the burst's intensity. Specifically, once Pallas enters the *clone mode*, it clones requests from the overloaded group to the underutilized ones, determining the total number of requests to be cloned as $\mathbb{C}_{clone} = \mathbb{C}_{g,t} - \mathbb{C}_{thres}$. When a request is selected for cloning, Pallas clones it and then dispatches the original request to its serving group and the cloned request to the group which exhibits the lowest degree of burst (Figure 7a). The `req.flag.clone` of the original request and the cloned one are set to 1 and 2, respectively. To avoid the potential HoL blocking caused by cloning long requests to short request groups, the server receiving the cloned request (`req.flag.clone` = 2) will directly drop it if it has no idle CPU cores (Figure 7b). To avoid redundant processing of multiple responses for the cloned requests, Pallas's switch dataplane filters out the slower responses with the globally unique request ID (`req.index`). Besides, experiments in §6.4.2 show that Pallas's cloning does not sacrifice sustainable throughput, as it only handles the bursty portion of requests. The overall request scheduling procedures are summarized in Algorithm 2 in Appendix §B for reference.

5 Implementation

We have implemented a fully functional Pallas prototype, including the inter-server scheduler in programmable switch, intra-server schedulers and paired agents in each server and request clients. 1) The inter-server scheduler within switch dataplane is written in P4 [18] and compiled to switch ASIC with P4 Studio [4]. Pallas uses 2.8% SRAM and 10.4% Stateful ALUs of Tofino ASIC resources. Pallas dataplane composes 763 lines of P4 code. The control plane was implemented with 1067 lines of python code, which uses the switch SDK to read aggregated statistics from data plane registers and update the policy tables if required. More details of dataplane implementation are presented in the Appendix §C. 2) The server is built atop Perséphone [29]. We have extended Perséphone to support the packet header and functionality of Pallas for measuring request actual execution time, and encode this time into Pallas responses for workload estimation. Besides, we have implemented the agent to relay intra-server scheduling policies, and request bouncing mechanism to protect system performance during switch reconfiguration of policy tables. 3) The client is open-loop, implemented in C, and utilizes DPDK for high-speed user-space networking. It can generate Pallas requests at high rate and measure the throughput and latency for each request accurately.

6 Evaluation

We evaluate Pallas to answer the following key questions:

- **How does Pallas compare against state-of-the-art sys-**

[5] Addressing the burst happening on the entire workload level usually requires techniques such as system overload control [20, 21] or resource auto-scaling [48, 65], which is beyond the scope of this paper.

	Synthetic Workloads												Real World Applications			
	Bimodal				Trimodal			TPC-C					RocksDB			
	Normal		Port										Normal		Port	
Type	S	L	S	L	S	M	L	P	OS	NO	D	SL	GET	SCAN	GET	SCAN
Time (µs)	10	100	10	100	5	50	500	5	6	16	62	74	12	650	12	650
Ratio	90%	10%	50%	50%	33.3%	33.3%	33.3%	43%	4%	45%	4%	4%	90%	10%	50%	50%

Table 2: Workloads with dispersion. In Bimodal and Trimodal workloads, S, M, L represent short, medium, and long request, respectively. In TPC-C workload, P, OS, NO, D, and SL represent Payment, OrderStatus, NewOrder, Delivery, and StockLevel, respectively.

tems in static workloads? We show that Pallas achieves near-optimal tail latency in statically synthetic workloads (§6.2.1) and real-workload applications (§6.2.2), which significantly outperforms RackSched. For example, Pallas reduces the tail latency by 8.5× and 5.5× compared to RackSched in the Bimodal distribution and RocksDB experiments at medium load, respectively. Specifically, Pallas reduces the tail latency by two orders of magnitude compared to RackSched in the Port RocksDB experiment at high load. We also show that Pallas delivers low tail latency under light-tailed workloads (Appendix §D.1) and outperforms other solutions (§6.5), including R2P2 [56], Draconis [76] and Horus [82].

- **How does Pallas respond to workload changes?** We demonstrate that Pallas can agilely respond to workload changes and gracefully handle system reconfiguration (§6.3), maintaining the stable tail latency in both synthetic and real-world workloads.
- **How effective is Pallas in terms of its components and scalability?** First, we show that workload shaping significantly helps load balancing and intra-server scheduling (§6.4.1). Besides, we demonstrate that our request bouncing (§6.4.2) and burst handling mechanisms (§6.4.3) can both effectively mitigate the performance degradation under workload dynamics. We also show that Pallas can almost scale out linearly with more servers without compromising performance in Appendix §D.2.

6.1 Experimental Setup

Testbed. The experiments are performed on a testbed of ten machines connected by a Intel Tofino switch. Each machine is equipped with two 12-core CPU (Intel Xeon(R) CPU E5-2630 v2, 2.60GHz), 64GB memory, and one NVIDIA ConnectX-4 100G NIC. Each machine runs Ubuntu 18.04 with Linux kernel 5.4.100. We use eight machines as servers to process requests and two machines as clients to generate requests, which is the similar setting in RackSched. Both clients and servers runs DPDK 22.11.1 [5] for high-performance packet processing. All servers use 10 worker threads running on dedicated CPU cores. To improve system stability and reduce jitter, we disable TurboBoost, C-states, and CPU frequency scaling as recommended by [29, 68].

System compared. We compare the performance of Pallas against RackSched [85]. We note that RackSched leverages Shinjuku [45] as the default intra-server scheduler. However, as Shinjuku only supports Intel NICs [45, 63, 85], we replace it with Perséphone (DARC) within RackSched's framework for intra-server scheduling, since our motivation experiment (Figure 2b) has shown that DARC delivers comparable performance compared to TS (Shinjuku) for mixed workloads. We have extended Perséphone to support RackSched packet header and encode queue length in its response for JSQ-based load balancing. For ease of reference in the subsequent experiments, we name the system as *RS-DARC* or simply *RS*.

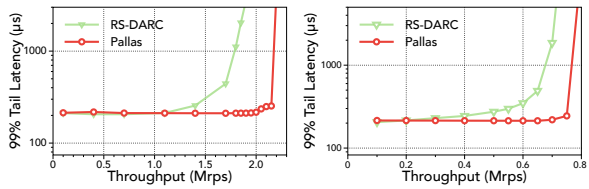
Workloads. We use a combination of synthetic and real world application workloads to evaluate Pallas, following the similar settings in recent works [29, 45, 72, 85]. By default, both request and response contain one packet. In the context of synthetic workloads, servers conduct the fake work by spinning the CPU for the specified number of cycles. For the real world application workloads, we harness RocksDB [13], an in-memory key-value store, as the experimental platform. We summarize the workloads in Table 2. Notably, we profile *TPC-C* [11] transactions with silo [75], an in-memory database, and evaluate it as a synthetic workload to showcase how Pallas performs on the n-modal workload. Additionally, we craft dynamic workloads characterized by fluctuating request distributions. This setting aims to elucidate Pallas's adaptive capacities and response mechanisms when confronted with variations in workload patterns. Note that Pallas targets stateless or replicated stateful services. The profiled TPC-C transactions are used primarily as a source of realistic, multi-modal execution times for synthetic load generation to test shaping effectiveness. In addition, the evaluation on RocksDB uses an in-memory setup with read-only GET/SCAN requests.

Evaluated metrics. We adjust the request rates (request per second (rps)) from the clients to modulate the system load, and report the 99% tail latency, measured in microseconds, corresponding to each specific rps adjustment.

6.2 Static Workloads

6.2.1 Synthetic Workloads

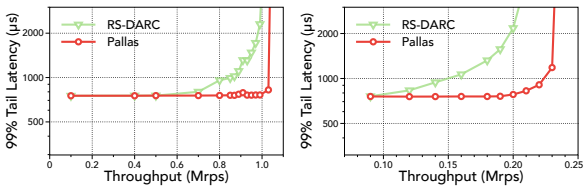
Figure 8 compares Pallas to RS-DARC for three service time distributions. Note that the y-axis is log-scaled for good visibility. Overall Pallas achieves significantly better tail latency than RS-DARC. For example, as shown in Figure 8a, the tail latency of RS-DARC starts to increase when system load is above 1.5Mrps, and its tail latency reaches 1112µs when



(a) Normal Bimodal (b) Port Bimodal

(c) Trimodal (d) TPC-C

Figure 8: Experimental results for synthetic workloads.



(a) Normal RocksDB (low) (b) Port RocksDB (high)

(c) GET Latency (high) (d) SCAN Latency (high)

Figure 9: Experimental results for RocksDB.

the load reaches 1.8Mrps, which is 5.2× higher than Pallas (211µs) at the same load. In contrast, Pallas maintains a stably low tail latency (∼200µs) before the load reaches 2.15Mrps. In specific, Pallas reduces the latency by 16× when the system load is 1.9Mrps. For the 250µs tail latency objective, Pallas achieves 1.5× higher throughput. The reason of Pallas's significant performance improvement is twofold. First, by employing the workload shaping, Pallas adeptly steers short and long requests to different subsets of servers for intra-group scheduling. Therefore, under this workload configuration, seven servers exclusively process a singular type of requests (either short or long), where the simple yet efficient cFCFS scheduler is utilized to achieve optimal tail latency. This entirely addresses the HoL blocking problem. Only one residual server works in the hybrid mode, accommodating both short and long requests. Second, the approach to load balancing also undergoes refinement through workload shaping. In Pallas, intra-group scheduling only requires a weighted balancing of request numbers among servers. This mechanism demonstrates better load balancing than the queue-length based solution adopted in RackSched.

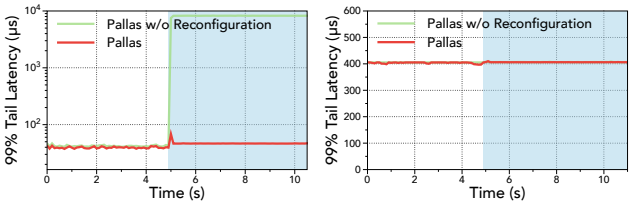
Figure 8b depicts the results for a more challenging Port Bimodal workloads, which represents the workloads characterized by an even split between short and long requests. We find that Pallas maintains a consistently low tail latency until the system reaches its saturation point. In specific, Pallas reduces the tail latency by 8.5× compared to RS-DARC at a load of 0.7Mrps. For the 300µs tail latency objective, Pallas achieves 1.4× higher throughput. Figure 8c illustrates the performance of Pallas under a trimodal workload. The results show that Pallas delivers 2.0× higher throughput than RS-DARC when targeting a tail latency objective of 1200µs. Furthermore, at a system load of 0.18Mrps, Pallas manages to reduce the tail latency by a factor of 2.3× compared to RS-DARC.

Finally, we evaluate Pallas with the profiled TPC-C workload. In this context, the best performant grouping policy is: the Payment and OrderStatus transactions are cohesive in one group, the Delivery and StockLevel transactions form another group, and the NewOrder transaction represents solely the third group. The results are shown in Figure 8d. Notably, when targeting a tail latency objective of 200µs, Pallas delivers as much as 2.5Mrps high throughput, which is 1.7× superior to that of RS-DARC.

6.2.2 Real Application: RocksDB

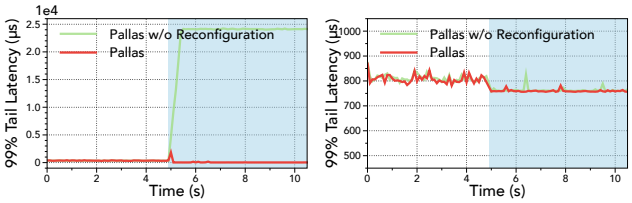
In this section, we showcase the performance gain of Pallas when applied to real-world applications RocksDB [13]. Adopting the configuration presented in [45, 85], we configure the database to operate on an in-memory file (/tmpfs/) for processing database transactions. Our evaluation incorporates two distinct request types: the GET request, which retrieves 10 key-value pairs with a median service time of 12µs, and the SCAN request, designed to scan 5000 key-value pairs with a median service time of 650µs. Figure 9a shows the performance under a workload mix of 90% GET and 10% SCAN requests. We observe that Pallas adeptly handles system loads up to 1Mrps without increasing tail latency.

Figure 9b presents the performance results for RocksDB workloads with an equal mix (50-50) of GET and SCAN requests. We find that Pallas maintains stable tail latency until the system load reaches 0.21Mrps, achieving a 5.5× latency reduction when compared to RS-DARC. Notably, Pallas achieves as much as two orders of magnitude lower latency at the 0.22Mrps and 0.23Mrps load. For a more granular analysis, we break down the results of GET and SCAN requests under the port RocksDB workload, respectively, as depicted in Figure 9c and Figure 9d. Owing to the application-agnostic nature of inter-server scheduling in RS-DARC, each server continues to handle heterogeneous workloads. This necessitates the reservation of at least one CPU core for short requests exclusively, resulting in an over-provisioning of GET requests and an under-provisioning of SCAN requests. This imbalance results in a rapid surge in the tail latency of SCAN requests under RS-DARC as the system load escalates. In contrast, Pallas optimizes resource provisioning for each request



(a) Short requests (b) Long requests

Figure 10: Results on the changing synthetic workload.



(a) GET requests (b) SCAN requests

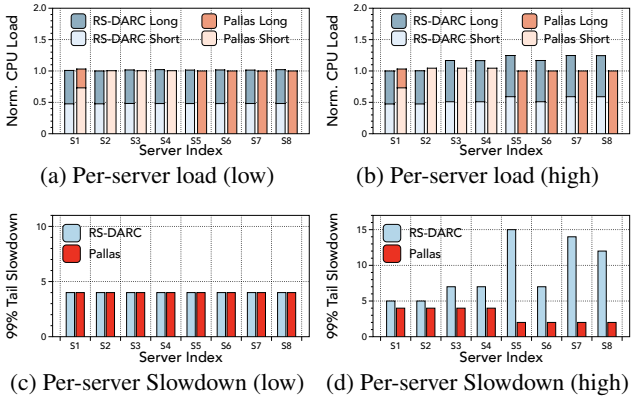
Figure 11: Results on the changing RocksDB workload.

type at rack-scale, yielding a far more efficient and effective allocation. Consequently, Pallas significantly curtails the tail latency of SCAN requests compared to RS-DARC, and yet without compromising the tail latency performance of GET requests.

6.3 Dynamic Workloads

Synthetic workloads. We initiate by dispatching requests that follow the Port Bimodal (50%-10µs, 50%-100µs) distribution to the system, subsequently transitioning the workload distribution to Normal Bimodal (90%-10µs, 10%-100µs). Throughout this experiment, we maintain the system load at approximately 80%. The results are shown in Figure 10, where the shaded region signifies the period of new workload. The performances of long requests under these two schemes remain relatively stable due to their decreasing resource demands (Figure 10b). For short requests, we observe that Pallas without policies reconfiguration cannot adapt to workload shifts and hence results in explicit tail latency inflation (Figure 10a). On the other hand, during this transition phase, the Pallas workload estimator discerns the workload pattern and promptly updates both the inter-server scheduling policies and intra-server scheduling policies. Further, as the workload changes, the proportion of short requests surges. Consequently, there is a noticeable spike in the 99% tail latency for Pallas's short requests, which is attributed to system reconfiguration and transient overloads. Nevertheless, the temporal overload can be mitigated by the burst handling mechanism, which takes effect before the control plane detects the workload change by cloning increasing short requests to the group for long requests to reduce its latency. Besides, our dedicated request bouncing strategy reconciles this latency elevation. These two mechanisms work in concert to ensure that Pallas accommodates the changing workload gracefully.

RocksDB. We adopt a similar setting to assess Pallas's performance on dynamic RocksDB workloads. The request distribution transitions from Port RocksDB (50% GET, 50% SCAN) to Normal RocksDB (90% GET, 10% SCAN). The results are depicted in Figure 11. We observe that Pallas swiftly adapts to the workload changes, ensuring consistent tail latency. Its counterpart of Pallas without workload adaptation delivers a significant worse performance, as current scheduling policies are largely inefficient for GET requests. During the workload transition phase, there is also a slight surge. However, the tail latency inflation is effectively bounded by the mechanisms of burst handling and request bouncing.



(a) Per-server load (low) (b) Per-server load (high)

(c) Per-server Slowdown (low) (d) Per-server Slowdown (high)

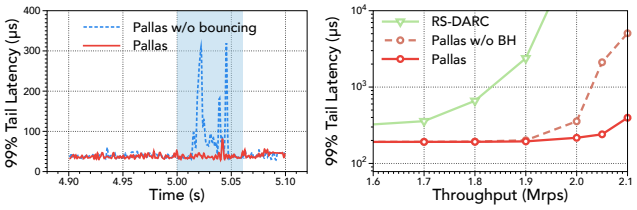
Figure 12: Comparison of per-server perceived load and performance between Pallas and RS-DARC.

6.4 Pallas Deep Dive

6.4.1 Effectiveness of Workload Shaper

To delve deeper into why Pallas significantly outperforms RackSched in terms of tail latency, we analyze the perceived request load and resultant performance slowdown for each server during the experiments under Normal Bimodal workload in §6.2.1. This analysis is conducted at two distinct system loads: 0.05Mrps (referred to as 'low') and 1.8Mrps (referred to as 'high').

In the low load scenario (first column), it is evident from Figure 12a that both RackSched and Pallas manage to achieve commendable inter-server load balancing. Consequently, their overall slowdowns are comparable, as seen in Figure 12c. However, when we examine RS-DARC at high load, a clear disparity in server load balance emerges. As Figure 12b shows, servers S1 and S2 in RS-DARC handle noticeably fewer requests than their counterparts, while S5 is scheduled to handle more requests. This imbalance is attributed to the application-agnostic nature and outdated load information used by the JSQ-based load balancer in RackSched, resulting in the divergent slowdown performance. Additionally, each server in RackSched continues to process a mixture of short and long requests, necessitating the use of complex algorithm (DARC) for ongoing workload monitoring and core allocation adjustments for different type of requests. This dynamic adjustment, however, still culminates in suboptimal performance. In con-



(a) Request bouncing (b) Burst handling

Figure 13: Effectiveness of Pallas's components.

trast, Pallas showcases impressive load balance across servers, even at high load, as the results shown in Figure 12b. This balance is achieved through in-network workload shaping and nearly optimal intra-group scheduling. Consequently, servers S2-S8 in Pallas exclusively handle a single request type, allowing them to employ cFCFS to optimize tail latency and slowdown. For instance, when examining server S5, we find that Pallas reduces the slowdown by 7× compared to RS-DARC, as highlighted in Figure 12d.

6.4.2 Request Bouncing

We offer a granular breakdown of the experiments under dynamic Bimodal workloads in §6.3, with results presented in Figure 13a. We observe that the bouncing policy adeptly facilitates Pallas's management of workload shifts, particularly in curtailing the tail latency of short requests. For instance, the fluctuations of tail latency for short requests in Pallas are effectively tempered in comparison to the system without bouncing mechanism.

6.4.3 Burst Reaction

To showcase the efficacy of Pallas's burst handling mechanism, we evaluate it under a challenging bursty workload comprised of 90% short requests of 10µs and 10% long requests of 90µs. The arrival pattern for short requests adheres to the bursty gamma distribution with a standard deviation of 5, whereas the long requests conform to a standard Poisson arrival pattern. In this scenario, both short and long requests are allocated to two distinct serving groups, with each group being provisioned with 4 servers, offering 40 cores cumulatively. After profiling the maximum throughput under this configuration, we configure the burst detection threshold C_{thresh} as 42 for the group of short requests. The tail latency under varying system loads is illustrated in Figure 13b. Our results reveal that both Pallas and its variant, i.e., with the Burst Handling (BH) disabled, significantly outperform RS-DARC in terms of reduced latency. Furthermore, Pallas reduces the tail latency by an average factor of 6.1× compared to its variant when the load surpasses 1.9Mrps. This is because Pallas clones the bursty short requests to the group for long ones, which effectively masks the latency inflation and hence mitigates the performance degradation of short requests.

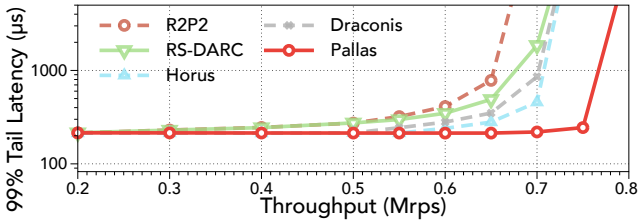


Figure 14: Comparison with other solutions.

6.5 Comparison with Other Solutions

Several other solutions aim to improve rack-scale scheduling by leveraging in-network support. R2P2 [56] introduces a join-bounded-shortest-queue (JBSQ) policy for dynamically distributing requests across servers. Horus [82] enhances scheduling accuracy by proactively monitoring server states and loads via network switches. Draconis [76] is the latest proposal for optimizing latency for µs-level services, which implements centralized first-come-first-served (cFCFS) scheduling directly within the switches. Figure 14 compares Pallas with them under a port bimodal workload. We find that Pallas achieves superior performance, delivering lower tail latency and greater sustainable throughput. The reason is that they distribute mixed workloads to each server, where HoL blocking may still occur. However, Pallas eliminates it through its efficient in-network workload shaping and thus delivers the near-optimal performance.

7 Discussion

Target workloads and applications. Pallas can support CPU-intensive stateless or replicated stateful services, which is consistent with recent proposals [53, 56, 76, 82, 85]. Example applications comprise in-memory databases, replicated caches and storages, function-as-a-service and replicated inference services for machine learning. Pallas currently does not support stateful services, as it is less likely to replicate them within a rack and scheduling may not be required. Pallas may be extended to support stateful applications by implementing a *sticky* policy, similar to [56], which distributes stateful operations to determined master nodes to maintain consistency.

Workload practicality. Pallas assumes that the workloads can expose statistical characteristic in terms of the CPU demands, which is practical in many workloads [11, 14, 22]. This characteristic helps Pallas to generate effective workload shaping policies. As a result, Pallas's performance gain may diminish if the workloads are highly dynamic and unpredictable. We acknowledge that it is Pallas's limitation to rely on the correlation between request types and execution time, which is a simplified assumption. For more complex workloads whose request execution time cannot be related to their types alone, such as Lucene [1], we environ Pallas with a more intelligent workload monitoring mechanism, e.g., using a machine learning model, to estimate the request execution time based on the necessary information at packet headers

exposed by the programmers.

Scaling to multiple racks. Currently, Pallas is designed within a single rack, as a modern rack can already provide hundreds of cores and is expected to pack thousands of cores [3, 7], which is sufficient for many services [85]. We further note that *in-network workload shaping* of Pallas can naturally be extended to multiple racks or even the entire datacenter in a hierarchical manner, e.g., leveraging both core switches and ToR switches. Realizing this requires efficient collaboration between core switches and ToR switches to enforce a holistic workload shaping policy across racks. Meanwhile, managing heterogeneous servers across racks and handling more frequent workload changes and spikes may also present challenges. We leave it as future work.

Optimization of intra-server scheduling. By design, the majority of servers in Pallas are expected to employ the simple and efficient cFCFS scheduling algorithm. A natural enhancement would be to transition this scheduling algorithm to hardware [6, 10, 63], which promises to curtail scheduling overheads substantially and improve intra-server scalability.

Resource multiplexing and placement constraints. In the context of CPU scheduling, Pallas's workload shaping strategy may not be efficient regarding the resource multiplexing of other resources, e.g., memory and disks. Extending Pallas to schedule microsecond-level services with multi-resource multiplexing is an interesting future direction. Moreover, Pallas's workload shaping strategy essentially implies the placement constraints of requests. Therefore, system operators and designers may utilize the outcome of Pallas's workload shaping to guide specific optimizations for different types of requests [58, 79].

Deployment considerations. Pallas is incrementally deployable. Its major components are implemented in the programmable switch, without requiring complex modifications to end hosts, and may even simplify server-side designs. Pallas uses reserved UDP ports to trigger its logic, allowing it to co-exist with other applications. Practical deployment of Pallas requires access to programmable switches with sufficient dataplane and control plane resources. As shown in the implementation (§5), Pallas consumes modest hardware resources, suggesting the feasibility on modern switch hardware. Integrating Pallas into production environments may also require coordination with existing dataplane programs [33] and control plane infrastructure, particularly to support multiple concurrent applications. We leave it as future work.

8 Related work

Rack-scale scheduling for µs-level services. There exist some solutions to optimize the tail latency for rack-scale µs-level services [53, 56, 76, 82, 85]. However, all of them may still suffer from the load imbalance and the HoL blocking under workloads with highly diverse execution times. Pallas optimizes the tail latency through an entirely different way: instead of scheduling requests, Pallas poses the concepts of workload shaping and schedules the workload to each server, which significantly reduces the complexities of both load balancing and intra-server scheduling and improves the scheduling quality. NetClone [53] employs general request cloning as a primary mechanism for tail latency optimization. Pallas utilizes cloning differently and more strategically: as a conditional mechanism (§4.5) integrated within its shaping framework, and triggered only for detected bursts within specific and pre-shaped workload groups. This makes Pallas's cloning a precise tool to maintain stability for already optimized workloads and maintains resource efficiency.

Intra-server optimizations. Several intra-server designs have been proposed to reduce latency and improve throughput by introducing optimized network stacks [5, 24, 42, 49, 67], designing dataplane operating systems [17, 23, 29, 31, 32, 39, 41, 45, 46, 50, 62, 66, 71, 72, 84] and performing hardware-based optimizations [38, 40, 51, 63, 78]. For example, Perséphone [29] focus on mitigating HoL blocking within a single server by isolating requests with different execution times to separate CPU cores. These optimizations are orthogonal to Pallas and can be integrated in it to further reduce latency and improve throughput.

Job scheduling. There is a long line of research on the job scheduling at cluster-level [26, 27, 28, 34, 35, 36, 47, 70, 77]. These systems target second-level jobs and therefore allow for complex scheduling algorithms to achieve good performance. However, Pallas is designed for microsecond-level services which requires the scheduler to make simple and effective scheduling decisions at line speed.

Programmable networks. Programmable switches enable the network to perform more complex operations to improve datacenter applications [43, 44, 54, 57, 59, 61, 64, 73, 83]. To the best of our knowledge, Pallas is the first to design application-aware in-network scheduling for microsecond-level services with significant performance improvements.

9 Conclusion

This paper presented Pallas, a new solution to schedule µs-level services at rack-scale. Pallas leverages the core idea of workload shaping by proactively transforming mixed workloads into uniform ones within the network and performs simple yet near-optimal inter-server load balancing and intra-server CPU scheduling. Our evaluation demonstrates that Pallas significantly outperforms the state-of-the-art solution in terms of both throughput and tail latency.

Acknowledgments

We thank our shepherd Gyuyeong Kim and anonymous ATC reviewers for their constructive feedback. This work is supported in part by Hong Kong RGC TRS T41-603/20R, GRF 16213621, ITC ACCESS, NSFC 62402407, National Natural Science Fund for the Excellent Young Scientists Fund Program (Overseas). Kai Chen is the corresponding author.

References

[1] Apache lucene. https://lucene.apache.org/core/2_9_4/queryparsersyntax.html.

[2] Google. protocol buffers - google's data interchange format. https://github.com/protocolbuffers/protobuf.

[3] Hp the machine. https://www.hpl.hp.com/research/systems-research/themachine/.

[4] Intel barefoot p4 studio. https://www.intel.com/content/www/us/en/products/details/network-io/intelligent-fabric-processors/p4-studio.html.

[5] Intel data plane development kit (dpdk). https://www.dpdk.org/.

[6] Intel ethernet flow director. https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/intel-ethernet-flow-director.pdf.

[7] Intel rack scale design. https://www.intel.com/content/www/us/en/architecture-and-technology/rack-scale-design-overview.html.

[8] Intel tofino series. https://www.intel.com/content/www/us/en/products/details/network-io/intelligent-fabric-processors/tofino.html.

[9] Memcached key-value store. https://memcached.org/.

[10] Nvidia bluefield data processing units. https://www.nvidia.com/en-us/networking/products/data-processing-unit/.

[11] On-line transaction processing benchmark. https://www.tpc.org/tpcc/.

[12] Redis: In-memory data store. https://redis.io/.

[13] Rocksdb. https://rocksdb.org/.

[14] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE joint international conference on Measurement and Modeling of Computer Systems*, pages 53–64, 2012.

[15] Luiz Barroso, Mike Marty, David Patterson, and Parthasarathy Ranganathan. Attack of the killer microseconds. *Commun. ACM*, 60(4):48–54, mar 2017.

[16] Luiz André Barroso, Jeffrey Dean, and Urs Holzle. Web search for a planet: The google cluster architecture. *IEEE micro*, 23(2):22–28, 2003.

[17] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. IX: A protected dataplane operating system for high throughput and low latency. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 49–65, Broomfield, CO, October 2014. USENIX Association.

[18] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, 44(3):87–95, 2014.

[19] Sol Boucher, Anuj Kalia, David G Andersen, and Michael Kaminsky. Putting the "micro" back in microservice. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 645–650, 2018.

[20] Inho Cho, Ahmed Saeed, Joshua Fried, Seo Jin Park, Mohammad Alizadeh, and Adam Belay. Overload control for {μs-scale}{RPCs} with breakwater. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 299–314, 2020.

[21] Inho Cho, Ahmed Saeed, Seo Jin Park, Mohammad Alizadeh, and Adam Belay. Protego: Overload control for applications with unpredictable lock contention. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 725–738, Boston, MA, April 2023. USENIX Association.

[22] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154, 2010.

[23] Alexandros Daglis, Mark Sutherland, and Babak Falsafi. Rpcvalet: Ni-driven tail-aware balancing of μs-scale rpcs. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 35–48, 2019.

[24] Michael Dalton, David Schultz, Jacob Adriaens, Ahsan Arefin, Anshuman Gupta, Brian Fahs, Dima Rubinstein, Enrique Cauich Zermeno, Erik Rubow, James Alexander Docauer, et al. Andromeda: Performance, isolation, and velocity at scale in cloud network virtualization. In *15th USENIX symposium on networked systems design and implementation (NSDI 18)*, pages 373–387, 2018.

[25] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Commun. ACM*, 56(2):74–80, feb 2013.

[26] Pamela Delgado, Diego Didona, Florin Dinu, and Willy Zwaenepoel. Job-aware scheduling in eagle: Divide and stick to your probes. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, pages 497–509, 2016.

[27] Pamela Delgado, Diego Didona, Florin Dinu, and Willy Zwaenepoel. Kairos: Preemptive data center scheduling without runtime estimates. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 135–148, 2018.

[28] Pamela Delgado, Florin Dinu, Anne-Marie Kermarrec, and Willy Zwaenepoel. Hawk: Hybrid datacenter scheduling. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 499–510, 2015.

[29] Henri Maxime Demoulin, Joshua Fried, Isaac Pedisich, Marios Kogias, Boon Thau Loo, Linh Thi Xuan Phan, and Irene Zhang. When idling is ideal: Optimizing tail-latency for heavy-tailed datacenter workloads with perséphone. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 621–637, New York, NY, USA, 2021. Association for Computing Machinery.

[30] Mo Dong, Qingxi Li, Doron Zarchy, P Brighten Godfrey, and Michael Schapira. {PCC}: Re-architecting congestion control for consistent high performance. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 395–408, 2015.

[31] Dawson R Engler, M Frans Kaashoek, and James O'Toole Jr. Exokernel: An operating system architecture for application-level resource management. *ACM SIGOPS Operating Systems Review*, 29(5):251–266, 1995.

[32] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. Caladan: Mitigating interference at microsecond timescales. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 281–297, 2020.

[33] Jiaqi Gao, Jiamin Cao, Yifan Li, Mengqi Liu, Ming Tang, Dennis Cai, and Ennan Zhai. Sirius: Composing network function chains into P4-Capable edge gateways. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 477–490, Santa Clara, CA, April 2024. USENIX Association.

[34] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. Dominant resource fairness: Fair allocation of multiple resource types. In *8th USENIX symposium on networked systems design and implementation (NSDI 11)*, 2011.

[35] Ionel Gog, Malte Schwarzkopf, Adam Gleave, Robert NM Watson, and Steven Hand. Firmament: Fast, centralized cluster scheduling at scale. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 99–115, 2016.

[36] Robert Grandl, Mosharaf Chowdhury, Aditya Akella, and Ganesh Ananthanarayanan. Altruistic scheduling in {Multi-Resource} clusters. In *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, pages 65–80, 2016.

[37] Mor Harchol-Balter, Cuihong Li, Takayuki Osogami, Alan Scheller-Wolf, and Mark S Squillante. Cycle stealing under immediate dispatch task assignment. In *Proceedings of the fifteenth annual ACM symposium on Parallel algorithms and architectures*, pages 274–285, 2003.

[38] Jack Tigar Humphries, Kostis Kaffes, David Mazières, and Christos Kozyrakis. Mind the gap: A case for informed request scheduling at the nic. In *Proceedings of the 18th ACM Workshop on Hot Topics in Networks*, pages 60–68, 2019.

[39] Jack Tigar Humphries, Neel Natu, Ashwin Chaugule, Ofir Weisse, Barret Rhoden, Josh Don, Luigi Rizzo, Oleg Rombakh, Paul Turner, and Christos Kozyrakis. ghost: Fast & flexible user-space delegation of linux scheduling. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 588–604, 2021.

[40] Stephen Ibanez, Alex Mallery, Serhat Arslan, Theo Jepsen, Muhammad Shahbaz, Changhoon Kim, and Nick McKeown. The nanopu: A nanosecond network stack for datacenters. In *15th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 21)*, pages 239–256, 2021.

[41] Rishabh Iyer, Musa Unal, Marios Kogias, and George Candea. Achieving microsecond-scale tail latency efficiently with approximate optimal scheduling. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, New York, NY, USA, 2023. Association for Computing Machinery.

[42] Eun Young Jeong, Shinae Woo, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. mtcp: A highly scalable user-level tcp stack for multicore systems. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI'14, page 489–502, USA, 2014. USENIX Association.

[43] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. {NetChain}:{Scale-Free}{Sub-RTT} coordination. In *15th USENIX Symposium on Networked*

Systems Design and Implementation (NSDI 18), pages 35–49, 2018.

[44] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. Netcache: Balancing key-value stores with fast in-network caching. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 121–136, 2017.

[45] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. Shinjuku: Preemptive scheduling for μsecond-scale tail latency. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 345–360, Boston, MA, February 2019. USENIX Association.

[46] Kostis Kaffes, Jack Tigar Humphries, David Mazières, and Christos Kozyrakis. Syrup: User-defined scheduling across the stack. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 605–620, 2021.

[47] Kostis Kaffes, Neeraja J Yadwadkar, and Christos Kozyrakis. Centralized core-granular scheduling for serverless functions. In *Proceedings of the ACM symposium on cloud computing*, pages 158–164, 2019.

[48] Vasiliki Kalavri, John Liagouris, Moritz Hoffmann, Desislava Dimitrova, Matthew Forshaw, and Timothy Roscoe. Three steps is all you need: fast, accurate, automatic scaling decisions for distributed streaming dataflows. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 783–798, 2018.

[49] Anuj Kalia, Michael Kaminsky, and David Andersen. Datacenter {RPCs} can be general and fast. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 1–16, 2019.

[50] Rishi Kapoor, George Porter, Malveeka Tewari, Geoffrey M Voelker, and Amin Vahdat. Chronos: Predictable low latency for data center applications. In *Proceedings of the Third ACM Symposium on Cloud Computing*, pages 1–14, 2012.

[51] Antoine Kaufmann, SImon Peter, Naveen Kr Sharma, Thomas Anderson, and Arvind Krishnamurthy. High performance packet processing with flexnic. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 67–81, 2016.

[52] David G Kendall. Stochastic processes occurring in the theory of queues and their analysis by the method of the imbedded markov chain. *The Annals of Mathematical Statistics*, pages 338–354, 1953.

[53] Gyuyeong Kim. Netclone: Fast, scalable, and dynamic request cloning for microsecond-scale rpcs. In *Proceedings of the ACM SIGCOMM 2023 Conference*, pages 195–207. Association for Computing Machinery, 2023.

[54] Gyuyeong Kim. Pushing the limits of In-Network caching for Key-Value stores. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*, pages 1155–1168, Philadelphia, PA, April 2025. USENIX Association.

[55] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. Pocket: Elastic ephemeral storage for serverless analytics. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 427–444, 2018.

[56] Marios Kogias, George Prekas, Adrien Ghosn, Jonas Fietz, and Edouard Bugnion. {R2P2}: Making {RPCs} first-class datacenter citizens. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 863–880, 2019.

[57] ChonLam Lao, Yanfang Le, Kshiteej Mahajan, Yixi Chen, Wenfei Wu, Aditya Akella, and Michael Swift. {ATP}: In-network aggregation for multi-tenant learning. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 741–761, 2021.

[58] Bojie Li, Zhenyuan Ruan, Wencong Xiao, Yuanwei Lu, Yongqiang Xiong, Andrew Putnam, Enhong Chen, and Lintao Zhang. Kv-direct: High-performance in-memory key-value store with programmable nic. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 137–152, 2017.

[59] Jialin Li, Ellis Michael, and Dan RK Ports. Eris: Coordination-free consistent transactions using in-network concurrency control. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 104–120, 2017.

[60] Jialin Li, Naveen Kr Sharma, Dan RK Ports, and Steven D Gribble. Tales of the tail: Hardware, os, and application-level sources of tail latency. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 1–14, 2014.

[61] Xiaozhou Li, Raghav Sethi, Michael Kaminsky, David G Andersen, and Michael J Freedman. Be fast, cheap and in control with {SwitchKV}. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 31–44, 2016.

[62] Yueying Li, Nikita Lazarev, David Koufaty, Tenny Yin, Andy Anderson, Zhiru Zhang, G Edward Suh, Kostis Kaffes, and Christina Delimitrou. Libpreemptible: Enabling fast, adaptive, and hardware-assisted user-space scheduling. In *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 922–936. IEEE, 2024.

[63] Jiaxin Lin, Adney Cardoza, Tarannum Khan, Yeonju Ro, Brent E Stephens, Hassan Wassel, and Aditya Akella. {RingLeader}: Efficiently offloading {Intra-Server} orchestration to {NICs}. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 1293–1308, 2023.

[64] Zaoxing Liu, Zhihao Bai, Zhenming Liu, Xiaozhou Li, Changhoon Kim, Vladimir Braverman, Xin Jin, and Ion Stoica. {DistCache}: Provable load balancing for {Large-Scale} storage systems with distributed caching. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 143–157, 2019.

[65] Tania Lorido-Botran, Jose Miguel-Alonso, and Jose A Lozano. A review of auto-scaling techniques for elastic applications in cloud environments. *Journal of grid computing*, 12:559–592, 2014.

[66] Zhihong Luo, Sam Son, Dev Bali, Emmanuel Amaro, Amy Ousterhout, Sylvia Ratnasamy, and Scott Shenker. Efficient microsecond-scale blind scheduling with tiny quanta. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS '24, New York, NY, USA, 2024. Association for Computing Machinery.

[67] Ilias Marinos, Robert NM Watson, and Mark Handley. Network stack specialization for performance. *ACM SIGCOMM Computer Communication Review*, 44(4):175–186, 2014.

[68] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. Shenango: Achieving high CPU efficiency for latency-sensitive datacenter workloads. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 361–378, Boston, MA, February 2019. USENIX Association.

[69] John Ousterhout, Arjun Gopalan, Ashish Gupta, Ankita Kejriwal, Collin Lee, Behnam Montazeri, Diego Ongaro, Seo Jin Park, Henry Qin, Mendel Rosenblum, Stephen Rumble, Ryan Stutsman, and Stephen Yang. The ramcloud storage system. *ACM Trans. Comput. Syst.*, 33(3), aug 2015.

[70] Yanghua Peng, Yixin Bao, Yangrui Chen, Chuan Wu, and Chuanxiong Guo. Optimus: an efficient dynamic resource scheduler for deep learning clusters. In *Proceedings of the Thirteenth EuroSys Conference*, pages 1–14, 2018.

[71] Simon Peter, Jialin Li, Irene Zhang, Dan RK Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. Arrakis: The operating system is the control plane. *ACM Transactions on Computer Systems (TOCS)*, 33(4):1–30, 2015.

[72] George Prekas, Marios Kogias, and Edouard Bugnion. Zygos: Achieving low tail latency for microsecond-scale networked tasks. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 325–341, New York, NY, USA, 2017. Association for Computing Machinery.

[73] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan Ports, and Peter Richtarik. Scaling distributed machine learning with In-Network aggregation. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 785–808. USENIX Association, April 2021.

[74] Robert Schweller, Ashish Gupta, Elliot Parsons, and Yan Chen. Reversible sketches for efficient and accurate change detection over network data streams. In *Proceedings of the 4th ACM SIGCOMM conference on Internet measurement*, pages 207–212, 2004.

[75] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 18–32, 2013.

[76] Sreeharsha Udayashankar, Ashraf Abdel-Hadi, Ali Mashtizadeh, and Samer Al-Kiswany. Draconis: Network-accelerated scheduling for microsecond-scale workloads. In *Proceedings of the Nineteenth European Conference on Computer Systems*, EuroSys '24, page 333–348, New York, NY, USA, 2024. Association for Computing Machinery.

[77] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at google with borg. In *Proceedings of the tenth european conference on computer systems*, pages 1–17, 2015.

[78] Zilong Wang, Layong Luo, Qingsong Ning, Chaoliang Zeng, Wenxue Li, Xinchen Wan, Peng Xie, Tao Feng, Ke Cheng, Xiongfei Geng, et al. {SRNIC}: A scalable architecture for {RDMA}{NICs}. In *20th USENIX*

Symposium on Networked Systems Design and Implementation (NSDI 23), pages 1–14, 2023.

[79] Xingda Wei, Rongxin Cheng, Yuhan Yang, Rong Chen, and Haibo Chen. Characterizing off-path SmartNIC for accelerating distributed systems. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 987–1004, Boston, MA, July 2023. USENIX Association.

[80] Adam Wierman and Bert Zwart. Is tail-optimal scheduling possible? *Operations research*, 60(5):1249–1257, 2012.

[81] Tong Yang, Jie Jiang, Peng Liu, Qun Huang, Junzhi Gong, Yang Zhou, Rui Miao, Xiaoming Li, and Steve Uhlig. Elastic sketch: Adaptive and fast network-wide measurements. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 561–575, 2018.

[82] Parham Yassini, Khaled Diab, Saeed Mahloujifar, and Mohamed Hefeeda. Horus: Granular In-Network task scheduler for cloud datacenters. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 1–22, Santa Clara, CA, April 2024. USENIX Association.

[83] Zhuolong Yu, Yiwen Zhang, Vladimir Braverman, Mosharaf Chowdhury, and Xin Jin. Netlock: Fast, centralized lock management using programmable switches. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 126–138, 2020.

[84] Irene Zhang, Amanda Raybuck, Pratyush Patel, Kirk Olynyk, Jacob Nelson, Omar S Navarro Leija, Ashlie Martinez, Jing Liu, Anna Kornfeld Simpson, Sujay Jayakar, et al. The demikernel datapath os architecture for microsecond-scale datacenter systems. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 195–211, 2021.

[85] Hang Zhu, Kostis Kaffes, Zixu Chen, Zhenming Liu, Christos Kozyrakis, Ion Stoica, and Xin Jin. RackSched: A Microsecond-Scale scheduler for Rack-Scale computers. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 1225–1240. USENIX Association, November 2020.

Algorithm 1: GenerateMapping(requests)

```
1  latencies ← empty list
2  mappings ← empty list
   /* Find most possible mappings based on
      total request types.                    */
3  for k ← 1 to type(requests) do
4  |   model ← KMeans(k)
5  |   mappings[k] = model.fit(requests)
   /* Obtain the optimal mapping via
      micro-benchmarking.                     */
6  for mapping in mappings do
7  |   latency ← micro-benchmark(mapping)
8  |   latencies.append(latency)
9  opt_mapping ← argmin_{m∈mappings}{latencies}
10 return opt_mapping
```

Appendix

A Algorithm of Workload Mapping Generation

We summarize the algorithm for generating Pallas workload shaping policy in Algorithm 1. It consists of two stages. First, Pallas generates several group mapping candidates with a tailored k-means algorithm (Lines 3-5). Then, Pallas determines the best performant mapping policy through offline simulations (Lines 6-8).

B Overall Request Scheduling

We summarize the scheduling workflow in Algorithm 2. Initially, Pallas identifies the serving group corresponding to each request (Line 4), and ascertains whether the group has encountered a burst (Line 5). Upon detecting a burst, Pallas clones the request and recalibrate it with a proper serving group ID (Lines 6-12). Pallas then sends the request to a server within the determined serving group, ensuring load balancing (Line 13). Once the server receives the request, it leverages its intra-server scheduling algorithm to allocate the request to a CPU core (Lines 16-20). In the concluding step, the server generates a response and sends it back to the client.

C Switch Dataplane Implementation

Weighted round-robin. Pallas employs a weighted round-robin (WRR) algorithm for the intra-group scheduling. In this approach, Pallas assigns a unique counter for each group to determine the destination servers for incoming requests. This counter is incremented sequentially with each request and is programmed to reset to zero upon reaching the aggregate limit, defined by the WRR parameters of the group. The request finds its destination by aligning the counter's current value with the predetermined weight range of servers. The counter is implemented with the register in P4, and updated with the ALU operation.

Algorithm 2: Pallas Request Scheduling

```
1  Event: On request arrival at switch:
2  Init the metadata (meta) for each request
3  if req.flag.clone = 0 then
4      meta.group ← GroupMapping(req.type)
5      burst ← BurstDetection(meta.group)
6      if burst = True then
7          Event: On burst detected:
8          cloned_req, cloned_meta ←
             CloneRequests(req, meta)
9          req.flag.clone ← 1
10         cloned_req.flag.clone ← 2
           /* Assign group for the cloned
              request                          */
11         cloned_meta.group ← FindCloneGroup(req)
12         resubmit(cloned_req, cloned_meta)
13 server ← WeightedRoundRobin(meta.group)
14 send(req, server)
15 Event: On request arrival at server:
16 if server.policy = cFCFS then
17     FCFSSchedule(req)
18 else
19     cores ← core_reservation(req)
20     DARCSchedule(req, cores)
21 send_response(req)
```

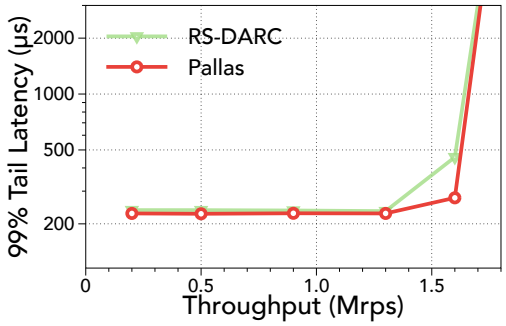


Figure 15: Exponential

Workload recording. Pallas records the workload information of each type of request with two registers in P4. The first register is tasked with recording the cumulative count of monitored requests, while the second is dedicated to computing the Exponentially Weighted Moving Average (EWMA) of the execution times of these requests. The EWMA factor is set to 0.125 in our current prototype, which helps compute the division by bit-shifting operations in P4. The register responsible for counting request numbers is programmed to reset periodically by the regular workload monitoring schedule.

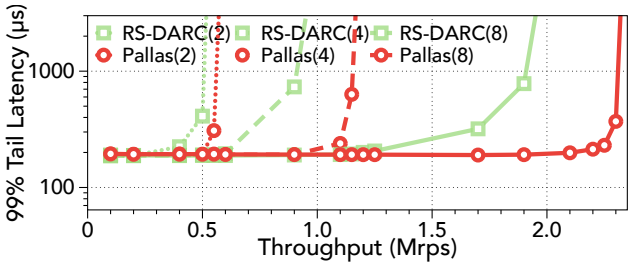


Figure 16: Scalability results of Pallas.

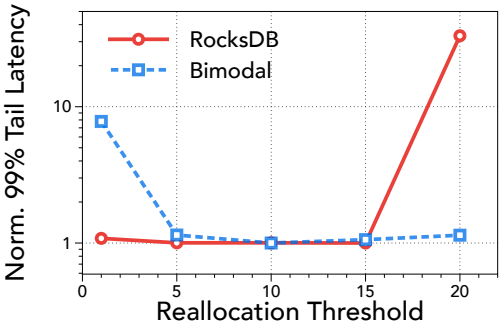


Figure 17: Sensitivity analysis of the reallocation threshold δ.

D Supplementary Experiments

D.1 Performance under Light-tailed Workload

Figure 15 compares Pallas and RackSched under a light-tailed exponential workload with an average request execution time of 25µs. As suggested in RackSched paper, its intra-server scheduler is configured to use cFCFS. We find that RackSched exhibits very close performance to Pallas. This is attributed to the uniformity of request execution times characteristic of the exponential workload. Consequently, both JSQ and Pallas's intra-group load balancer can effectively distribute the load across servers, in which the optimal intra-server cFCFS scheduling is utilized.

D.2 Scalability

We show the scalability of Pallas with two, four and eight servers. We use the Bimodal workload comprised of 90% short requests of 10µs and 10% long requests of 90µs. Figure 16 shows the results of 99% tail latency. We observe that the sustainable throughput of Pallas almost scales linearly with the number of servers, exhibiting much better scalability than RS-DARC. This is because RackSched's application-agnostic inter-server load balancing incurs more variability when facing high request rates and hence results in load imbalance and degrades tail latency. The linear scalability of Pallas can attribute to the near-optimal load balancing and intra-server scheduling with in-network workload shaping, demonstrating that Pallas is able to scale out with more servers without compromising performance.

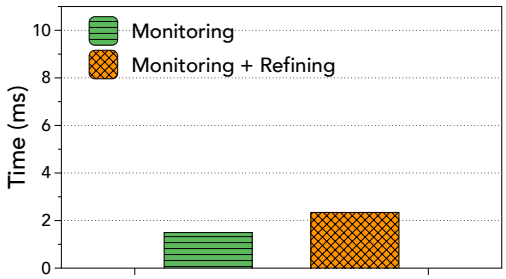


Figure 18: Monitoring overhead of Pallas.

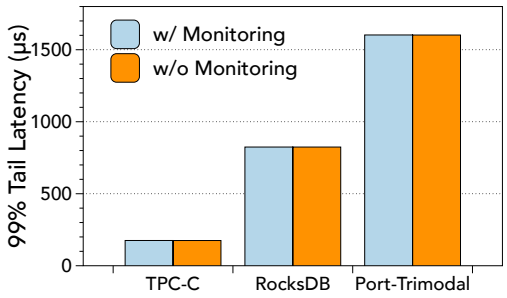


Figure 19: Impact of workload monitoring under static workloads.

D.3 Parameter Sensitivity

Sensitivity to realloaction threshold. We vary the reallocation threshold δ from 1 to 20 and investigate the performance of Pallas with different thresholds. We use both the synthetic workload in Figure 10 and real application RocksDB in Figure 11 to evaluate the sensitivity of Pallas to the reallocation threshold. Figure 17 shows the overall normalized 99% tail latency.

From our observations, Pallas exhibits relative insensitivity to the reallocation threshold within the range of 5-15. This can be attributed to the fact that this threshold range effectively captures workload fluctuations while simultaneously ensuring system stability by avoiding excessive resource reallocations and switch reconfigurations. Specifically, in the context of the synthetically changing bimodal workload, a threshold set at 1 renders the system over-responsive to minor workload variations, thereby substantially impairing system performance. On the other hand, in the scenario of the varying RocksDB workload, a threshold of 20 makes the system oblivious to workload shifts, consequently deteriorating the tail latency.

D.4 Monitoring Overhead

We evaluate the overhead introduced by workload monitoring in Pallas's switch control plane. Figure 18 reports the execution time of two key control plane operations: periodic register reading (workload monitoring only) and infrequent register updates (monitoring together with refining) upon workload shift. Results show that both operations is lightweight relative to the 10 ms monitoring interval, confirming the practicality of our design. We also note that the 10 ms interval targets long-term workload changes and represents a configurable trade-off. If needed, Pallas can scale to support more workload groups by operating at longer monitoring intervals.

D.5 Impact of Workload Monitoring

We further examine the impact of workload monitoring under static workloads. Specifically, we evaluate three static workloads at their respective maximum sustainable request rates in Pallas. As shown in Figure 19, enabling monitoring has no observable effect on system performance, since policy updates are not triggered when no workload shift is detected. This confirms that the monitoring logic is lightweight and non-intrusive.

TGW: Operating an Efficient and Resilient Cloud Gateway at Scale

Yifan Yang[†*], Lin He[†*], Jiasheng Zhou[†], Xiaoyi Shi[†], Yichi Xu[†], Shicheng Wang[†], Jinlong E[◇], Ying Liu[†], Junwei Zhang[○], Zhuang Yuan[○], Hengyang Xu[○]
[†]*Tsinghua University,* [◇]*Renmin University of China,* [○]*Tencent*

Abstract

Large-scale cloud data centers have become a critical Internet infrastructure. As the cloud entrance, today's cloud gateways have integrated multiple functions such as elastic public access and load balancing to cope with the rapid growth of services and requirements. To meet the demands of large-scale clouds for efficient packet forwarding, scalable state management, and high resilience, we design, deploy, and operate Tencent Gateway (TGW), an efficient and resilient cloud gateway at scale. Compared to other large cloud providers that primarily offer services like search, e-commerce, or short-form video, the "killer services" of Tencent Cloud are online gaming and live streaming, which come with much stricter requirements for latency, jitter, and packet loss. From a technological perspective, TGW is highly decoupled and modular, with core components focused on efficient forwarding planes, a scalable state migration mechanism, a resilient failure recovery mechanism, and a failure detection and localization system. In terms of engineering, TGW has been operating in large-scale, real-world industrial environments for eight years, during which we have gained extensive insights and experience. We evaluate TGW both in testbed and real-world scenarios. In our testbed, TGW's single node achieves 2.9× the forwarding capacity of prior systems. Between clusters, states and traffic can be migrated in 4s without packet loss. In our real-world environment, TGW handles tens of Tbps of traffic, with a worst-case packet drop rate ranging from 10^{-7} to 10^{-4}, while balancing traffic across clusters. Additionally, TGW can quickly migrate states and traffic and recover from failures without tenant awareness, guided by our failure localization system, achieving 100% availability for years.

1 Introduction

Large-scale public clouds have increasingly become an essential Internet infrastructure, providing services to millions of clients [2, 4, 12, 17, 32, 35]. We, Tencent Cloud [52], as one of the largest cloud service providers, have been facing rapid growth in traffic volume, service type, and service demand. Our service number and gateway traffic have grown at an average rate of over 100% and 60% per year, respectively, in the past few years. To meet the service level agreements (SLAs) for the high volume and uptime [42] of inter- and intra-data center traffic, efficient and resilient services must be provided in a large-scale cloud carrying tens of Tbps traffic.

As the primary component of the cloud, the cloud gateway plays a vital role in ensuring high-performance and high-availability networking. With the rapid advancement of virtualization technologies and the increasing performance demands of backend services, cloud gateways have evolved beyond merely providing connectivity between external and internal networks. They now also integrate additional complex functions such as load balancing, network intrusion detection, and stateful firewalls. These functions are crucial for maintaining the robustness and security of cloud services.

However, operating such a large-scale networked system is challenging. Traditional gateways, consisting of commodity hardware routers and switches, can provide high throughput, but their fixed functionality does not adapt to the changing scheduling dynamics and proprietary protocols in the cloud. The slow iteration cycle of fixed-function hardware also becomes a bottleneck for the cloud's rapid growth [40]. While there have been efforts in cloud gateway design and implementation using software solutions [3, 31, 47, 58], most of them have been evaluated in small-scale laboratory environments and cannot meet the large-scale cloud requirements for efficiency, scalability, and resilience. Some hybrid solutions [40] utilize programmable hardware to improve performance, but they suffer from vendor dependency and supply chain challenges [10], due to the uncertain outlook of programmable switches.

To bridge the gap, we propose Tencent Gateway (TGW), the gateway system of Tencent Cloud with efficient packet forwarding, scalable state management, and high resilience. TGW has been in operation for eight years and deployed in more than 30 regions. In our largest region, TGW scales to

*Both authors contributed equally to this work.

hundreds of clusters and thousands of machines, handling tens of Tbps of traffic.

TGW's main functions are elastic public access [5, 51] and cloud-level load balancing [50] (§2.1). Compared to other large cloud providers that primarily offer services like search [6, 23], e-commerce [2, 4], or short-form video [8], the "killer services" of Tencent Cloud are online gaming and live streaming, which have much more stringent requirements for latency, jitter, and loss [33]. Based on these requirements and after weighing various alternatives in performance and cost, TGW leverages DPDK-based [27] software server clusters to ensure the efficiency, scalability, and resilience of the gateway system with sequentially increasing priorities (§2.2 and §2.3).

Architecturally, we decouple and modularize TGW's functions at different levels (§3). For the forwarding plane, we decouple public access and load balancing by designing the corresponding TGW instances separately in a hierarchical manner. We design TGW-EIP for stateless elastic public access in regional data centers and TGW-CLB for stateful cloud load balancing in each available zone (AZ) of the region. The tenant traffic will be processed by different forwarding instances and load-balanced. In addition to tenant traffic, the TGW also hosts and manages its internal traffic, including state migration, state synchronizing, and running logs.

TGW mainly consists of efficient forwarding planes, a scalable state migration mechanism, a failure resilience mechanism, and a failure detection and localization system. First, we unleash the forwarding capacity of DPDK by utilizing and optimizing different forwarding models based on the specific functions of TGW-EIP and TGW-CLB (§4). We implement a scalable run-to-completion (RTC) model for TGW-EIP and a two-stage pipeline model for more complex TGW-CLB with empirical tuning and optimization. Second, we design an efficient, lockless, and lossless connection state migration mechanism for fast cluster scaling (§5). In case of overload or failure, TGW can quickly scale the cluster and steer the traffic at runtime. Third, we develop different recovery mechanisms for various levels of failures (§6). TGW uses an empirical multi-cycle synchronization mechanism for stateful backup and establishes failure recovery plans at all levels. TGW can react to failures as soon as they are discovered to minimize their impact, without knowing the cause. Fourth, we design a taint-based telemetry system to detect and localize failures (§7). By distributed analyzing collected logs from active probers and forwarding planes, TGW can automatically detect failures and make decisions within 1 minute.

In addition to technical details, in contrast to most existing work, which is only evaluated in the laboratory, TGW is a mature system that has been commercially operated for years. We summarize our insights and experiences during years of operating TGW, from network topology and hardware setup, to cluster construction and management (§8). We believe that these experiences are helpful for both academia and industry.

We prove TGW's efficiency by listing the evaluation data of TGW running both in a testbed and a real-world environment (§9). In our testbed environment, TGW's single node achieves 2.9× forwarding capacity than in prior work. Between two clusters, states, and traffic can be migrated in 4s without packet loss. In our real-world environment, TGW carries tens of Tbps of traffic with the worst-case packet drop rate of 10^{-7} to 10^{-4} and divides traffic among clusters in balance. TGW can also quickly migrate states and traffic and recover from failures without tenant awareness, achieving 100% availability for multiple years.

2 Background and Motivation

This section describes the background and motivation for building TGW. We commence by providing some foundational background on cloud gateways. Subsequently, we analyze and contrast the alternatives that we have contemplated. Finally, we illustrate our motivation for building TGW under the requirements of our cloud services. For clarity and to make the paper more self-contained, we provide explanations for technical acronyms whose full names do not appear in the main text in Table 1 in the Appendix.

2.1 Background

Today's cloud gateway has evolved far beyond its traditional role as a simple network address translation (NAT) device. It now functions as a traffic scheduler, offering flexible flow distribution with high scalability and availability to meet the demands of modern data centers [40]. The cloud gateway serves as a critical bridge between the public network and the data center network, facilitating seamless communication between users and applications. Services such as online gaming or video streaming often deploy application instances beneath the cloud gateway, which are accessed by users worldwide. Modern cloud gateways, including TGW, support two primary traffic scheduling services:

Elastic public access. Elastic access [5, 51] has become a fundamental service in modern cloud data centers. It allows the assignment of a static public IP address to one or more cloud virtual machines (VMs) or network functions (NFs), such as L4 load balancers. Each VM is typically assigned a private IP address called a direct IP address (DIP). A service exposed to the public is given a public IP address called a virtual IP address (VIP). A cloud gateway distributes traffic destined for the VIP to enable connectivity while ensuring tenant isolation. A client's external public IP address is called a client IP address (CIP). TGW's elastic access service is particularly noteworthy for its transparency, scalability, and high availability. Firstly, TGW ensures seamless access to internal resources and services for public users across various Internet service providers (ISPs) worldwide. Secondly, TGW associates an anycast public IP address with service instances deployed across multiple regions globally. This allows incoming traffic

to be routed to the nearest cloud gateway entrance. Thirdly, by steering traffic through the data center's backbone infrastructure and dynamically selecting the nearest gateway via multiple ISPs, TGW minimizes latency for users worldwide.

Cloud-scale load balancing. Load balancing is one of the most essential services in modern cloud data centers [34, 42], enabling efficient distribution of traffic across a pool of resources. Taking the L4 load balancing as an example, the cloud load balancer maps a single (*VIP, VPort*) tuple to a pool of (*DIP, DPort*) tuples. When the first packet of a new flow arrives, the load balancer selects one (*DIP, DPort*) tuple from the backend pool based on a predefined load-balancing algorithm. This mapping rule is applied to all subsequent packets of the same flow. The load balancer ensures traffic is evenly distributed across instances in the pool while maintaining service performance and reliability. With the rapid growth of the cloud, the traffic scale in data centers has increased significantly, including both north-south and east-west traffic [40]. Besides inter-cloud traffic from public access, there is also tremendous traffic from tenant data centers, on-premise cloud VMs, and other public clouds [45]. In our cloud, the number of data center instances accounts for more than 45%, and the intra-cloud traffic accounts for about 80%. For both inter- and intra-cloud traffic, TGW provides multi-granularity cloud load balancing services with the awareness of real-time workload distribution and link health.

2.2 Alternatives

Due to the variability of services and proprietary protocols in cloud data centers, fixed-function switches are usually not suitable for cloud gateways. Over the years, many advanced schemes, both software-based and hardware-based, have been proposed or matured during TGW's evolution. Despite these advancements, completely replacing TGW's DPDK-based infrastructure with these emerging solutions is extremely challenging in terms of CapEx and OpEx. Nonetheless, we have experimented with these alternatives and even deployed them at a small scale in our cloud. The details are as follows:

XDP [55]. In terms of performance, we find that the XDP-based forwarding plane is not suitable for complex network functions that process large traffic volumes. However, it is simple to develop and deploy as it runs in the kernel without external dependencies. As a result, we deploy XDP-based network functions only at the edge of the data center rather than at its entrance. For example, we deploy XDP on the edge hosts of our cloud CDN, where it achieves load balancing and anti-DDoS functions at 10 Mpps per host, utilizing just 5% of CPU resources.

VPP [54]. VPP offers rich routing features and good performance. However, in a cloud gateway, the functions of elastic access and load balancing are logically simple yet highly coupled. Decoupling these functions into multiple graph nodes does not yield any performance improvement. Instead, it increases OpEx due to the need to manage an additional overlay on top of DPDK. Furthermore, VPP distributes packets to different CPU cores using the NIC's receive side scaling (RSS), which fails to meet the requirements of cloud-scale load balancing in our infrastructure (see §4).

DPUs, FPGAs, and smartNICs. DPUs, FPGAs, and smartNICs offer full or partial programmability and are effective for accelerating NFs [36, 57]. However, their relatively high cost and power consumption [40] make them less appealing for large-scale deployment within cloud gateway infrastructures.

Programmable switches. Programmable switches have proven beneficial in large-scale productive environments [40], offering line-rate throughput of ∼Tbps with minimal CapEx and power costs. For example, Tofino [28] achieves around 3 Tbps throughput at a cost of less than 10$ per Gbps and 1 Watt per Gbps [59]. Consequently, we have introduced programmable switches to accelerate gateway functions within our data center. We directly replace LDs and their upstream and downstream access routers with programmable switches. These switches are directly connected to regional core routers. Each cluster consists of two switches, which serve as backups for each other. To maximize throughput and optimize memory usage, we identify the largest elephant IP flows and distribute them across multiple programmable switches. Each switch manages O(50K) IP addresses and contains O(200K) rules, including GRE [15] encapsulation, rate limiting, and statistics. This setup provides up to 2 Tbps throughput. However, two significant challenges prevent the large-scale deployment of programmable switches: 1) Current programmable ASICs are highly resource-constrained, with tightly coupled network functions that are challenging to manage. 2) The programmable switch ecosystem and community remain underdeveloped and fragmented. Existing programmable chips and associated programming languages lack a unified standard. The programming community has yet to mature, and the software toolchain is still incomplete and evolving. Prematurely committing to a specific type of programmable ASIC risks vendor dependence and potential supply chain issues, such as the discontinuation of Tofino chips [10].

Hyper-converged gateway. As discussed above, the significant advantages of deploying programmable hardware are highly appealing. Combining high performance with extensive flexibility, the hyper-converged gateway [39] integrates the entire cloud infrastructure using CPUs, ASICs, and DPUs. We believe that the hyper-converged solution will be a key area of exploration for the next generation of TGW, although it lies beyond the scope of this paper.

2.3 Motivation

Before discussing our motivation for building TGW, it is important to highlight that its design principles are rooted in the stringent requirements for low latency, jitter, and packet loss demanded by our cloud services. Unlike other major

cloud providers that focus primarily on search [6, 23], e-commerce [2, 4], or short-form video [8], Tencent Cloud's killer services—online gaming and live streaming—demand much stricter latency, jitter, and loss thresholds. For users of online gaming and live streaming services, high round-trip time (RTT) or reconnections caused by network latency, jitter, or packet loss can drastically degrade the QoS (quality of service) [7, 11, 33]. In particular, online gaming users require latency to remain below 100 ms [18,33,41], which means TGW must deliver µs-level latency to compensate for the delays introduced by the Internet and backend servers. With these foundational requirements in mind, we outline our motivation for building TGW, with increasing priorities:

Efficient packet processing. The rapid growth of the cloud has significantly elevated the performance demands of cloud services. The deployment of more and larger services has resulted in a rapid surge in bandwidth requirements. To meet these demands, our cloud gateways must deliver tens of Tbps throughput with µs-level latency. Among the alternatives discussed in §2.2, DPDK stands out as the preferred choice due to its robust performance, flexible development environment, rich community support, and long-term stability, backed by consistent version updates and a reliable compilation toolchain. However, while DPDK offers numerous advantages, scaling it for large-scale environments presents challenges. Simply adding more servers horizontally leads to substantial CapEx and OpEx increases. Therefore, TGW leverages specialized DPDK-based forwarding models tailored to specific forwarding functions, optimizing performance while maintaining scalability and efficiency.

Scalable state management. Data centers handle immense traffic volumes, expansive address spaces, and a vast number of concurrent flows [21, 34, 42, 58], resulting in the need to manage a significant number of states. Additionally, the variety of services requires flexible and fine-grained connection state maintenance. Frequent occurrences of microbursts and software or hardware failures further complicate operations, leading to state loss, flow interruptions, load imbalances, or collateral damage to other tenants, ultimately disrupting performance isolation. These challenges impose stringent requirements on precise state management and scheduling. Traditional cold migration, which causes reconnections, fails to meet the demands for low latency, jitter, and loss. Therefore, TGW needs a scalable connection state management mechanism that ensures seamless, lossless runtime state migration without impacting service continuity.

Fast failure recovery. Large-scale data centers frequently encounter machine and link failures [22, 24, 53]. These failures in the cloud gateway can lead to significant performance degradation or even downtime. In commercial environments, the consequences of such frequent failures include customer losses and reduced revenue. As a result, ensuring the reliability of TGW has become our top priority. TGW must exhibit high resilience to network failures to maintain high availability.

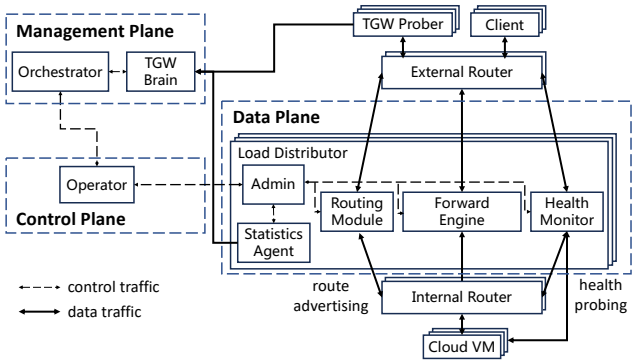


Figure 1: The architecture of a TGW instance. All components are loosely coupled to isolate failures and allow independent scaling.

When network failures of varying severity are detected, TGW should react quickly and recover swiftly without significantly affecting tenant traffic.

3 TGW Overview

In this section, we first illustrate the architecture of TGW by listing its components across different levels. We then present the workflow of TGW's packet processing.

3.1 Architecture

TGW instances are mainly deployed in both the regional entrances, i.e., TGW-EIP, and each AZ of every data center, i.e., TGW-CLB. Each TGW instance is a loosely coupled, disaggregated system composed of multiple components. As illustrated in Fig. 1, a TGW instance spans different operational planes, including a global management plane, cluster-local control planes, and distributed data planes. Besides, TGW has routers located upstream and downstream of the data planes and probers located on the Internet. The details are as follows:

Routers. Commodity external and internal routers are employed between upstream devices, TGW instances, and backend servers. These routers establish BGP [43] connections with the data-plane routing modules and distribute incoming traffic using the ECMP protocol [26].

Load distributors. We refer to the data plane of a TGW instance as a load distributor (LD). The LD is the core component of TGW and comprises several modules. The routing module advertises BGP routes to neighboring routers. The forwarding engine is responsible for handling workload traffic. The admin is a control process within the LD, tasked with interacting with the controller, updating data plane rules, and maintaining network status, such as link connectivity and backend server workload. The health monitor continuously sends probes to detect the activity of routers and servers. It alerts the admin process to adjust scheduling policies based on network health, e.g., by adjusting the weight of servers in load balancing. The statistics agent aggregates LD usage metrics (e.g., CPU, memory, and disk usage), collects data

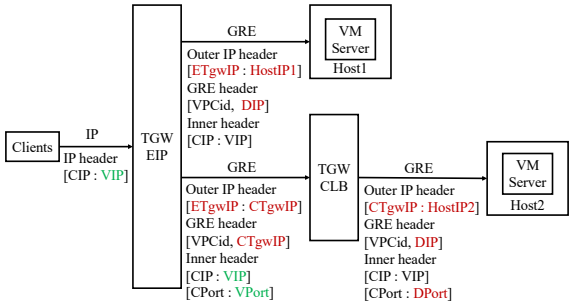


Figure 2: TGW's inbound packet processing flow. Green fields are checked by downstream LDs, and red ones are set by upstream LDs.

from the health monitor, and records the abnormal packet loss information in the forwarding engine. It periodically reports to TGW brains.

Operator. The operator serves as the local controller for an LD cluster and provides northbound APIs to the management plane for resource management tasks, such as managing hardware clusters and network addresses. The operator translates abstract resource configurations defined in the management plane into low-level, concrete rules (e.g., VIP-DIP mappings and rate-limit rules). It then installs these rules into the LDs through their respective admin modules.

Orchestrator. The orchestrator is a global controller, offering comprehensive observability across the entire cloud data center. It evaluates multiple metrics, such as workload, available capacity, and the health status of network links and devices in each cluster. Using a multi-dimensional scheduling algorithm, the orchestrator makes decisions on traffic scheduling and state migration to ensure balanced workload distribution.

TGW probers. TGW probers are active monitoring tools that periodically send probing packets to backend servers and get the probing result from the response packet.

TGW brains. TGW brains are distributed log analyzers that gather runtime telemetry and health data from TGW probers and LDs' statistics agents. They analyze the logs within seconds and report the results to the orchestrator.

3.2 Workflow

The TGW-EIP at the regional entrances primarily handles elastic access, while the TGW-CLB in each AZ manages cloud load balancing for traffic involving cloud VMs or other resources within the zone. In a TGW instance, LDs are configured with rules that map VIPs to DIPs in the cloud data center. A DIP can represent a cloud VM or other successive NF instances, such as a TGW-CLB. Additionally, a VIP can integrate multiple services identified by different VPorts, allowing for more granular mapping rules. Fig. 2 shows the inbound traffic processing flow in TGW. When traffic arrives, external routers initially distribute it based on L3 information via ECMP. The LD clusters then distribute the load at different granularities based on the rules. TGW-EIP and TGW-CLB configure different functions in the LDs and are adapted to distinct forwarding models (§4). TGW-EIP looks up backend cloud VMs or NFs based on the rules and encapsulates the traffic in a GRE tunnel. The DIP and additional critical information, such as the identifier of the VPC (i.e., VPCid), are encoded into the GRE header. These packets are then forwarded as standard IP packets. TGW-CLB distributes traffic based on the 5-tuple and supports other scheduling methods, such as load balancing for QUIC [29]. It selects DIPs and cloud VMs using load-balancing algorithms and pre-configured rules. It then encapsulates the traffic into GRE tunnels, similar to TGW-EIP, but establishes and maintains connection states within the LDs. Subsequent traffic for the same connection is directed to the same DIP and DPort to ensure flow affinity. Internal routers forward these packets as standard IP packets to the corresponding backend servers. Virtual switches in the VPCs, where the cloud VMs are deployed, decapsulate the tunnel, rewrite the destination IP address to the DIP, and add a new Ethernet header for forwarding to the host cloud server. This process uses the VPCid and VM DIP encoded in the header and the routing table maintained by the switch. Outbound traffic processing is the inverse of the inbound packet processing flow. Additionally, there is also traffic between TGW components, such as state migration traffic for cluster scaling (§5), state synchronization traffic for failure recovery (§6), and log aggregation traffic for failure detection and localization (§7). This internal traffic primarily consists of UDP packets with an IP header specifying the source and destination TGW components.

4 Packet Forwarding

This section outlines our optimization of the DPDK-based forwarding plane for efficient packet forwarding. We first unleash DPDK's potential for efficient and scalable forwarding by decoupling the forwarding behaviors of TGW. Specifically, we employ different DPDK forwarding models tailored for TGW-EIP and TGW-CLB. In addition, we enhance the performance of these models by leveraging the traffic characteristics in our cloud and drawing from years of operational experience.

RTC model. As a regional gateway for elastic IP access, TGW-EIP employs an RTC model where packets in a single connection are processed in a single core. This approach minimizes latency and achieves better scalability as cores can operate in parallel for stateless packet forwarding. To further accelerate the forwarding process, we optimize the forwarding table lookup. The forwarding table is implemented as a hash table that maps packet features to the memory address of a forwarding entry. This entry includes the DIP and additional flow control information, such as the flow rate limit or maximum connection limit. However, frequent hash lookups often lead to CPU cache misses, accounting for up to 20% of CPU usage and causing a performance bottleneck in our practice. To mitigate this issue, we utilize a batch-based hash lookup approach. First, we calculate the hash values for packets sequentially in a batch and prefetch them into the cache. The maximum

batch size is set to 64, and the actual packet number in a batch depends on the throughput. We use a separate-chaining-based hash table and extend each list node to store up to four forwarding rule indexes. Storing multiple indexes reduces the overhead of chain expansion when multiple entries share the same hash. We limit each node to four indexes to constrain the node size to 64 bytes, which aligns with the CPU cache line size and maximizes cache utilization. Second, to handle hash collisions—when more than four forwarding rules correspond to entries with the same hash—we prefetch only the first node of the corresponding slot into the CPU cache. Thanks to the CPU's sufficiently large RAM space for hash buckets and the extended linked list nodes, the first node is highly likely to be the one we need. Prefetching it greatly reduces cache misses. This optimization improves L2 and L3 cache hit rates by 8% each and reduces CPU usage for hash lookups from 20% to 5%.

Pipeline model. Although the RTC model has good scalability and performance in stateless packet processing, it cannot be directly applied to `TGW-CLB`. `TGW-CLB` requires more complex and fine-grained L4 flow scheduling than `TGW-EIP`, leading to more intricate connection states indexed by various combinations of flow identifiers. For example, in latency-sensitive services like online gaming, TCP and UDP flows originating from the same client IP must be processed on a single core to minimize delay jitter. This means that the IP-pair-based flow scheduling is more suitable than the traditional 5-tuple approach. Another case is QUIC-aware load balancing, where it is more efficient to distribute and maintain connection states at the granularity of (*srcIP, connectionID*). Additionally, the backend server associated with a VIP may dynamically change based on load distribution algorithms, meaning that only the core processing the flow's first packet has knowledge of the required state. To maintain flow affinity across various service scenarios, the dispatcher must flexibly allocate packets using service-specific identifiers, which are not supported by the NIC's RSS mechanism. Therefore, we implement the forwarding plane of `TGW-CLB` as a two-stage pipeline model. The model consists of 1) the dispatching stage, which maps packets to the appropriate packet processing cores using the service-specific identifiers defined by the rules, and 2) the processing stage, which performs rule and state table lookups, encapsulates packets into a tunnel, and transmits them. Based on our operational experience, a typical proportional relationship between dispatcher cores and processing cores is 1:2, reflecting the flow distribution patterns in our cloud environment. To further enhance performance, each dispatcher core is equipped with its own dedicated ring buffer to avoid locking operations.

5 Live Migration of Connection States

In data centers, core routers commonly employ ECMP to distribute traffic to the cloud gateway. One huge challenge we have faced when building `TGW` is the well-known but tricky "80/20 rule" [40]. In LD, each flow—identified by a unique 5-tuple—is processed by a single forwarding core. However, elephant flows can consume excessive resources on one core, resulting in core overload and rapid performance degradation. Sailfish [40] mitigates this issue by using programmable switches to absorb most of the traffic. However, the remaining responsive traffic is still processed by x86 nodes, creating performance bottlenecks. To address the issue, we design a scalable state migration mechanism. This mechanism enables the seamless migration of network states and traffic from an overloaded cluster to a more capable standby cluster or even multiple standby clusters during runtime. Importantly, this migration is carried out without any packet loss or awareness on the part of the tenant, ensuring uninterrupted service and maintaining the overall performance of the gateway.

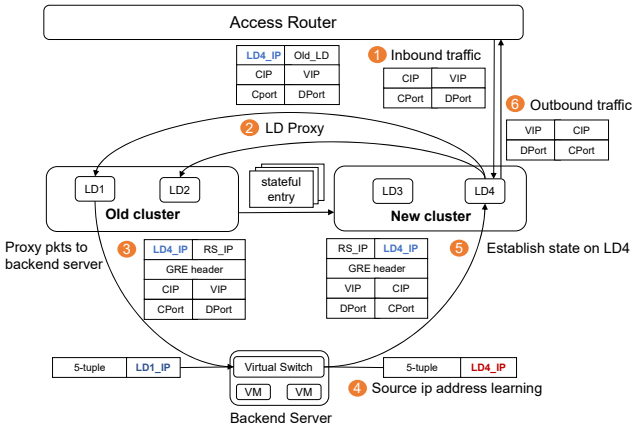


Figure 3: The process of LD proxy for new connections when we migrate connection states during runtime.

Live migration and cold migration. `TGW` supports both live and cold migration of traffic. The key distinction between live migration and cold migration lies in whether the connection information stored in the old cluster is transferred to the new cluster before redirecting traffic. For the critical services hosted in our data centers, such as video gaming and live streaming, the latency overhead caused by reconnections in cold migration is unacceptable. Consequently, we primarily rely on live migration during runtime, enabling traffic redirection without tenant awareness or service disruption. Cold migration, on the other hand, is utilized in scenarios where the original cluster has failed, rendering the connection information invalid, or when the data center faces an urgent fault that requires immediate traffic redirection to maintain availability.

Major state migration. In the live migration process, to migrate the existing states from the old cluster when the migration process begins, the operator module first replicates the stateless configurations to the new cluster. Subsequently, the states are then transferred to this new cluster. `TGW` realizes the migration through a customized application-layer protocol under UDP. This is more engineering work, so we omit this part from this paper. To prevent traffic bursts dur-

ing this process, we employ a time-slice-based mechanism to control the number of states synchronized per slice. Once the majority of connection states—typically over 90%—have been successfully migrated, the operator module directs the routing module in the LD to send BGP advertisements for the migrated VIPs, thereby steering the inbound traffic to the new cluster.

LD proxy for new connections. To avoid dropping new incoming flows during migration, the old cluster remains unlocked during the major state migration, allowing new states to be established. This introduces a consistency issue, as new connections are continuously created in the old cluster. As a result, the new cluster may lack the connection states for subsequent packets of these flows after the routing update. To address this, the LDs in the new cluster will act as proxies, forwarding packets from unseen flows to all LDs in the old cluster during the migration process (② in Fig. 3). Specifically, the proxy LD will forward these packets to every LD in the old cluster. To prevent duplicate responses, a flag is set in the connection state to identify the LD where the state was initially created (i.e., the LD that processed the first packet of the flow). Only the correct LD will process the proxied packets and forward them to the VMs.

Fast convergence. While the LD proxy method helps prevent packet loss during migration, it will amplify traffic, leading to increased latency and potential security risks. To minimize these impacts on tenant traffic, we accelerate the migration process by leveraging the source IP address learning capabilities of the backend server's virtual switch. As shown in Fig. 3, the new cluster initially forwards packets to the old cluster via GRE tunnels, using the source IP in the outer header as its own *LD_IP* (②). The old cluster then forwards the packets to the VM without modifying the outer source IP (③). This action updates the learning table of the virtual switch, causing outbound traffic for these flows to be directed to the new cluster (④). The new cluster will then establish the connection state based on the outbound traffic, completing the migration for this flow (⑤). The amplification is significantly reduced since each new flow is proxied only for the brief period required to receive the outbound traffic from the first packet.

State aggregation. In our forwarding cores, we typically use hash tables to maintain connection states, indexed by the 5-tuple. However, in practice, we often migrate packets at the granularity of the VIP, as a VIP generally corresponds to one or more specific types of services. Within a single LD of our data center, there can be up to 240M states distributed across the cores. Aggregating these states across forwarding cores by VIP would introduce an unacceptable overhead if we were to directly use the connection state tables for migration. Therefore, we decouple the migration from packet processing by introducing an independent thread called the migration agent. This agent manages migration states using reorganized logic and data structures. Specifically, the migration agent reorganizes the states by VIPs as the key, storing only the state addresses of these states. It also maintains additional information, such as the connection ID, to ensure consistency throughout the migration process.

6 Failure Resilience

Failures in cloud data centers occur at different levels, including links, devices, clusters, and regions. These failures can stem from diverse causes such as human errors (e.g., misconfiguration), hardware malfunctions, and software bugs [22, 24, 53]. In this section, we illustrate our multi-cycle connection synchronization scheme for state backup, a fault tolerance model designed to address issues across different levels, and a dispersal migration scheme for blast radius isolation and attack defense. With the continuous evolution of `TGW`'s failure resilience strategies, the unavailability time has been reduced to zero by the first half of 2021.

6.1 Multi-cycle State Synchronization

Naively replicating and broadcasting new states to other LDs would impose significant overhead on link bandwidth and CPU usage. Therefore, based on three practical observations, we design a multi-cycle synchronization mechanism to back up states efficiently and consistently at runtime.

First, to reduce the number of replicated states, we set a timer for each new flow and only replicate those that have been active for at least 3 seconds. The 3-second time limit is an empirical value that reduces the number of replicated flows by 70% to 80% in practice. This is based on the observation that most connections in our cloud, such as HTTP requests and Radius database queries, last only a short time. Their states do not need to be replicated, as they are typically terminated before replication is completed.

Second, we find that exporting all states for replication simultaneously can lead to traffic bursts, while keeping separate export cycles for each state generates a large number of short packets, which decreases throughput. To reduce the replication frequency, we create a batch to cache the states to be exported, with a size of MTU (e.g., 1500 B in `TGW`). Each state identified for replication is inserted into the batch every 4 minutes. The batch is then exported to other LDs via UDP packets every 2 seconds. If the batch becomes full in less than 2 seconds, it is exported immediately. These processing time cycles are empirically defined based on the observation that long-lived connections do not modify their states frequently.

Third, for connections that are being established or closed, the LD will not replicate their states to further reduce overhead. This is because we observe that there is no need to synchronize the states of inactive connections, as they do not compromise failure resilience. We preserve consistency through the timeout recycling mechanism of peer LDs. Each synchronization event refreshes the timer of a connection state, and the state will be recycled if it times out. We do not

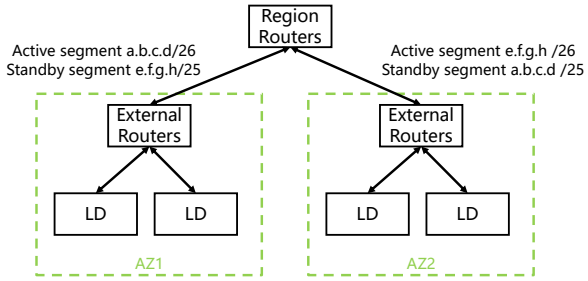


Figure 4: TGW's inter-AZ Active-Standby model. AZ1 and AZ2 operate on different IP segments but back up each other's states.

use dirty flags to enable incremental state updates, as every active state's timer needs to be refreshed by synchronization.

6.2 Fault Tolerance Model

In TGW's fault tolerance model, we combine both the Active-Active model and the Active-Standby model. Given synchronization cost and service requirements, we employ different fault tolerance models under different levels.

Intra-AZ Active-Active model. The failure of one LD within an AZ is common but typically has minimal impact. In each AZ, we use an Active-Active model, where LDs share the same connection state table. This allows traffic previously handled by a failed LD to be seamlessly redirected to another LD within the same AZ. Since LDs in the same AZ are closer in terms of network topology, frequent state synchronization incurs lower costs. The routing modules of every LD in an AZ cluster continuously send BGP advertisements with the same IP addresses to neighboring routers. If an LD fails and stops sending advertisements, routers recognize it as unreachable and reroute traffic to other LDs. The routers distribute traffic with a consistent hash algorithm [13] to minimize traffic redistribution and ensure flow affinity for ongoing connections. Thanks to mature BGP and BFD [30] technologies in data center networks, we can reliably achieve millisecond-level failure detection and failover delays of no more than a few seconds. Because of state synchronization (§6.1), re-diverted traffic is processed and correctly mapped to the VMs, ensuring continuity in service for unaffected connections.

Inter-AZ Active-Standby model. In extreme cases, such as when routers crash or an entire LD group fails, an entire AZ may become inaccessible. Given that intra-cloud traffic accounts for about 80% of our total traffic, the latency difference between AZs is significant. If AZs were set to Active-Active, different connections from the same tenant might be routed to different AZs, causing noticeable latency jitter. To recover from AZ-level failure, we implement an Active-Standby model for AZ pairs, where AZs operate on different network segments but back up each other's states. This ensures consistent latency for all connections from the same tenant, preventing jitter. Specifically, we configure AZs into redundancy pairs. These pairs synchronize their states, as outlined in §6.1, but handle traffic from distinct network segments. We achieve this by configuring IP prefix advertisements for each redundancy pair, where different-length prefix announcements are configured for a subnet in both AZs. For example, as shown in Fig. 4, AZ1 and AZ2 form a redundancy pair. AZ1 is responsible for the subnet a.b.c.d/26, while AZ2 handles e.f.g.h/26. To recover from an AZ1 failure, AZ2 additionally advertises the same IP prefix as AZ1, but with a shorter mask, such as a.b.c.d/25. Symmetrically, AZ1 advertises e.f.g.h/25. When both AZ1 and AZ2 are operational, they handle distinct traffic because of the longest-prefix matching behavior of the regional routers. However, if one AZ fails, traffic is seamlessly redirected to the other, ensuring uninterrupted services.

Inter-region failure reaction. A more disastrous case is the disability of an entire regional-level data center, meaning the complete TGW system in a physical region becomes unavailable. This is an extremely severe failure that is rarely encountered. In this case, the affected routing table is so large for synchronization that re-routing at Layer 3 is not feasible, as the routes converge too slowly. To counter this, we can modify the DNS records of the affected IP addresses to divert the massive traffic to other TGW regions or allow client applications to connect to access point IPs in other regions. This will inevitably bring higher loads to other regions, and all the connections will be broken due to a lack of backup. However, survivability is more important than performance in this case.

6.3 Decentralized Migration

When a failure or attack occurs, it is too late to react after localizing the issue. Furthermore, the backup-based fault tolerance model cannot handle all failures. For example, when abnormal packets or unexpected IP flags appear in the traffic destined for a VIP, bugs in the forwarding plane may be triggered, leading to an LD failure. Since the root of the failure lies in the traffic instead of TGW components, steering the traffic from a crashed LD to other LDs or AZs will only cause more LDs to fail due to abnormal or malicious traffic. To mitigate this as quickly as possible, we design a decentralized migration approach to isolate failures and defend against attacks. This decentralized migration is based on the observation that the vast majority of such failures are caused by traffic destined for a single VIP. This is because different VIPs correspond to different service types, each exhibiting distinct traffic patterns. The abnormal traffic that causes LD crashes may originate from client software associated with one of our services or from malicious attackers targeting a specific service. Specifically, when a set of LDs fails, the orchestrator will schedule the affected traffic by evenly dividing the corresponding affected VIPs into k parts. These parts of VIPs will be distributed to k standby LD sets, called buffer sets, respectively. The states of these VIPs will be concurrently migrated to these buffer sets. This migration follows a cold migration mechanism, as the crashed LDs can no longer proxy new connections and have lost their

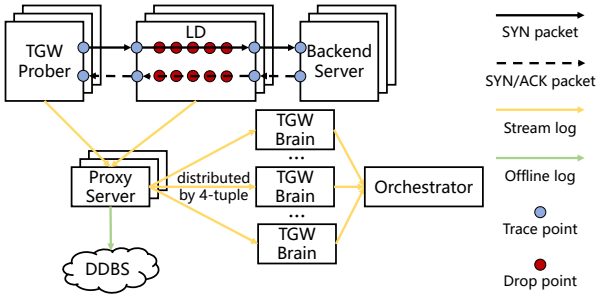


Figure 5: TGW's taint-based telemetry system.

latest states. A decentralized migration using k buffered LD sets can be viewed as k concurrent cold migrations from the failed LD set to the k buffered sets. Through this, failure or attack will only affect one buffer set, reducing the impact by a factor of $1/k$. For the affected buffer set, we can continue to perform one or more decentralized migrations, exponentially reducing the impact of failure or attack. The number of buffer sets k is determined by the hot standby capacity of the cloud region and the priority of the affected traffic. In TGW, the typical value of k is 10. Decentralized migration is triggered when the backup-based recovery mechanisms fail, without the need to locate the root cause of the failure. After decentralized migration, most services will be restored, and the affected scope will be greatly reduced, which also facilitates subsequent failure location and recovery.

7 Failure Detection and Localization

Although TGW can recover most services before we locate the failure, it is crucial to detect the failure as quickly as possible and gather sufficient logs for subsequent failure localization and improvement. However, achieving high observability is always challenging in such a large-scale, complex network. First, it is difficult to locate the exact failed LD when a cluster's performance decreases or experiences jitter. This is because, in our Active-Active fault tolerance model, the LDs within the same AZ cluster are equivalent and load-balanced. Second, when the root of the failure is not LD itself (i.e., the situation described in §6.3), it becomes harder to identify the root cause and reason. Third, our failure localization must be imperceptible to tenants and services, minimizing the impact of failure detection and localization on tenant traffic. To counter this, we design a taint-based telemetry system, inspired by taint checking in program analysis [14, 37]. We efficiently detect and locate failures by active probing and log analysis.

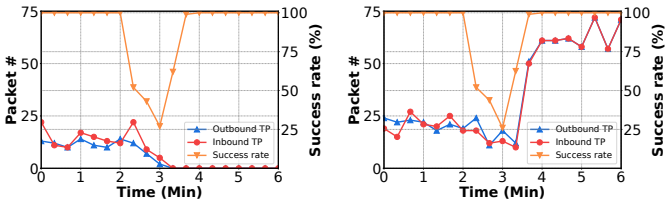
7.1 Taint-based Telemetry

As shown in Fig. 5, the taint-based telemetry system consists of TGW probers, probing proxy servers, and TGW brains. TGW probers continuously generate probing packets, while the LDs and backend servers apply taints to these packets according to the monitoring rules. These rules here are pre-configured static rules. They typically originate from two sources: abnormal forwarding rules that need to be inspected (based on user bug reports), and pre-configured forwarding rules specifically used for probing operational status. Logs are generated by both TGW probers and LDs, and are transferred through the probing proxy servers to the distributed TGW brains. The results are analyzed by the probing brain servers and stored in our data management system. After analysis, the orchestrator will automatically respond to the results.

Taint points. Naively assigning a unique rule to each LD solves the problem of locating a single LD, but it breaks the equivalence of LDs and introduces management overhead. To counter this, we set up several trace points (TPs) and drop points (DPs) to locate the failed LD. TPs are located at the RX and TX of TGW probers, LDs, and backend servers. When a probe packet arrives at a TP, it is tainted according to the rules. The LD also taints the packet with its unique LD ID. DPs are the points where an LD drops a packet. We have more than 20 types of DPs, such as *RX_ENQUEUE_FAIL*, *INVALID_VLANID*, *FLOW_LIMIT*, and *TUNNEL_ENCAP_FAIL*. When a packet is dropped in the LD, the DP is recorded in its statistics agent. The number of TPs in received probing packets and DPs from statistics agents helps us infer single-LD failure, and the type of DPs helps us pinpoint the root cause of the failure when misconfiguration or abnormal traffic occurs.

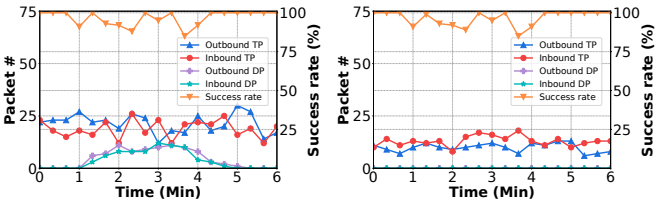
Handshake-based probing. Existing telemetry systems perform telemetry for web applications by constructing application-layer packets [19]. However, this method is not suitable for LDs with only L4 functions. In addition, it also requires more than three interactions with the backend servers, consuming bandwidth and CPU resources. Thus, we utilize the TCP handshake process for telemetry, just like in IP address probing [9, 60]. We use the $(CIP, CPort, VIP, VPort)$ tuple as a unique probing ID for probing packets. A TGW prober will probe a specific service, i.e., a certain $(VIP, VPort)$ tuple, every 5 seconds with a different CPort. For one probing process, the probing SYN packet is first sent by the prober. The tainted contents of the sending SYN packet are written into the response SYN/ACK packet by the backend server. To reduce overhead on bandwidth and CPU usage, we set up special firewall rules in advance to prevent the connection from being established on the backend server. Specifically, after the SYN/ACK packet arrives at the TGW prober, the prober drops the ACK packets for the third handshake and actively sends an RST packet to terminate the half-connection, thereby freeing the backend server resources.

Vantage points. To cover all LDs in the inbound direction, we deploy dozens of servers with public IP addresses as TGW probers. They are deployed across three geographic areas and connected to BGP-enabled routers to mitigate the impact of ISPs. To cover all LDs in the outbound direction, we also deploy dozens of backend servers for responses. These backend



(a) LD1 (b) LD2

Figure 6: Single-LD crash in an LD cluster.



(a) LD1 (b) LD2

Figure 7: Single-LD jitter in an LD cluster.

servers are distributed across multiple regions and racks.

Log analysis. Due to the large scale of TGW, the logs generated by TGW probers and LDs are vast. Aggregating them for analysis could result in GBps-level traffic and pose significant challenges for real-time processing. Therefore, we combine distributed streaming analysis with offline analysis for efficient log processing. For stream analysis, we aggregate and analyze probing logs from both probers and LDs every 20 seconds. These logs are transferred via proxy servers to distributed TGW brains. Logs sharing the same probing ID (i.e., the 4-tuple) are aggregated and directed to the corresponding TGW brain. Within seconds, the TGW brains analyze the logs and send the results to the orchestrator. For common failures, TGW can automatically complete detection, analysis, and decision-making within 1 minute, significantly reducing the need for manual intervention. In addition to online monitoring, we also inspect historical logs when diagnosing issues. These logs are stored in commercial distributed database systems (DDBS), with the total volume exceeding 10 TB per day. To facilitate fast queries, we utilize the inverted index algorithm that allows TB-scale data to be queried in under one second.

7.2 Case Study: Problematic LD location

We show how we figure out the problematic LD through our telemetry system in two real-world cases.

Single-LD crash. As shown in Fig. 6(a) and Fig. 6(b), we present the probe success rate of the cluster and the number of partial TPs during the failure of a single LD. When an LD crashes at 2 Min, the probing success rate of the cluster decreases. The number of TPs for inbound and outbound packets arriving at LD1 drops to nearly zero, significantly lower than that of LD2. This indicates that LD1 has completely crashed, and all packets fail to reach its RX. TGW detects and recovers this failure within 1 minute and responds at 3 Min, redirecting the traffic to LD2.

Single-LD jitter. As shown in Fig. 7(a) and Fig. 7(b), we additionally present the total number of DPs in both inbound and outbound directions during the failure of a single-LD jitter. In LD1, the number of DPs begins to rise at 1 Min, while in LD2, the number of DPs remains at zero. This indicates that the jittery packet loss is caused by LD1. Further analysis of the DP types allows us to pinpoint the root cause of the jittery packet loss.

8 Operational Experiences

Operating a large-scale gateway system is not a trivial task. TGW has been in operation for eight years, during which it has significantly evolved in forwarding performance and availability. Here, we share our operational experiences derived from our long-term operation of TGW.

Blast radius isolation. A crucial principle of availability is limiting the "blast radius" [16] that failures can impact: 1) We restrict the fault domain in architectural design. TGW is designed to be distributed and multi-layered, with different modules in regions, AZs, and clusters. Each module in a particular layer is decoupled from the others to reduce the impact of failures. Each module can be independently scaled or backed up. 2) We implement ad-hoc availability mechanisms at each layer, including customer service partitioning and placement (§6.3), routing or connection state management, and underlying hardware arrangements. 3) We divide the global controller based on geographic areas and deploy separate scheduler systems for each area. Each scheduler system only accesses the status of the data center it is responsible for and performs scheduling within that area.

Redundancy. Redundancy is always the best practice to ensure reliability. Considering the trade-off between cost and effectiveness, we use 50% redundancy, which ensures that: 1) At the AZ level, for two available zones as a redundancy pair, the clusters of one AZ can handle all traffic if the other AZ fails. 2) At the rack level, machines within a cluster are distributed across multiple racks, ensuring that the cluster remains available if half of the racks fail. 3) At the machine level, if half of the machines in a cluster fail, the remaining machines can carry all the traffic. 4) At the link level, the failure of half of the network cards or fibers of one or more machines does not affect the cluster.

Cluster construction. When a new cluster is set up after the machines and software are deployed, static parameter checks are first performed in a no-traffic environment. Once the cluster is in place, it will begin carrying traffic by sending BGP advertisements. After a batch of addresses is placed in the cluster, one of them is selected for long-term probing to monitor the forwarding status of the cluster before it becomes available for purchase. Our empirical cluster size is 4. At one point, we scaled a single cluster to support up to 8 LDs per AZ. However, we found that during operations, a small cluster composed of 4 LDs is sufficient for nearly all scenarios, while

also providing the advantage of a smaller failure domain.

Cluster management. When an LD in the cluster needs to be maintained, its BGP advertisements will first be terminated, and the traffic originally handled by it will be automatically routed to other LDs. To manage dynamic changes in traffic volume, some gateway operators may choose to enable the CPU's low-power mode to reduce power consumption. However, we find that frequent mode switching introduces processing delays in the CPU and causes jitter. As a result, we adjust the capacity of the cluster by performing fast scaling instead of switching CPU C-States. When the traffic volume exceeds the threshold, we migrate part of the traffic to standby LDs (§5). When the traffic volume decreases, we first isolate the extra LDs' routing and then release them.

Odd-even routing. In the case of packet fragmentation and session persistence [49], we need to ensure that packets with the same IP pair are processed by the same LD. To achieve this, we configure odd-even routing between core routers and access routers, ensuring that packets with the same parity are forwarded by the same access router and distributed to LDs via the IP-pair-based ECMP protocol.

From multicast to unicast. In the early versions of TGW, we used multicast to transfer state synchronization and migration packets between LDs. However, we found that multicast was not suitable for large-scale gateways spanning multiple regions. Initially, small-scale TGW clusters were deployed in isolated environments within a single region, where multicast could be achieved reliably and stably by the data center network. However, as TGW evolved to support live migration functions and cross-region cluster configurations, the complexity of configuring multicast across regions increased, and its reliability diminished. As a result, unicast UDP packets became the preferred choice due to their lower OpEx and greater reliability. The trade-off of using unicast is that each LD must maintain its own list of peer LDs for communication.

DDoS defense. The main function of TGW is to schedule traffic. TGW's DDoS defense relies on the security gateway deployed between the Internet and TGW, which monitors and cleans attack traffic. TGW supports configuring rate-limiting functions across multiple dimensions, including bandwidth, packet volume, total number of connections, and new connection rate, all at the service granularity. This allows for fine-grained isolation when services are under attack.

9 Evaluation

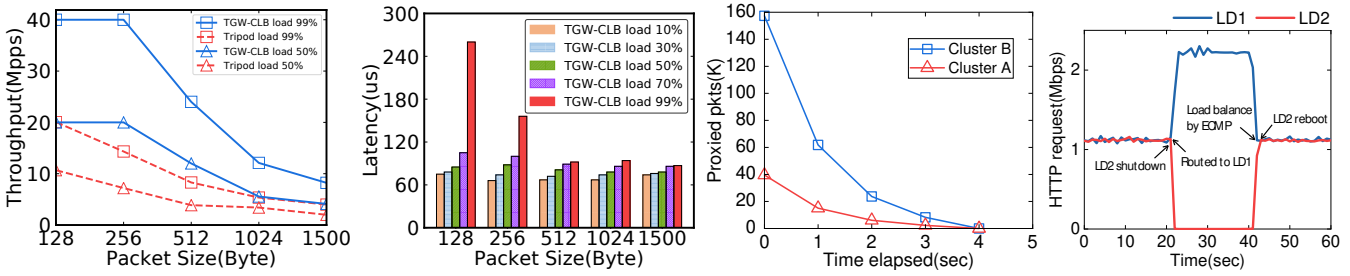
We demonstrate TGW's capabilities in performance, flow migration, and failure resilience. Our results are from two parts: experimental data from the testbed and real-world data from one of our regions. The testbed experiments are conducted on servers with 528 GB of RAM and 96 cores (AMD EPYC 7K62 Processor @2.6Ghz). They are connected via Mellanox MT28800 Family 100 Gbps NICs and commercial switches. In our testbed experiment, a single LD node is a test server running the LD forwarding plane along with the necessary management and control components.

9.1 Testbed Scenario

Single-node forwarding performance. We evaluate a single-node TGW-CLB instance in terms of throughput and latency. In this instance, 50 cores are used for two-stage pipelining forwarding, 18 cores are used for the first stage, and 32 are for the second. For throughput, we compare TGW-CLB with Tripod [58] under different loads with various packet sizes ranging from 128 B to 1500 B. Tripod is also configured with 50 cores for forwarding. We generate a huge amount of UDP traffic and perform at least 5 table lookups for each flow with O(1M) static rules and O(100M) dynamic connection states. As shown in Fig. 8(a), TGW-CLB achieves 2× (128 B) to 2.9× (512 B) throughput than Tripod for different packet sizes under the 99% CPU load and 1.9× (128 B) to 3.1× (512 B) under the 50% CPU load. TGW-CLB achieves a maximum of 96.8 Gbps bandwidth for the 1024 B packet. Although both TGW-CLB and Tripod are based on DPDK, TGW achieves significantly better performance due to extensive tuning and optimizations built on top of DPDK. These optimizations stem from long-term operational experience and testing, and include not only model and algorithmic optimizations discussed in §4, but also a range of engineering optimizations, such as memory management, code structure and code execution efficiency, as well as OS and hardware configurations. We also measure the average processing latency of TGW-CLB for each packet size under five different loads in Fig. 8(b). In most cases, for different packet sizes, TGW-CLB can provide a stable latency of 66 μs to 105 μs. The latency for the same packet size also increases slightly when the load rises.

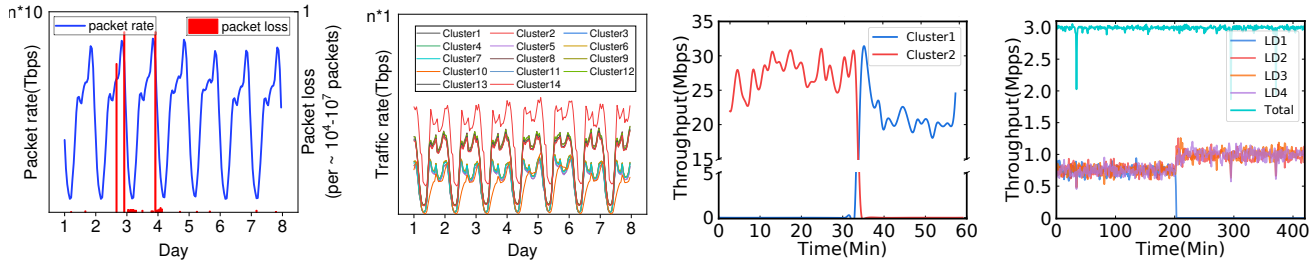
Two-cluster migration. We prepare two clusters, A and B, each with 4 LDs deployed to measure TGW's LD proxy and state convergence. We set up 8M connections in cluster A and started the migration at 0 s, as shown in Fig. 8(c). The throughput of proxied traffic in cluster B is 4× of that of cluster A at the beginning because the LDs in cluster B will forward traffic to all four LDs in cluster A due to state miss, and only one LD in cluster A will process the proxied packet to reduce overhead. Since cluster B creates states based on the outbound traffic and each connection only has a few packets to be proxied, the migration process is greatly sped up and converges in less than 4 seconds.

Two-LD failure recovery. To illustrate TGW's failure resilience, we set up a cluster with 2 LDs, send HTTP requests to the LDs via ECMP, and monitor the traffic handled by each LD during a failure. As shown in Fig. 8(d), equal traffic passes through two LDs at the beginning. At 22 s, LD2 is shut down. The traffic through LD2 is immediately reduced to 0, while the traffic is almost immediately steered to LD1 via ECMP. Due to their advanced state synchronization, LD1 successfully handles almost all the incoming requests. At around 42



(a) Single-node throughput (b) Single-node latency (c) Two-cluster state migration (d) Two-LD failure recovery

Figure 8: TGW's performance in the testbed environment.



(a) Traffic rate and packet drop rate (b) Load balancing for clusters (c) Migration between 2 clusters (d) Failure recovery between 4 LDs

Figure 9: TGW's productive performance in one of our large cloud regions.

s, LD2 reboots. Since LD1 and LD2 use BGP to advertise the same IP addresses, the upstream sender distributes the traffic equally between the two LDs eventually.

9.2 Real-world Scenario

Throughput and packet loss rate. Fig. 9(a) shows the performance of TGW in a large cloud region during a one-week online shopping festival. Note that for privacy reasons, we muddle the integer n lying between 1 and 10. TGW has tens of Tbps of throughput and provides a low loss rate while carrying huge amounts of traffic. Even in the worst case, when certain cores are hit by heavy-hitter flows on day 3 and day 4, the packet loss rate only reaches 10^{-7} to 10^{-4}.

Traffic load distribution among clusters. The "80/20" rule makes it impossible to balance the load across clusters fully. Fig. 9(b) shows that the traffic is split horizontally among clusters within a region by TGW and is well balanced across most clusters at different workload levels. TGW sets different workload constraints among clusters according to the performance sensitivity of the deployed services and sets low-loaded clusters as hot backups so that traffic from high-loaded clusters can be quickly migrated to these clusters to maintain SLAs.

Live migration. We first evaluate the migration speed by recording one major state migration process of our productive cloud data center. In this process, we track the migration of 1371 VIPs, which host 130M connections. We find that 95.6% VIPs are migrated in 20 s, with 24.5% in 10 s and 71.1% in 10∼20 s. The left 4.4% VIPs are also completed in 20∼30 s. We secondly show the traffic migration process between two running clusters in Fig. 9(c). At 33 Min, we start migrating a /24 VIP segment from one cluster to another. Though new connections are continuously established on the old cluster, the migration process migrates the major traffic quickly and converges in 2 Min without any packet loss. Our real-time monitoring and customer feedback indicate that during the migration, no connections are interrupted, and the migration of services is not perceived by the customer.

Failure recovery. According to our record, TGW can synchronize 130M connections in 13 s with a peak bandwidth of 350 Mbps on each LD NIC. Fig. 9(d) shows one real-world failure in one of our commercial clusters with 4 LDs. The cluster carries 3 Mpps traffic and LD1 fails at 200 Min. Since the states of LD1 are synchronized in advance and the other LDs have redundant computing power and throughput, the traffic carried by LD1 is immediately taken over equally by other LDs. The total throughput is not affected, and the service is still available.

10 Lessons Learned

TGW has high scalability. Thanks to the distributed and modular design of TGW, it can be scaled at every level. TGW is currently scaled in more than 30 regions with the maximum scale of hundreds of clusters, thousands of machines, and tens of Tbps throughput. Our single cluster can support more than 200K rules and 100M connections. TGW carries almost all the public network traffic of our company's services and the public or load-balancing traffic of our cloud tenants.

TGW is proved easy to port. Compared to prior work [40], TGW's portability has been proved by practice. In addition to our own cloud gateway, we have already deployed TGW for dozens of other large custom companies, including famous

banks, TV stations, and car groups. The forwarding plane of TGW is a software component developed based on DPDK. It can be easily compiled and deployed on DPDK-enabled NICs and CPUs. The control plane and management plane components of TGW are hardware-independent application-layer logic that can run on general-purpose Linux OSes. Therefore, TGW can be smoothly ported to other data centers operated by large companies. Operators with knowledge of data center network management and DPDK programming can operate the entire system effectively. In contrast, hybrid solutions like Sailfish [40], which rely on P4 [1]-centric programmable hardware, present a high technical barrier due to the niche programming language and the immature compilation toolchain, requiring specialized hardware and professional expertise for deployment and operation.

TGW can be integrated with other solutions. Ease of integration with non-DPDK solutions and flexibility for future evolution are key design considerations in TGW. For example, when integrating TGW with alternative packet forwarding solutions, such as XDP and programmable switches (discussed in §2.2), we only need to implement corresponding APIs for the existing control plane and replace them with the DPDK-based forwarding plane.

The service type determines the properties of the infrastructure. As the most critical infrastructure of Tencent Cloud, TGW's properties are primarily determined by the service type. Tencent's core services—such as online gaming and live streaming—require TGW to guarantee low latency and jitter with high availability. It is worth mentioning that TGW-CLB supports port-segment-based configuration, i.e., a forwarding rule can specify a range of ports as a VPort, which maps to a corresponding port range on the backend server. This feature is particularly useful for multiplayer competitive games. Since each game room usually uses one port, a port segment forwarding rule can support multiple game room network connections in bulk. This capability is unique to TGW among cloud vendors and is driven by the specific requirements of our services.

11 Related Work

Cloud gateway. There are some related frameworks for cloud gateways: NVP [31] uses software to implement a cloud gateway to host thousands of tunnels, but does not support multi-tenancy in large-scale clouds. Tripod [58] provides an NF framework to host various NFs in cloud gateway scenarios. However, its unoptimized forwarding performance and coupled state management logic are insufficient for scaling in large-scale clouds. Protego [47] is a centrally controlled cloud-scale multi-tenant gateway. However, it does not solve the problem of heavy-hitter flows making certain cores overload. Arashloo et al. [3] use fixed-function switches to accelerate the cloud gateway, but the solution lacks the flexibility to meet our requirements in NFs. Sailfish [40] and Luoshen [39] use programmable switches and DPUs to enhance the performance of cloud gateways. However, they are suffering from supply chain breakdown by using P4-centric Tofino chips [10], lacking viability. DSR [45] is a disaggregated software-defined router system to provide reliable access. TGW can work with DSR when it acts as an edge access and steers the traffic to TGW, as Facebook [44] and Google [56] do.

Cloud load balancing. Load balancing is a fundamental building block in cloud gateway systems. Beamer [38] and Yoda [20] provide L4 or L7 load balancing in the cloud gateway using software methods. Ananta [42] focuses on the ECMP to scale, and Maglev [13] utilizes the consistent hash to map VIP and connections. These methods give less consideration to cooperation with public access. Large-scale deployment may cause considerable inter-AZ traffic because of the coupling with public access. They also sacrifice the flexibility of state management for different services in our practice. Alibaba Cloud [2] and AWS [4], as typical cloud providers, have similar services, including elastic IP address and cloud load balancing. Duet [21], SilkRoad [34], Prism [25], ConWeave [48] and Miresga [46] utilize programmable switches to accelerate load balancing. Tiara [57] and AccelTCP [36] use FPGAs and smartNICs to balance the traffic. However, they cannot support flexible DIP selection and are not general for all kinds of L4 traffic.

12 Conclusion

We introduce TGW, a long-term operated, large-scale cloud gateway system. It consists of efficient forwarding planes, a scalable state migration mechanism, a multi-level failure resilience mechanism, and a taint-based failure localization system. The experimental data from both the testbed and real-world scenarios illustrate TGW's capability in performance, scalability, and resilience. We believe that TGW's designs and the experiences gained from long-term, large-scale network operations are beneficial to both academia and industry. TGW will continue to evolve in the future as a key infrastructure of Tencent Cloud.

13 Acknowledgments

We sincerely thank our shepherd, Daehyeok Kim, and the anonymous reviewers for their insightful comments and helpful feedback. This work is supported by the National Natural Science Foundation of China under Grant No. 62302253 and 62102224, the Beijing Natural Science Foundation under Grant No. 4252017, and the research fund of Tsinghua University-Tencent Joint Laboratory for Internet Innovation Technology. Lin He is the corresponding author of this paper.

References

[1] P4 language. https://p4.org, 2025.

[2] Alibaba. Alibaba Cloud Services. https://us.alibabacloud.com/, 2025.

[3] Mina Tahmasbi Arashloo, Pavel Shirshov, Rohan Gandhi, Guohan Lu, Lihua Yuan, and Jennifer Rexford. A scalable vpn gateway for multi-tenant cloud services. *SIGCOMM Comput. Commun. Rev.*, 48(1):49–55, apr 2018.

[4] AWS. AWS Cloud Services. https://aws.amazon.com/, 2025.

[5] AWS. Elastic IP addresses. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/elastic-ip-addresses-eip.html, 2025.

[6] Baidu. Baidu Cloud Services. https://intl.cloud.baidu.com, 2025.

[7] Tom Beigbeder, Rory Coughlan, Corey Lusher, John Plunkett, Emmanuel Agu, and Mark Claypool. The effects of loss and latency on user performance in unreal tournament 2003®. In *Proceedings of 3rd ACM SIGCOMM Workshop on Network and System Support for Games*, NetGames '04, page 144–151, New York, NY, USA, 2004. Association for Computing Machinery.

[8] ByteDance. ByteDance Cloud Services. https://bytehouse.cloud, 2025.

[9] Daguo Cheng, Lin He, Chentian Wei, Qilei Yin, Boran Jin, Zhaoan Wang, Xiaoteng Pan, Sixu Zhou, Ying Liu, Shenglin Zhang, Fuchao Tan, and Wenmao Liu. Luori: Active probing and evaluation of internet-wide ipv6 fully responsive prefixes. In *2024 IEEE 32nd International Conference on Network Protocols (ICNP)*, pages 1–12, 2024.

[10] Max A. Cherney. Intel is halting development of the networking chip it got from barefoot networks. https://www.bizjournals.com/sanjose/news/2023/01/26/intel-halts-development-of-tofino-switch-chips.html, 2023.

[11] Mark Claypool and Kajal Claypool. Latency and player actions in online games. *Commun. ACM*, 49(11):40–45, November 2006.

[12] Michael Dalton, David Schultz, Jacob Adriaens, Ahsan Arefin, Anshuman Gupta, Brian Fahs, Dima Rubinstein, Enrique Cauich Zermeno, Erik Rubow, James Alexander Docauer, Jesse Alpert, Jing Ai, Jon Olson, Kevin DeCabooter, Marc de Kruijf, Nan Hua, Nathan Lewis, Nikhil Kasinadhuni, Riccardo Crepaldi, Srinivas Krishnan, Subbaiah Venkata, Yossi Richter, Uday Naik, and Amin Vahdat. Andromeda: Performance, isolation, and velocity at scale in cloud network virtualization. In *Proceedings of NSDI 2018*.

[13] Daniel E. Eisenbud, Cheng Yi, Carlo Contavalli, Cody Smith, Roman Kononov, Eric Mann-Hielscher, Ardas Cilingiroglu, Bin Cheyney, Wentao Shang, and Jinnah Dylan Hosein. Maglev: A fast and reliable software network load balancer. In *Proceedings of NSDI 2016*.

[14] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Trans. Comput. Syst.*, 32(2), jun 2014.

[15] D. Farinacci, T. Li, S. Hanks, D. Meyer, and P. Traina. Rfc2784: Generic routing encapsulation (gre), 2000.

[16] Andrew D. Ferguson, Steve Gribble, Chi-Yao Hong, Charles Killian, Waqar Mohsin, Henrik Muehe, Joon Ong, Leon Poutievski, Arjun Singh, Lorenzo Vicisano, Richard Alimi, Shawn Shuoshuo Chen, Mike Conley, Subhasree Mandal, Karthik Nagaraj, Kondapa Naidu Bollineni, Amr Sabaa, Shidong Zhang, Min Zhu, and Amin Vahdat. Orion: Google's software-defined networking control plane. In *Proceedings of NSDI 2021*.

[17] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, Harish Kumar Chandrappa, Somesh Chaturmohta, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, and Albert Greenberg. Azure Accelerated Networking: SmartNICs in the Public Cloud. In *Proceedings of NSDI 2018*.

[18] Tobias Fritsch, Hartmut Ritter, and Jochen Schiller. The effect of latency and network limitations on mmorpgs: a field study of everquest2. In *Proceedings of 4th ACM SIGCOMM Workshop on Network and System Support for Games*, NetGames '05, page 1–9, New York, NY, USA, 2005. Association for Computing Machinery.

[19] Cloud Native Computing Fundation. OpenTelemetry. https://opentelemetry.io, 2025.

[20] Rohan Gandhi, Y. Charlie Hu, and Ming Zhang. Yoda: A highly available layer-7 load balancer. In *Proceedings of the Eleventh European Conference on Computer*

Systems, EuroSys '16, New York, NY, USA, 2016. Association for Computing Machinery.

[21] Rohan Gandhi, Hongqiang Harry Liu, Y. Charlie Hu, Guohan Lu, Jitendra Padhye, Lihua Yuan, and Ming Zhang. Duet: Cloud scale load balancing with hardware and software. In *Proceedings of SIGCOMM 2014*.

[22] Phillipa Gill, Navendu Jain, and Nachiappan Nagappan. Understanding network failures in data centers: Measurement, analysis, and implications. In *Proceedings of SIGCOMM 2011*.

[23] Google. Google Cloud Services. https://solutions.cloud.google.com/home, 2025.

[24] Albert Greenberg, James R. Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A. Maltz, Parveen Patel, and Sudipta Sengupta. Vl2: A scalable and flexible data center network. In *Proceedings of SIGCOMM 2009*.

[25] Yutaro Hayakawa, Michio Honda, Douglas Santry, and Lars Eggert. Prism: Proxies without the pain. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 535–549, 2021.

[26] C. Hopps. Rfc2992: Analysis of an equal-cost multi-path algorithm, 2000.

[27] Intel. DPDK: Data plane development kit. https://https://www.dpdk.org/, 2025.

[28] Intel. Intel tofino series programmable ethernet switch asic. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-series/tofino.html, 2025.

[29] Jana Iyengar and Martin Thomson. QUIC: A UDP-Based Multiplexed and Secure Transport. RFC 9000, May 2021.

[30] Dave Katz and David Ward. Bidirectional Forwarding Detection (BFD). RFC 5880, June 2010.

[31] Teemu Koponen, Keith Amidon, Peter Balland, Martín Casado, Anupam Chanda, Bryan Fulton, Igor Ganichev, Jesse Gross, Paul Ingram, Ethan Jackson, et al. Network virtualization in multi-tenant datacenters. In *Proceedings of NSDI 2014*.

[32] Ze Li, Qian Cheng, Ken Hsieh, Yingnong Dang, Peng Huang, Pankaj Singh, Xinsheng Yang, Qingwei Lin, Youjiang Wu, Sebastien Levy, and Murali Chintalapati. Gandalf: An Intelligent, End-To-End Analytics Service for Safe Deployment in Large-Scale Cloud Infrastructure . In *Proceedings of NSDI 2020*.

[33] Florian Metzger, Stefan Geißler, Alexej Grigorjew, Frank Loh, Christian Moldovan, Michael Seufert, and Tobias Hoßfeld. An introduction to online video game qos and qoe influencing factors. *IEEE Communications Surveys and Tutorials*, 24(3):1894–1925, 2022.

[34] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics. In *Proceedings of SIGCOMM 2017*.

[35] Microsoft. Microsoft Azure: Cloud Computing Services. https://azure.microsoft.com/en-us/, 2025.

[36] YoungGyoun Moon, SeungEon Lee, Muhammad Asim Jamshed, and KyoungSoo Park. {AccelTCP}: Accelerating network applications with stateful {TCP} offloading. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 77–92, 2020.

[37] James Newsome and Dawn Xiaodong Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of NDSS 2005*.

[38] Vladimir Olteanu, Alexandru Agache, Andrei Voinescu, and Costin Raiciu. Stateless datacenter load-balancing with beamer. In *Proceedings of NSDI 2018*.

[39] Tian Pan, Kun Liu, Xionglie Wei, Yisong Qiao, Jun Hu, Zhiguo Li, Jun Liang, Tiesheng Cheng, Wenqiang Su, Jie Lu, Yuke Hong, Zhengzhong Wang, Zhi Xu, Chongjing Dai, Peiqiao Wang, Xuetao Jia, Jianyuan Lu, Enge Song, Jun Zeng, Biao Lyu, Ennan Zhai, Jiao Zhang, Tao Huang, Dennis Cai, and Shunmin Zhu. LuoShen: A Hyper-Converged programmable gateway for Multi-Tenant Multi-Service edge clouds. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 877–892, Santa Clara, CA, April 2024. USENIX Association.

[40] Tian Pan, Nianbing Yu, Chenhao Jia, Jianwen Pi, Liang Xu, Yisong Qiao, Zhiguo Li, Kun Liu, Jie Lu, Jianyuan Lu, Enge Song, Jiao Zhang, Tao Huang, and Shunmin Zhu. Sailfish: Accelerating cloud-scale multi-tenant multi-service gateways with programmable switches. In *Proceedings of SIGCOMM 2021*.

[41] Lothar Pantel and Lars C. Wolf. On the impact of delay on real-time multiplayer games. In *Proceedings of the 12th International Workshop on Network and Operating Systems Support for Digital Audio and Video*, NOSSDAV '02, page 23–29, New York, NY, USA, 2002. Association for Computing Machinery.

[42] Parveen Patel, Deepak Bansal, Lihua Yuan, Ashwin Murthy, Albert Greenberg, David A. Maltz, Randy Kern, Hemant Kumar, Marios Zikos, Hongyu Wu, Changhoon Kim, and Naveen Karri. Ananta: Cloud scale load balancing. In *Proceedings of SIGCOMM 2013*.

[43] Y. Rekhter, T. Li, and S. Hares. Rfc 4271: A border gateway protocol 4 (bgp-4), 2006.

[44] Brandon Schlinker, Hyojeong Kim, Timothy Cui, Ethan Katz-Bassett, Harsha V. Madhyastha, Italo Cunha, James Quinn, Saif Hasan, Petr Lapukhov, and Hongyi Zeng. Engineering egress with edge fabric: Steering oceans of content to the world. In *Proceedings of SIGCOMM 2017*, New York, NY, USA, 2017. Association for Computing Machinery.

[45] Hua Shao, Xiaoliang Wang, Yuanwei Lu, Yanbo Yu, Shengli Zheng, and Youjian Zhao. Accessing Cloud with Disaggregated Software-Defined Router. In *Proceedings of NSDI 2021*.

[46] Xiaoyi Shi, Lin He, Jiasheng Zhou, Yifan Yang, and Ying Liu. Miresga: Accelerating layer-7 load balancing with programmable switches. In *Proceedings of the ACM on Web Conference 2025*, WWW '25, page 2424–2434, New York, NY, USA, 2025. Association for Computing Machinery.

[47] Jeongseok Son, Yongqiang Xiong, Kun Tan, Paul Wang, Ze Gan, and Sue Moon. Protego: Cloud-Scale multitenant IPsec gateway. In *Proceedings of ATC 2017*.

[48] Cha Hwan Song, Xin Zhe Khooi, Raj Joshi, Inho Choi, Jialin Li, and Mun Choon Chan. Network load balancing with in-network reordering support for rdma. In *Proceedings of the ACM SIGCOMM 2023 Conference*, ACM SIGCOMM '23, page 816–831, New York, NY, USA, 2023. Association for Computing Machinery.

[49] Tecent. Session Persistence. https://www.tencentcloud.com/document/product/214/6154?lang=en, 2025.

[50] Tencent. Cloud Load Balance. https://intl.cloud.tencent.com/document/product/214, 2025.

[51] Tencent. Elastic IP addresses. https://intl.cloud.tencent.com/document/product/213/5733, 2025.

[52] Tencent. Tencent Cloud Services. https://www.tencentcloud.com, 2025.

[53] Kashi Venkatesh Vishwanath and Nachiappan Nagappan. Characterizing cloud computing hardware reliability. In *Proceedings of SoCC 2010*.

[54] VPP. The Vector Packet Processing platform. https://github.com/FDio/vpp, 2025.

[55] XDP. The eXpress Data Path inside the Linux kernel. https://github.com/xdp-project/, 2025.

[56] Kok-Kiong Yap, Murtaza Motiwala, Jeremy Rahe, Steve Padgett, Matthew Holliman, Gary Baldus, Marcus Hines, Taeeun Kim, Ashok Narayanan, Ankur Jain, Victor Lin, Colin Rice, Brian Rogan, Arjun Singh, Bert Tanaka, Manish Verma, Puneet Sood, Mukarram Tariq, Matt Tierney, Dzevad Trumic, Vytautas Valancius, Calvin Ying, Mahesh Kallahalla, Bikash Koley, and Amin Vahdat. Taking the edge off with espresso: Scale, reliability and programmability for global internet peering. In *Proceedings of SIGCOMM 2017*, New York, NY, USA, 2017. Association for Computing Machinery.

[57] Chaoliang Zeng, Layong Luo, Teng Zhang, Zilong Wang, Luyang Li, Wenchen Han, Nan Chen, Lebing Wan, Lichao Liu, Zhipeng Ding, Xiongfei Geng, Tao Feng, Feng Ning, Kai Chen, and Chuanxiong Guo. Tiara: A scalable and efficient hardware acceleration architecture for stateful layer-4 load balancing. In *Proceedings of NSDI 2022*.

[58] Menghao Zhang, Jun Bi, Kai Gao, Yi Qiao, Guanyu Li, Xiao Kong, Zhaogeng Li, and Hongxin Hu. Tripod: Towards a scalable, efficient and resilient cloud gateway. *IEEE Journal on Selected Areas in Communications*, 37(3):570–585, 2019.

[59] Menghao Zhang, Guanyu Li, Shicheng Wang, Chang Liu, Ang Chen, Hongxin Hu, Guofei Gu, Qi Li, Mingwei Xu, and Jianping Wu. Poseidon: Mitigating volumetric ddos attacks with programmable switches. In *Proceedings of NDSS 2020*.

[60] Jiasheng Zhou, Lin He, Yifan Yang, Xiaoyi Shi, Daguo Cheng, Jinlong E, Ying Liu, and Dong Zhang. 6map: Enabling fast active ipv6 address discovery with programmable switches. In *Proceedings of the 44th IEEE Conference on Computer Communications (INFOCOM 2025)*, London, United Kingdom, 2025.

Appendix

Appendices are supporting material that has not been peer-reviewed.

A Acronym Explanation

We provide explanations for technical acronyms whose full names do not appear in the main text, as shown in Table 1.

Acronym	Explanation
DPDK	Data Plane Development Kit, a high-performance packet processing library bypassing the Linux kernel protocol stack.
XDP	eXpress Data Path, a kernel integrated packet processor based on eBPF.
VPP	Vector Packet Processor, a packet processing framework providing rich routing functions, supporting DPDK as the forwarding plane.
GRE	Genetic Routing Encapsulation protocol that stipulates how datagrams under one network protocol are encapsulated in another.
BGP	Border Gateway Protocol, a routing protocol used to exchange network layer reachability information between routing domains.
BFD	Bidirectional Forwarding Detection, a high-speed network fault detection mechanism for fault recovery.
ECMP	Equal-Cost Multi-Path, a flow-based load balancing strategy, enabling packets with the same destination to be forwarded to different equivalent optimal next hops.
QUIC	Quick UDP Internet Connection, a multiplexed transport protocol built on top of UDP that aims to improve the speed and reliability of the network connection.
HTTP	Hypertext Transfer Protocol, a client-server request response protocol.
CapEx	Capital expenditures.
OpEx	Operating expense.

Table 1: Acronym explanation.

MARC: Motion-Aware Rate Control for Mobile E-commerce Cloud Rendering

Yuankang Zhao[A§*], *Furong Yang*[A*], *Gerui Lv*[§*], *Qinghua Wu*[§‡†], *Yanmei Liu*[A†], *Jiuhai Zhang*[A], *Yutang Peng*[A], *Feng Peng*[A], *Hongyu Guo*[A], *Ying Chen*[A], *Zhenyu Li*[§‡], *Gaogang Xie*[§¶†]

[A] *Alibaba Group* [§] *University of Chinese Academy of Sciences* [‡] *Purple Mountain Laboratories*

[¶] *Computer Network Information Center, Chinese Academy of Sciences*

Abstract

Mobile e-commerce platforms increasingly integrate cloud rendering to deliver immersive 3D shopping experiences, where users interact with the rendered scenes through the network. Our large-scale online measurements reveal that users' Quality of Experience (QoE) preferences dynamically evolve with user motions in cloud rendering sessions. However, latency spikes occur more frequently during peak periods of user engagement, resulting in early session abandonment. To address this issue, we propose *MARC*, a motion-aware rate control framework that aligns bitrate decisions with user QoE preferences in real-time. *MARC* sets dynamic QoE objectives based on real-world user engagement behavior, captures the different latency and quality requirements for motion and non-motion frames, and employs stochastic optimization to maximize QoE. Extensive deployment of over 1 million user sessions demonstrates that *MARC* reduces session freeze rates by 71% and increases user interaction time by 20%, significantly improving user engagement for e-commerce cloud rendering.

1 Introduction

As mobile e-commerce continues to flourish, an increasing number of merchants are turning to 3D rendering to deliver immersive product experiences [22,41,50]. By enabling users to interact with high-fidelity 3D models in real-time on their local devices, these technologies increase user engagement and purchase conversions [5, 67]. However, rendering 3D content locally requires substantial computing power, which conflicts with the limited performance and battery capacity of mobile devices. Additionally, downloading high-fidelity models imposes significant storage requirements and introduces considerable loading times. Cloud rendering has emerged as an attractive solution by offloading the computationally intensive rendering tasks to powerful remote servers and transmitting only the resulting video frames to the user. As a result, cloud rendering reduces the client's processing load and improves the overall user experience.

A key determinant of this user experience is the balance between motion-to-photon (MTP) latency and visual quality. Increasing the bitrate improves quality, but can also inflate MTP latency by causing network congestion. To this end, cloud rendering systems often use real-time communication (RTC) protocols such as WebRTC [6, 45]. Central to such systems is the rate controller [20, 37, 68], which adjusts the encoding bitrate based on estimated network bandwidth.

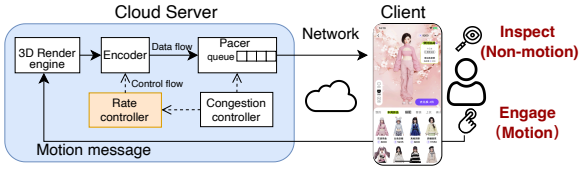


Figure 1: Architecture of the cloud rendering system. User behavior, including engage (motion) and inspect (non-motion) actions, follows an on-off pattern.

Although the cloud rendering system (Fig. 1) shares a similar technical framework with cloud gaming [53, 60, 69], our measurement on Taobao's mobile cloud rendering platform involving more than 100,000 users reveals unique QoE requirements for cloud rendering (§ 2.2). The results indicate that even with advanced optimization techniques, *users experience excessively high latency precisely when they are most actively engaged*, leading to early session abandonment. Specifically, user behavior follows an on-off pattern, alternating between "motion" phases (e.g., rotating or zooming in on a 3D product) and "non-motion" phases (e.g., inspection without manipulation). During motion phases, when users are most sensitive to high latency, video frames tend to be larger (22% on average) and tail latency increases significantly (1.9 times higher at the 99th percentile). Nevertheless, existing rate controllers, which use the same decision logic for both motion and non-motion phases, fail to distinguish between the different behaviors and QoE preferences of the two. Therefore, these solutions cannot optimize QoE for cloud rendering systems (§ 2.3).

Our observations highlight a key insight: *QoE requirements change dynamically with user motion, necessitating "motion-aware" rate control strategies.* Despite its simplicity, translating this insight into a practical system presents several design challenges:

***(i)* Mapping user engagement to QoE preference.** As observed in § 2.2, users prioritize low latency over high quality during motion phases, characterized by user engagement (e.g., session duration) decreasing more rapidly as latency increases.

*Co-first authors
†Corresponding authors

However, how to quantify this human-involved engagement pattern as a guideline for rate control remains unclear.

***(ii)* Striking a dynamic balance in bitrate adaptation.** The primary challenge in rate control is resolving the conflict between minimizing latency and maximizing quality under network dynamics. In addition, our measurements show that user behavior introduces QoE preference dynamics. These two types of system dynamics make it much more difficult to optimize QoE in e-commerce cloud rendering.

***(iii)* Enabling frame-level fine-grained decision.** Since user behavior exhibits an on-off pattern across frames, the rate controller must operate at the frame level, e.g., within 33.3 ms at a frame rate of 30 fps. This requirement imposes a severe time constraint on the decision logic.

To address the above challenges, we propose *MARC*, a dynamic QoE-driven **Motion-Aware Rate Control** framework for e-commerce cloud rendering. *MARC* sets a dynamic QoE objective function, which derives the importance of motion and non-motion frames directly from our measurements of user engagement as QoE preferences. *MARC* further models the dynamic QoE evolution of multiple future frames and utilizes stochastic optimization techniques to maximize cumulative QoE. In doing so, *MARC* incorporates a state predictor to predict user behavior and network conditions. Furthermore, *MARC* employs an efficient gradient-based method to determine the bitrate at the frame level.

We have implemented *MARC* in the WebRTC framework and deployed it in our real-world system. Extensive experiments based on simulation and large-scale A/B tests involving over 1 million user sessions demonstrate that *MARC* outperforms the state-of-the-art RTC rate control methods [6, 26, 28, 48]. Specifically, compared to baselines, *MARC* reduces the tail send duration and queueing latency for motion frames by 30% to 55%, tail frame latency by 22% to 60%, and session freeze rate by 71%. *MARC* also improves user engagement, with session durations increasing by 9% and interaction rates (i.e., the ratio of motion frames in all frames) rising by 20%. Moreover, *MARC* incurs no overhead on the client side and adds only a modest 1.3% CPU usage per session on the server side. These results confirm that *MARC* can achieve effective bitrate control in large-scale e-commerce cloud rendering systems.

To summarize, the main contributions of this paper are:

- Identifying unique dynamic QoE requirements that evolve with user motion in e-commerce cloud rendering through a large-scale measurement study;
- Proposing a dynamic QoE-driven motion-aware rate control framework *MARC*, which integrates user-behavior modeling and multi-frame QoE optimization;
- Implementing *MARC* in production, ensuring frame-level bitrate adaptation under strict time constraints;
- Extensively evaluating *MARC* in a real-world cloud rendering system and demonstrating its efficiency.

Claim: This work does not raise any ethical issues. All data collected in this research are desensitized and include performance-related information only.

2 Background and Motivation

2.1 Background: Cloud Rendering in Mobile E-commerce Applications

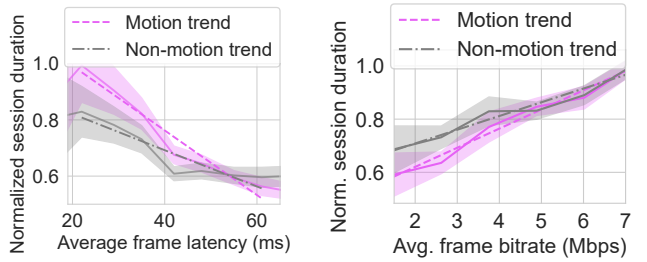
Cloud rendering is an emerging technology adopted by e-commerce platforms to deliver interactive 3D visuals on mobile devices [22, 41, 50]. This technology increases user engagement by allowing users to interact with their favorite products and avatars in real-time, potentially boosting the revenue of e-commerce companies [5, 67].

Cloud rendering system architecture. A typical cloud rendering architecture is illustrated in Fig. 1. The mobile device acts as the client, capturing user motions for the server. Here, *motion* denotes intentional interaction events such as tap, pinch-to-zoom, or drag gestures that generate explicit control commands. The cloud server deploys a powerful 3D rendering engine (e.g., Unreal Engine [14]) to generate high-quality 3D graphics, which are then compressed into a video stream and transmitted to the client for playout. Inside the server, the encoder converts the rendered graphics into video frames that are further packaged into media packets. These packets are queued at the pacer, awaiting transmission over the network. The pacer follows the transport rate set by the congestion controller (e.g., GCC [6], Copa [3], Pudica [53], or SQP [48]), which estimates available bandwidth to avoid network congestion. The rate controller decides the target encoding bitrate based on the bandwidth estimation from the congestion controller.

Rate control for real-time communications. Video rate control significantly affects user experience [10, 68]. While higher bitrates can improve video quality and thus user satisfaction [10], they may result in increased latency due to inflating the send queue, especially when a sudden drop in bandwidth causes the encoder's output rate to surpass the transmission rate. To maintain low latency, current encoder rate controllers typically reserve a fixed portion of the available bandwidth as headroom. For example, commercial interactive video applications set the target bitrate 11% to 28% lower than the estimated bandwidth [28], which, however, sacrifices video quality. Therefore, video rate control must be carefully designed to balance quality and latency, which further relies on a deep understanding of the relationship between these QoE metrics and user engagement.

2.2 Characterizing User Behavior and QoE Preferences in the Wild

To investigate how the above QoE metrics affect user engagement in real-world cloud rendering systems, we conduct



(a) Frame latency vs. session duration.

(b) Frame bitrate vs. session duration.

Figure 2: Large-scale QoE study: Relationship between QoE metrics and session duration.

a large-scale QoE study on Taobao's mobile cloud rendering platform.

System setup. Our cloud rendering system allows users to dress up their 3D avatars, create dresses, and change background scenes. The server employs Unreal Engine (UE) [14] to render 3D graphics and leverages Pixel Streaming [15], a UE plugin, to stream these graphics to the client via WebRTC [45]. We use WebRTC because it is the de facto infrastructure for real-time video streaming [21] and is widely supported by major web browsers and platforms [58], facilitating large-scale production deployments. When users access the service, they are assigned to the nearest rendering server and then connected by WebRTC. Users can interact with the virtual characters on the screen, and their actions (e.g., motions like zooming in and out or changing viewpoints) are sent to the server via the WebRTC Datachannel using the SCTP [1].

Video encoding. Following common practice [34], our system operates at a fixed video resolution of 1560x720 with a frame rate of 30 fps and a maximum bitrate of 8 Mbps. The encoder works in CBR (constant bitrate) mode. We employ an infinite Group of Pictures (GOP) size, resulting in 99.9% of the frames being P-frames (non-keyframes). P-frames rely on preceding frames for decoding, which reduces redundancy and ensures efficient compression. Only 0.1% of the frames are I-frames (keyframes), which typically appear at the start of the session to initialize decoding. The stream is encoded in H.264 format, which is preferred for its superior decoding performance and energy efficiency, leveraging client-side hardware decoders.

Incorporating advanced optimization. Our cloud rendering system shares a similar technical infrastructure with cloud gaming [53, 60, 69]. Therefore, this system uses the same cutting-edge optimization techniques as cloud gaming. Specifically, edge servers strategically deployed near users [60], equipped with industry-leading hardware encoders [39] that support ultra-low latency and constant bitrate encoding modes [2]. The online configuration also includes pacing-based sending [56] (discussion in § 6) and forward error correction (FEC) [4]. Furthermore, the RTC framework incorporates the standard low-latency congestion control algorithm and rate controller [6], as well as innovative client-side buffer management [67].

Measurement methodology. We measure user engagement by two behavioral interaction metrics [42]: *(i) session duration*, which quantifies how much time a user spends on the service in a session; and *(ii) motion phase duration*, which is equal to the total duration of motion frames. A frame is marked as a motion frame whenever the server receives a user motion command from the client (see § 4.1 for details). Therefore, each motion phase contains a single frame or a series of consecutive motion frames[1]. In addition, other frame-level metrics are also collected, including the target bitrate, actual frame size, and send duration (the time taken from sending the first packet to sending the last packet of the frame). All metrics are collected on the server side and do not include any user-related or content-related information.

Dataset description. The measurement spanned from February 23 to March 27, 2024, covering 1.8 billion frames across 200,846 sessions, accumulating over 16,000 hours of video, and involving more than 100,000 users. To the best of our knowledge, this dataset represents the first large-scale and comprehensive measurement of a real-world cloud rendering system, uniquely capturing both video stream metrics and complex user motion behavior. By analyzing the dataset, we have made the following observations.

***Observation 1*: In motion phases, frame latency has a greater impact on user engagement than bitrate.**

We first focus on the relationship between QoE metrics (i.e., frame latency and bitrate) and user engagement in both the motion and non-motion phases, as illustrated in Fig. 2, where translucent areas represent 95% confidence intervals.

(i) **Frame latency**. It is defined as the time it takes for a frame to be sent from the server to be fully received by the client, namely the send duration plus the one-way delay (OWD), where OWD is estimated as half the round-trip time (RTT) on the server side. Although MTP latency also includes components from the rendering engine and client side, prior work [67] and our measurements indicate that client-side optimizations reduce its contribution to only 22% of total MTP. Consequently, the frame latency remains the dominant contributor.

In Fig. 2a, the trend lines represent linear regressions of the session duration data. It can be seen that session duration decreases as latency increases, which is consistent with the previous studies [53]. Additionally, the slope of the trend line is 75.7% steeper in motion phases than in non-motion phases. Looking deeper, increasing the latency of motion frames consistently decreases session duration. In contrast, once the latency of non-motion frames reaches a certain point (i.e., 42 ms in Fig. 2a), the session duration remains relatively stable.

[1]We have excluded single-frame user motions in our analysis to avoid the interferences from users' unconsciously tapping on the screen.

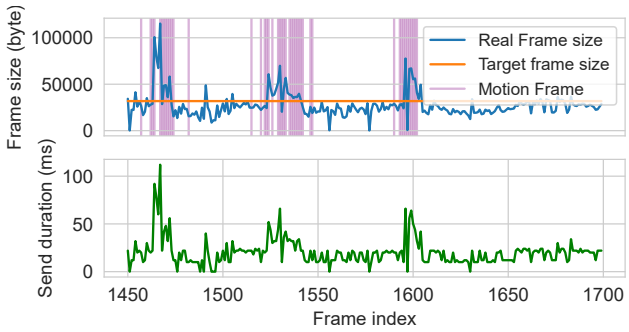
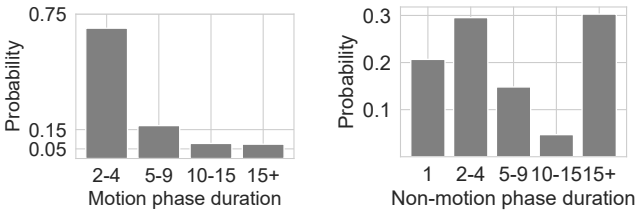


Figure 3: The on-off pattern of user motions.



(a) Motion phase. (b) Non-motion phase.

Figure 4: The probability distribution of consecutive motion/non-motion frames.

These results suggest that *users are much more sensitive to the high latency of motion frames.*

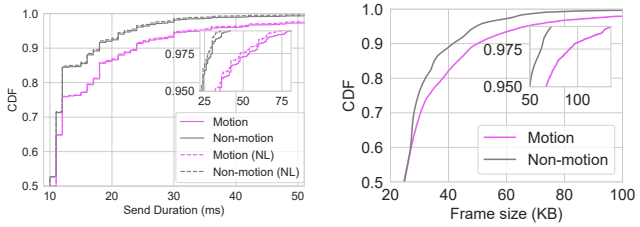
(ii) **Frame bitrate.** As shown in Fig. 2b, session duration increases with frame bitrate. Overall, the slope is 37.8% steeper in motion phases than in non-motion phases. However, it is worth noting that when the frame bitrate is high enough (i.e., over 5 Mbps), session duration is hardly affected by user motion.

***Observation 2*: User behavior exhibits an "on-off" pattern, with short "on" (motion) and long "off" (non-motion) periods.**

Fig. 3 illustrates the "on-off" pattern of user behavior, where the "on" phase indicates successive motion frames while the "off" phase corresponds to the intervals between non-motion frames. Fig. 4 depicts the duration distribution of motion and non-motion phases. Motion phases are relatively short: 74% of them last 2-4 frames, while 15% last 5-9 frames (Fig. 4a). A non-motion phase may appear between two adjacent motion phases. 70% of non-motion phases are brief pauses (no more than 15 frames, i.e., 0.5 seconds), and 30% are prolonged stops (over 15 frames), as shown in Fig. 4b. This behavior pattern is typical in e-commerce applications, where users frequently change viewpoints and then pause to view their avatars or products.

***Observation 3*: Despite the same target bitrate given by the rate controller, motion frames tend to have larger sizes and longer send durations than non-motion frames.**

As defined, frame latency contains the send duration and OWD. In cloud rendering systems, OWD is typically very low,



(a) CDF of send duration. (b) CDF of frame size.

Figure 5: Send duration and frame size distributions for motion and non-motion frames. Motion increases the tail frame size and the send duration.

with a 95th percentile of only 29.5 ms in our measurements. This result is consistent with previous studies [60, 61], which can be explained by the fact that servers are edge-deployed and thus close to users. Nevertheless, the send duration is significantly longer, with a 95th percentile of 59 ms, which is 2 times the OWD. Therefore, we next examine the send duration, which is directly affected by the rate controller.

As shown in Fig. 3, the send duration generally increases with the actual frame size, which is different for motion and non-motion frames. The statistical results in Fig. 5 further support this intuition. Fig. 5a shows that the send duration of motion frames is consistently longer than that of non-motion frames after the 50th percentile (P50). In particular, the P99 send duration of motion frames can reach up to 82 ms, which is 1.9 times longer than that of non-motion frames, resulting in significant latency during motion phases.

Essentially, the root cause of increased send duration during motion phases is that *motion frames tend to generate more data than expected*. As a result, the actual bitrate of motion frames is more likely to exceed the congestion controller's pacing rate, causing packets to queue up at the sender and ultimately increasing the send duration. Fig. 5b indicates that motion frames are significantly larger than non-motion frames after the P50. Remember that both motion and non-motion frames use CBR encoding, and most of them are not keyframes. Therefore, the difference in their sizes is because motion frames have more variable content and require more bits to encode[2], as reported in previous work [47]. Notably, packet loss is not the main factor affecting send duration. Even in no-loss (NL) sessions, the tail (P99) send duration of motion frames can still reach 72 ms, which is 87.8% of the 82 ms in packet-loss sessions.

Unfortunately, existing rate controllers use the same decision logic to determine the target bitrate for both motion and non-motion frames, and thus cannot distinguish between their different behaviors and QoE requirements.

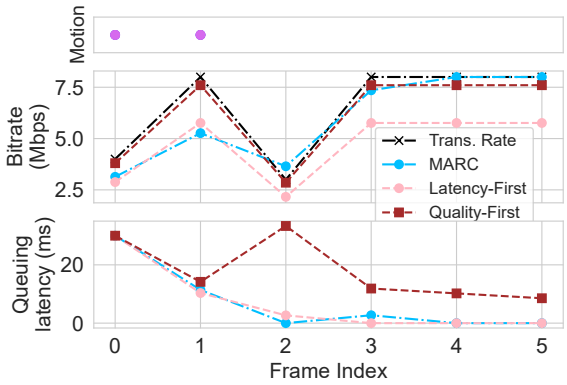


Figure 6: Frame bitrate and queuing latency. Queuing latency is the waiting time before transmission, determined by queue length, previous frame size, and pacing rate.

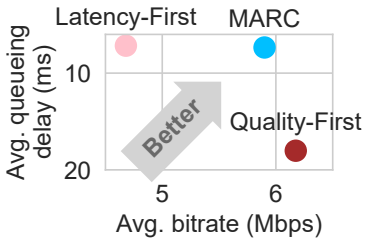


Figure 7: QoE performance comparison of *MARC* vs. existing rate controllers.

2.3 Pitfalls of Existing Rate Controllers

We provide a case study to illustrate how existing solutions fail to optimize QoE in the cloud rendering system due to ignorance of user motion.

Existing rate controllers typically use a fixed portion of estimated available bandwidth as video bitrate [28]. Two baseline strategies are considered in this controlled experiment (details in § 5.1): *(i) Latency-First* strategy, which allocates 72% of bandwidth as video bitrate (based on the measurement of a commercial video application in [28]); and *(ii) Quality-First* strategy, which uses 95% of bandwidth as video bitrate (an estimated value of GCC's [6] decision).

We also introduce our proposed method *MARC* (§ 3) as a comparison. To eliminate the effect of network conditions, we feed all strategies with the exact future bandwidth in the trace. In addition, all strategies begin with the same state, including user motion sequences and an initial sender buffer length of 15 KB (implying that a packet queue already exists on the sender). The performance is evaluated based on both *bitrate* allocation and *frame queuing latency*. Frame queuing latency is affected by the send duration of previous frames, referring to the time a frame spends waiting in the send queue before it is transmitted.

[2]However, this issue cannot be directly solved by modifying the video encoder, as discussed in § 6.

Fig. 6 illustrates the decision-making traces of these strategies. Note that frames #0 and #1 are motion frames. The Quality-First strategy outputs frames at higher bitrates and therefore struggles to drain the send queue quickly, especially when bandwidth suddenly drops (frame #2). On the other hand, the Latency-First strategy reduces queuing latency but does not effectively utilize the available bandwidth. While the queue is empty (frame #2), the Latency-First strategy still outputs low-level bitrates that reduce perceived quality.

Unlike the above, *MARC* maintains awareness of user motion. If the queueing latency is high, *MARC* timely reduces the bitrate/bandwidth ratio for motion frames (frames #0 and #1). Otherwise, for non-motion frames with low queueing latency, *MARC* chooses higher bitrates to improve quality (frames #2 to #5). As a result, *MARC* achieves the best balance between latency control and bandwidth utilization. Fig. 7 shows that compared to the Latency-First strategy, *MARC* improves the average bitrate by 26.1% for a similar average latency (2.5% higher). *MARC* also reduces queueing latency by 59.2% over the Quality-First strategy with only a 4.5% bitrate reduction.

2.4 Implications for Video Rate Control

As described above, we have identified characteristics of user behavior and factors that impact user engagement in mobile e-commerce cloud rendering systems. All three observations have important implications for rate control:

***Requirement 1*: Awareness of user motion.** Users are more sensitive to the latency of motion frames (Observation 1). Hence, the rate controller should be able to sense the motion phase, in order to dynamically adjust the bitrate based on the user's preferences.

***Requirement 2*: Frame-level decision.** Observation 2 indicates that the duration of the motion phase is typically short. Consequently, a fine-grained, frame-level rate control mechanism is essential to swiftly adapt to the short motion intervals and avoid unnecessary quality reductions or latency buildups.

***Requirement 3*: Differentiated bitrate assignment.** Given that motion frames are generally larger than non-motion frames (Observation 3), the rate controller should assign lower bitrates to motion frames to avoid excessive latency.

Summary. The technical framework of e-commerce cloud rendering system seems similar to that of mobile cloud gaming [18, 53], which typically needs a consistently low latency of 50 ms to 150 ms [9, 23] for all video frames. However, our large-scale QoE study reveals that *the QoE requirements of cloud rendering are quite unique*. Specifically, a cloud rendering session can be divided into alternating motion and non-motion phases. Users' QoE preferences differ significantly in these two phases. Therefore, the rate controller should be *motion-aware* to optimize dynamic QoE objectives. Unfortunately, existing solutions [8, 26, 28, 48] overlook this principle, which motivates our work in this paper.

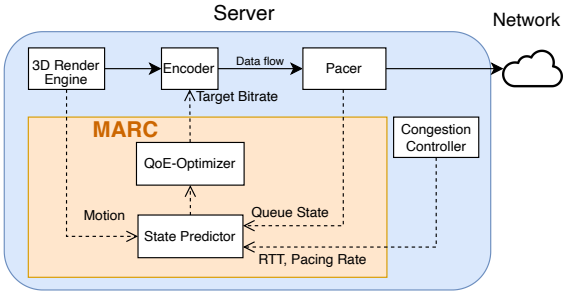


Figure 8: *MARC* system overview.

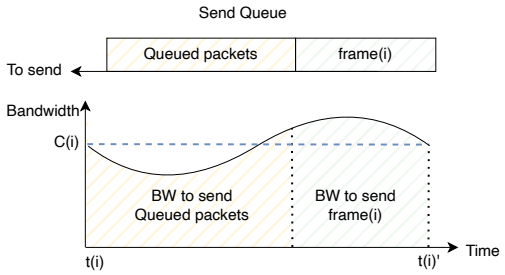


Figure 9: The relationship of frame enqueue time $t(i)$, frame departure time $t(i)'$, and avg. pacing rate $C(i)$.

3 *MARC* System Design

Based on our insights in § 2, we propose a Motion-Aware Rate Controller (*MARC*) for mobile e-commerce cloud rendering systems. *MARC* aims to maximize QoE by controlling the video encoding bitrate based on dynamic user preferences for quality and latency.

3.1 System Overview

Fig. 8 illustrates the high-level system architecture of *MARC*, which integrates several key components. First, the **Dynamic QoE Optimizer** (§ 3.5) determines the video bitrate on a per-frame basis, leveraging our time-based sending model (§ 3.3) and dynamic QoE objectives (§ 3.4). Second, the **State Predictor** (§ 3.6) provides short-horizon forecasts of user motion, pacing rate, and propagation delay.

Together, these components fulfill the three design requirements identified in § 2.4. Specifically, MARC is aware of user motion and makes frame-level decisions to optimize dynamic QoE preferences according to different user behaviors.

3.2 Mapping User Engagement to QoE Preference

Our analysis derives frame importance based on detailed online user interaction observations. We measure user engagement via *session duration* and study its relationship with latency and visual quality (bitrate). Specifically, we bin frames by latency/bitrate ranges, compute the average session duration in each bin, and then perform linear regressions separately for motion and non-motion frames, see Fig. 2.

Key findings. We observe that motion frames exhibit higher sensitivity to latency, with a slope ratio (non-motion : motion) $\approx 1:1.757$. Similarly, they also show moderately higher sensitivity to quality ($1:1.378$). This indicates that minimizing latency is particularly critical during motion phases, while maintaining adequate quality remains important for overall engagement.

3.3 Time-Based Formulation for Frame Sending Process

Our frame-level model captures the entire process from frame generation to transmission, accounting for encoding decisions, queuing, and transmission.

Frame generation and encoding. Video frames are generated at a nearly constant interval (e.g., 33 ms at 30 fps), denoted as I. For the i-th encoded frame, the encoding bitrate R_i is defined as Eq. (1). The encoded frame size is denoted by $d(R_i)$. Unlike typical VoD or live streaming scenarios [63,68] that have a set of discrete bitrates, *MARC* operates over a *continuous* bitrate space.

$$R_i \in \mathcal{R}, \quad \forall i = 1, \cdots, N. \tag{1}$$

In our implementation, we empirically derive $d(R_i)$ by performing separate linear regressions for motion and non-motion frames, based on real encoder output data. This approach more accurately captures the typically larger size of motion frames for the same bitrate, while still preserving a continuous domain for optimization.

Queuing and transmission. Fig. 9 illustrates the queueing and transmission process. The encoded frame(i) is packetized and added to the transmission queue, where the queue length is $b(i)$. We mark the moment of enqueueing as time $t(i)$. The frame(i) waits in the send queue until the previously queued packets have been transmitted. The time when the frame(i) finishes sending is marked as $t(i)'$. The duration from $t(i)$ to $t(i)'$, denoted as $s(i)$, corresponds to an average pacing rate $C(i)$ as defined in Eq. (4). This process is formulated in Eq. (2) and (3).

$$t(i)' = t(i) + s(i) \tag{2}$$

$$s(i) = \frac{b(i) + d(R_i)}{C(i)} \tag{3}$$

$$C(i) = \frac{1}{t(i)' - t_i} \int_{t_i}^{t(i)'} C_t dt, \tag{4}$$

Buffer state dynamics. We formulate send-buffer evolution by modeling the residual queue length $b(i)$, as shown in

Eq. (5). Here, $(x)_+$ means $\max(x,0)$, ensuring a non-negative queue size.

$$b(i) = \left(b(i-1) + d(R_{i-1}) - \int_{t_{i-1}}^{t_i} C_t dt \right)_+ \quad (5)$$

Total frame latency. The total frame latency includes queuing delay, send duration, and network propagation latency (denoted as $OWD(i)$) as defined in Eq. (6). Note that this differs from the "frame latency" defined in § 2, which does not account for queueing delay.

$$\mathrm{L}(i) = OWD(i) + s(i) \quad (6)$$

3.4 Dynamic QoE Objective Function

Based on the linear regression slopes in § 3.2, we normalize the base *non-motion* quality weight to 1.0, then we derive a *motion-to-non-motion* latency ratio of 1.275 (i.e., $\frac{1.757}{1.378}$). Concretely, for each frame i, we define:

$$\mathrm{QoE}(i) = q_R\big(R_i\big) - \gamma(i) \cdot q_L\big(L(i)\big),$$

where $q_R(\cdot)$ and $q_L(\cdot)$ are the respective quality and latency scoring functions. The latency penalty coefficient $\gamma(i)$ depends on whether the frame is non-motion ($M(i) = 0$) or motion ($M(i) = 1$):

$$\gamma(i) \;=\; \begin{cases} 1 & \text{if } M(i) = 0 \\ 1.275 & \text{if } M(i) = 1 \end{cases} \quad (7)$$

Equivalently, we may write $\gamma(i) = \lambda_s + M(i) \cdot \lambda_m$ with $\lambda_s = 1$ and $\lambda_m = 0.275$. (The impact of the two parameters is evaluated in § 5.2.)

To capture overall QoE across N consecutive frames, we sum the single-frame QoE as in Eq. (8):

$$QoE_1^N = \sum_{i=1}^{i=N} q_R(R_i) - \sum_{i=1}^{i=N} (\lambda_s + M(i) \times \lambda_m) \times q_L(L(i)) \quad (8)$$

This formulation *dynamically* places more weight on latency reduction in motion frames, aligning with users' observed engagement patterns.

3.5 Maximizing QoE with Stochastic Optimization

Given the constraints imposed by the frame-sending model and the QoE objective, we formulate the rate control problem as a QoE maximization problem:

$$\max_{R_1,\cdots,R_N} QoE_1^N \quad (9)$$
$$s.t. \text{ Equation (1) to (6)}$$

The *MARC* algorithm. The algorithm 1 shows a high-level overview of the *MARC*. Tab. 1 lists the key variables in the algorithm. The algorithm runs whenever a frame is encoded. It determines the target bitrate for *the next* frame. Thus, the decision time for the algorithm is about a frame interval (e.g., 33 ms in 30 fps). The algorithm employs an N-step lookahead strategy (moving horizon) to address a specific QoE maximization problem. The window size is set to $N = 10$, a fine-tuned parameter striking a good balance between performance and overhead, as evaluated in § 5.4 and Appendix A. This involves predicting motion ($\hat{M}_{[i,i+N]}$), propagation latency ($\hat{OWD}_{[i,i+N]}$), and pacing rate ($\hat{C}_{[i,i+N]}$) by the state predictor.

Table 1: Key Variables of the Model

Notation	Meaning
$q_R(R)$	Quality evaluation function
$q_L(L)$	Latency evaluation function
λ_m	Weight for motion frames
λ_s	Weight for non-motion frames
I	Frame interval (ms)
$d(R_i)$	Size of frame i
$b(i)$	Size of send queue when at $t(i)$
$t(i)$	Time when frame i is packetized
$s(i)$	Send time of frame i
$t(i)'$	Time when frame i departs from the send queue
$C(i)$	Average pacing rate from $t(i)$ to $t(i)'$
$L(i)$	Motion-to-photon latency of frame i
$M(i)$	Whether frame i is a motion frame (0 or 1)
C_t	Pacing rate at time t
$OWD(i)$	One way delay while sending frame i

Algorithm 1 *MARC*: Motion-aware Rate Control

input: $M_{[i-N,i]}$, $OWD_{[i-N,i]}$, $C_{[i-N,i]}$, λ_s, λ_m, $b(i)$
output: R_{i+1}: Encoder target bitrate for the next frame
On frame encoded

1: $\hat{M}_{[i,i+N]} = \texttt{MotionPredict}\,(M_{[i-N,i]})$
2: $\hat{OWD}_{[i,i+N]} = \texttt{OWDPredict}\,(OWD_{[i-N,i]})$
3: $\hat{C}_{[i,i+N]} = \texttt{PacingRatePredict}\,(C_{[i-N,i]})$
4: $R_{i+1} = f_{MARC}(b(i), \hat{M}_{[i,i+N]}, \hat{OWD}_{[i,i+N]}, \hat{C}_{[i,i+N]})$

In **Algorithm 1**, four functions act as the core building blocks. `MotionPredict(·)` takes the observed motion labels from the past N frames and predicts future labels ($\hat{M}[i, i+N]$). `OWDPredict(·)` uses recent RTT measurements to forecast propagation delay ($\hat{OWD}[i, i+N]$). `PacingRatePredict(·)` analyzes pacing rates from the previous N frames to estimate upcoming sending rates ($\hat{C}[i, i+N]$). Finally, $f_{MARC}(\cdot)$ combines these predictions with the current queue state $b(i)$ to determine the next-frame bitrate R_{i+1}, maximizing QoE by balancing latency and visual quality while adjusting to user motion.

3.6 State Predictor

The state predictor is crucial for forecasting user motion, pacing rate, and propagation delay, which are essential inputs for the *MARC* system.

User motion predictor. We have developed a Markovian model that predicts the motion sequence of user frames based on the types of the previous M frames. The model forecasts whether the next frame will be of type 0 or 1 and continues this prediction for future N frames. Its state transition probabilities are derived from statistical data observed online, as observed in § 2.2. Since each frame type is binary, the model encompasses 2^M states. We evaluate the M and the prediction accuracy in § 5.3.

Pacing rate predictor. While Eq. (4) provide a theoretical definition of the average pacing rate C(i), obtaining an exact solution for Eq. (4) in a real-time system is non-trivial. In our implementation, we approximate $C(i)$ by the current sending rate of recent pacing rates. This approximation assumes that over the short interval, the pacing rate does not fluctuate dramatically. Since our system primarily targets short pacer queues and a relatively stable sending rate, such approximation provides sufficient accuracy (see § 5.3).

Propagation delay predictor. We use the smoothed RTT to estimate the propagation delay (which is $1/2$ RTT). Similar to predicting the pacing rate, as the RTT variation within a short period is stable as evaluated in § 5.3. Note that the completion time of a frame may be delayed by packet disorder or jitter. We verified that 99.8% of disordered packets experience delays of less than 16 milliseconds. Therefore, packet disorders within a frame are less likely to severely increase the completion time. Thus, we can disregard the impact of the disorder.

4 System Implementation

4.1 Motion Frame Identification

The Unreal-Engine Pixel Streaming plugin [15] bridges the cloud renderer and its WebRTC transport. It passes video frames from the rendering application down to WebRTC for transmission and forwards user inputs received via WebRTC up to the rendering application. Among these inputs, each motion event is handled by our modified plugin, which forwards the event to a helper function added to WebRTC's `PeerConnectionInterface` [57]. The helper then notifies *MARC*, which marks the next encoded frame as a motion frame.

4.2 *MARC* System Implementation

The dynamic QoE optimization model is the core of the *MARC* system. It involves carefully designed QoE functions and an efficient solving process for real-time optimization. We implemented *MARC* within the PixelStreaming [15] and the WebRTC framework, using 1,246 lines of C++ code.

QoE functions. The principles for designing QoE objective functions are *(i)* they should be continuous and differentiable to allow adjustments in the continuous action space, and *(ii)* they should reflect the user's subjective experience.

Following these principles, we designed a non-linear quality QoE function as in Eq. (10). The first term in this function $(1-\frac{1}{R/a+1})$ characterizes the diminishing marginal effects of increasing bitrate on improving video quality, as observed in Vantage [49]. The second term $(\frac{R_{Max}+a}{R_{Max}})$ normalizes the QoE score to the [0,1] range. In our setting, $a=2, R_{max}=8$ *(Mbps)*. We evaluate the impact of QoE functions in Appendix B.

$$q_R(R) = (1-\frac{1}{R/a+1}) \times \frac{R_{Max}+a}{R_{Max}} \tag{10}$$

Similarly, the latency QoE function is defined as Eq. (11), following the same principles. It imposes minimal penalties for low-latency regions but increases progressively at high-latency regions, consistent with findings in the study [36]. In our setting, we normalize the maximum allowable latency with L_{max} = 150 ms.

$$q_L(L) = (\frac{L}{L_{max}})^2 \tag{11}$$

Continuous rate optimizer. To solve the problem in Eq. (9), we employ the `NLopt` [25] optimization library in C++. The optimizer treats the bitrates of the next N (default $N=10$) frames as real-valued decision variables, each box-constrained to the application-defined range $[R_{\min}, R_{\max}]$ ($[0,8]$ Mbps in our setting). Each call is warm-started with the optimal solution from the preceding frame, while the first frame is initialised with an initial bitrate of 1 Mbps. Optimization terminates when the relative change of all variables falls below 10^{-4}.

Reducing computing overhead. The primary computational cost arises from solving the N-step look-ahead optimization problem, which has a time complexity of O(KN), where K is the number of iterations per step. To improve performance, we *(i)* select a relatively small prediction window of N=10 (determined experimentally as the parameter that best balances overhead and solution quality; the computing overhead is evaluated in § 5.4, the quality is verified in Appendix A); and *(ii)* using `NLopt`'s gradient-based SLSQP [13] backend to reduce the number of iterations required per step (a comparison of different non-linear optimization solvers and their computational overhead is presented in § 5.4). Additionally, we further reduce computing overhead by designing a simple motion state predictor.

Motion state predictor. We represent the state of the M historical frames as the key by encoding the binary sequence $(S_{t-M},\ldots,S_{t-1})$ into an integer $k=\sum_{j=1}^{M} S_{t-j}\cdot 2^{j-1}$ and the probability of the next frame being a motion frame as the value, denoted as p_t, storing this data in a C++ map. The probability is built offline from a large dataset of interaction traces: for each context key k we compute $p_t = \Pr(S_t=1 \mid k) = \frac{N(k,S_t=1)}{N(k)}$, where $N(\cdot)$ counts occurrences in the dataset, thus p_t is the estimated conditional probability that a motion frame follows this context. Predicting the next frame

involves a simple hash lookup for the probability, followed by a Bernoulli draw with parameter p_t. This approach reduces computing overhead for online deployment.

The space complexity of storing the statistical user motion model is $O(2^M)$, where M is the number of historical frames. Each frame state is encoded as 1 bit (1 for motion, 0 for non-motion). The memory used for the map can be shared across multiple sessions, preventing linear scaling of costs as the number of concurrent sessions increases. The storage overhead for a single process is minimal, representing only a 3% increase in memory usage compared to the overall process.

5 Evaluation

This section comprehensively evaluates *MARC* using both trace-driven simulations and large-scale real-world deployments on mobile networks.

5.1 Evaluation Setup

Trace driven evaluation setup. To accurately compare and replicate the bitrate control algorithms, we implemented the simulation environment based on the architecture depicted in Figure 1. This setup allows us to replay traces representing user behavior and network conditions.

Trace collection. We collect traces from our online system. These traces record the application, encoder, and transport models' behaviors. Specifically, the application layer data encompasses the user motions (as measured in § 4.1). The encoder model provides the encoding bitrate of each frame, the frame intervals, and the frame sizes. The transport layer information includes the pacing rate and the Round-Trip Time (RTT) for each of the frame's packets. Our dataset, collected from February 23 to March 27, 2024, comprises over 200,000 sessions and 1.8 billion frames.

Online evaluation setup. We deployed *MARC* in a production environment as described in § 2.2 to assess its real-world performance. To conduct an A/B test, the online experiment randomly assigns each user request to either WebRTC's default rate control algorithm or the proposed *MARC* algorithm.

Baselines. We compare *MARC* with the following bitrate control algorithm as baselines:

- *WebRTC* [45]. The default transmission framework in our environment. It contains the *EncoderBitrateAdjuster* [8] module. The encoder bitrate adjuster dynamically adjusts the target bitrate of the encoder based on the observed average network utilization over a specified period, typically reducing the bitrate if the actual frame sizes exceed expected values.
- *SQP* [48]. SQP uses 90% of available bandwidth as the target bitrate to prioritize low transmission latency. Note that we only compare the rate control strategy of SQP, while we keep the bandwidth estimation strategy the same as WebRTC.
- *Vidaptive* [26]. Vidaptive's rate control logic dynamically adjusts the encoder's rate discount factor α to ensure that the λ^{th} percentile of frame send durations over a defined time window T is closely aligned with the target send duration Δ. We use $\lambda = 90$, $T = 1s$, $\Delta = 33ms$ in the evaluation. To ensure fairness, the Vidaptive frame-skipping strategy is disabled because frame rate control and rate control are orthogonal.
- Commercial video applications models. We utilize the inferred rate control models from the study [28]. Specifically, we employ the video bitrate control model represented by the parameters γ and μ. The parameter γ is a scaling factor that indicates the ratio of video bitrate to available bandwidth, while μ is a constant with units of kbps. We evaluated the performance of all the six application models in the study and selected three representative applications for demonstration: *Zoom* ($\gamma = 0.88$, $\mu = -1.57$), *Duo* ($\gamma = 0.72$, $\mu = -1.91$) and *GoTo Meeting* (abbreviated as *GoTo*) ($\gamma = 0.78$, $\mu = -1.36$). The other models exhibit performance similar to that of these applications.
- MARC *derivatives*. We vary the weight of motion frames λ_m and non-motion frames λ_s.

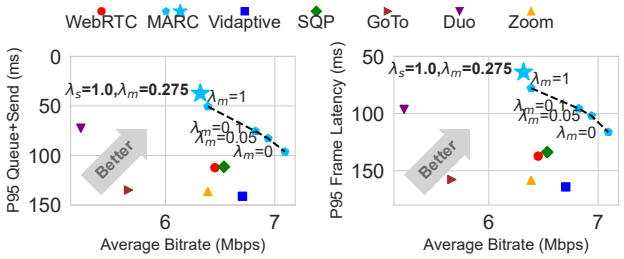
Metrics. We evaluate the quality by the target bitrate of each frame and the visual quality metrics like peak-signal-to-noise-ratio (PSNR), structural similarity index (SSIM) [55], and VMAF [29]. We evaluate latency by the send duration and frame latency.

5.2 Performance Improvement

We evaluate the performance of *MARC* and the baselines in the trace-driven simulation environment. *MARC* extends the performance boundary compared to the baseline algorithms.

Improving performance for motion frames. *MARC* achieves significantly lower motion frame latency compared to multiple baseline algorithms (e.g., Zoom, Duo, WebRTC, Vidaptive, etc.) under similar visual quality conditions. In particular, *MARC* reduces the 95th-percentile (P95) send duration plus queueing latency (shown in Fig. 10a) by 30% to 55% relative to the baselines, thanks to proactively allocating bitrate to consecutive motion frames and thus avoiding excessive queue buildup.

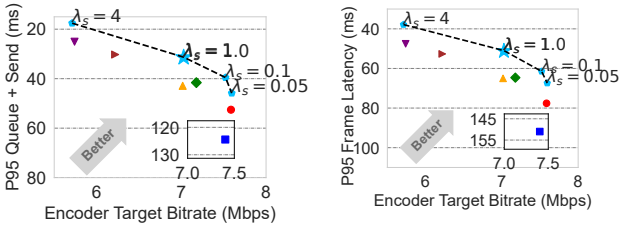
Additionally, *MARC* achieves better visual quality than the baseline under similar latency conditions. For instance, the optimal *MARC* configuration ($\lambda_s = 1, \lambda_m = 0.275$) boosts the average bitrate by 35% over Duo (see Fig. 10a), while simultaneously reducing the P95 frame latency (Fig. 10b) of motion frames by 33–101 ms relative to various baselines. We also examined *MARC*'s derivatives with $\lambda_s = 0.05$ (chosen for its high visual quality in Fig. 11b) and varied λ_m from 0 to 1, these variations are connected by dashed lines. Even without explicit motion awareness ($\lambda_m = 0$), *MARC* achieves the highest average bitrate among all tested methods, while maintaining a markedly lower P95 tail latency than most baselines.



(a) P95 send duration + queueing latency of motion frames.

(b) P95 frame latency of motion frames.

Figure 10: Bitrate and tail latency of motion frames.



(a) P95 queueing latency and send duration of sessions.

(b) P95 frame latency of sessions.

Figure 11: Encoder target bitrate and tail latency.

The gain arises from *MARC*'s queue-responsive optimization: a large frame inflates the queue length $b(i)$, increasing both its own send time (Eq. (3)) and the buffered data for subsequent frames (Eq. (5)). The larger send buffer lengthens the queueing delay of those later frames, which lowers the cumulative QoE in Eq. (8). To maximize QoE, *MARC* therefore lowers the bitrate of the upcoming frames. Because motion frames are typically large and arrive consecutively (see § 2.2), this mechanism implicitly prioritises motion phases even when $\lambda_m = 0$. As λ_m grows, *MARC* realizes better Pareto tradeoffs between quality and latency (i.e., moving closer to the lower-right corner in Fig. 10a and Fig. 10b), surpassing the baselines in both dimensions.

This indicates that *MARC*'s dynamic bitrate adjustment method utilizes bandwidth more effectively and reduces latency better than fixed-rate allocation methods or purely reactive methods. Compared to the Vidaptive method, which also adjusts the bitrate dynamically, *MARC* achieves lower motion frame latency. This is because *MARC* uses a motion predictor to allocate bitrate for motion frames ahead of time, whereas Vidaptive only reduces the bitrate upon detecting a latency increase. These results align with our analysis in § 2.4.

Improving performance across all sessions. To evaluate *MARC*'s overall performance, we collect the P95 queue+send latency and frame latency for each session and average these tail metrics across all sessions. Fig. 11(a) and (b) show the resulting tradeoffs between encoder target bitrate (x-axis) and tail latency (y-axis). By varying λ_s from 0.05 to 4, *MARC* traces out a Pareto frontier that clearly outperforms the baselines (Zoom, Duo, WebRTC, Vidaptive, etc.). At comparable

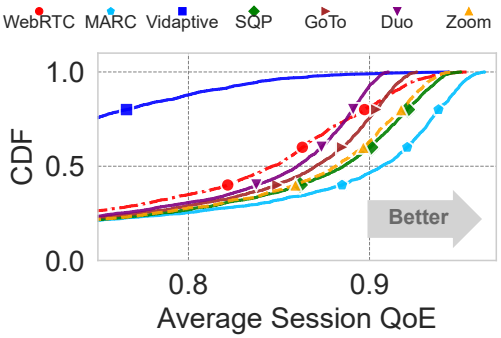


Figure 12: CDF of average session QoE across all sessions.

target bitrates, *MARC* achieves 22%–60% lower P95 frame latency than other algorithms. This performance gain arises primarily from λ_s, which governs all frames, whereas λ_m specifically targets motion frames (λ_m is fixed at 0.275 in this experiment).

When $\lambda_s = 0.05$, *MARC* achieves the best quality, while $\lambda_s = 4$ results in the lowest latency (Fig. 11b). Meanwhile, Vidaptive exhibits higher *P95* latency because its control logic focuses on keeping *P90* send duration below 33 ms; thus, the top 10% of frames can exceed the frame interval. Additionally, Vidaptive's frame-skipping mechanism was disabled to ensure a fair comparison. As a result, Vidaptive's more aggressive utilization of probed bandwidth, compared to other algorithms, leads to prolonged queueing delays and consequently the lowest overall QoE, as shown in Fig. 12. Overall, *MARC*'s approach provides a distinct advantage in both latency and video quality, validating our design hypothesis.

To further illustrate these QoE differences, Fig. 12 summarizes the overall QoE achieved by each algorithm. For every session, we first compute the per-frame QoE using the definitions in § 3.4, Eq. (10), and Eq. (11), and then average these values to obtain the average session QoE. The *MARC* curve corresponds to the default configuration of $\lambda_s = 1$ and $\lambda_m = 0.275$. As shown in Fig. 12, *MARC*'s curve is farthest to the right, indicating consistently higher QoE. Its median exceeds the baselines by 3%–36%, and its tail QoE at P95 is 1.3%–21% higher. The steeper rise of *MARC*'s CDF further suggests a smaller QoE variance across sessions.

5.3 *MARC* Deep Dive

The MPC algorithm's performance is influenced by *(i)* prediction accuracy [32, 62, 63] and *(ii)* QoE objectives [33, 63]. Therefore, we validate the impact of the state predictor and QoE functions on *MARC*'s performance.

Motion frame predictor. We first verify the accuracy of the motion frame predictor, followed by an analysis of its impact on the accuracy of the *MARC* algorithm.

To evaluate the accuracy of the predictor, we divided the dataset of user sessions collected online, allocating 80% of

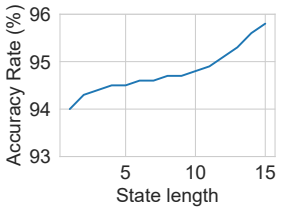


Figure 13: Accuracy rate of predicting the next frame's type.

the sessions for training and the remaining 20% for prediction. Within each session, frames were sequentially organized into samples, where each sample consisted of M historical frames followed by a single prediction frame.

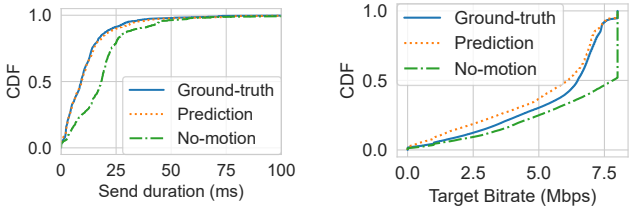
The accuracy of the motion frame predictor is affected by the state lengths (historical frame lengths). Fig. 13 illustrates the relationship between state length and prediction accuracy for the next frame type. As the state length increases, the accuracy of predicting the next frame type improves from 94% to 96%. The high accuracy is because most frames are non-motion frames, with motion frames constituting only 3.4% of all frames. However, the accuracy of predicting the start of a motion is relatively low, with a maximum of 36.8% and a recall of 36.7%. This is because the timing of the user's motions is random and therefore difficult to predict. Nonetheless, motion sequences tend to occur consecutively as observed in § 2.2, leading to a 72% improvement in the accuracy of predicting frames following an initial motion. This ensures the accuracy of *MARC*'s optimization for motion sequences.

To further evaluate the predictor's impact, we compared *MARC*'s performance under three scenarios: using ground-truth user motion sequences, predicted motion sequences, and without motion prediction (treating all frames as non-motion frames). Fig. 14a shows the CDF of send duration for these scenarios. The performance with predicted motion sequences closely matches that of ground-truth sequences, reflecting the high accuracy of our predictor. Without motion prediction, the median send duration doubles, emphasizing the importance of motion-aware rate control.

Fig. 14b compares the target bitrates for motion frames under these three strategies. Without motion frame prediction, the target bitrate is highest, increasing the likelihood of encoder overshooting and longer send durations. Using motion prediction results in a median target bitrate 5% lower than with ground truth, due to prediction errors.

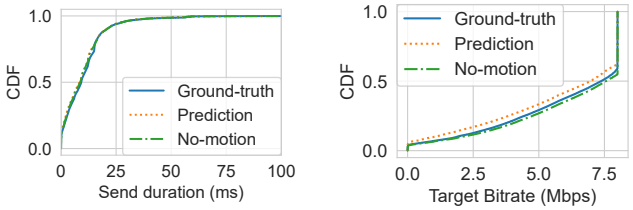
We also examined the impact on non-motion frames, as shown in Fig. 15. The send duration distribution remains consistent across all three strategies (Fig. 15a). However, the median target bitrate with motion prediction is 3% lower than with ground truth or no-motion strategies (Fig. 15b), as approximately 2% of non-motion frames are misclassified as motion frames, resulting in more conservative bitrate control.

Pacing rate and RTT predictor. We analyze the pacing rate and RTT for each frame in the motion sequence. Fig. 16a



(a) Send duration (b) Target bitrate

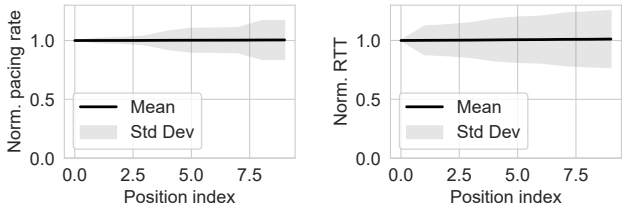
Figure 14: Performance of motion prediction model.



(a) Send duration (b) Target bitrate

Figure 15: Performance evaluation of the prediction model on non-motion frames.

illustrates the mean pacing rate for each frame position in motion sequences. The results show remarkable consistency, with the standard deviation remaining below 0.08 for the first four frames. This stability supports our strategy of using historical pacing rates to estimate future rates accurately. Similarly, in Fig. 16b, the mean RTT of consecutive frames remains stable with little variance, this consistency justifies our use of smoothed RTT values for future RTT estimation.



(a) Pacing rate (b) RTT

Figure 16: Stable pacing rate and RTT in motion phases.

5.4 Overhead Evaluation

Zero client-side overhead. *MARC* achieves efficiency with all functionalities implemented server-side, eliminating any impact on mobile device energy, storage, or computation, making it ready for large-scale deployment.

Minimal server-side overhead. We measure the inference time of two non-linear optimization solvers, SLSQP [13] and COBYLA [44], on our testbed server equipped with an Intel i9-13900KF CPU. In this experiment, the prediction window size was varied from 5 to 20 frames in increments of 5 frames.

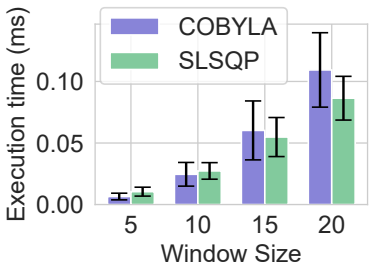


Figure 17: Per-frame execution time of two optimization solvers with different window sizes.

We varied the input transport rate from 0.1 to 8 Mbps in steps of 0.1 Mbps, and the send queue size from 0 to 150 KB in steps of 1.5 KB, as different inputs affect the execution time. Each input state was tested 1000 times, and the average execution time was recorded.

The results are presented in Fig. 17. COBYLA's execution time is lower than SLSQP's when the window size is 10 frames or fewer. However, the computation time of the COBYLA algorithm increases faster than SLSQP, resulting in longer execution times for COBYLA when the window size exceeds 15. During testing, we observed that COBYLA may reach its iteration limit without finding a solution for certain inputs when the prediction window is large. For these reasons, we choose to use SLSQP. Its average per-frame inference time is 24.6 µs, and the maximum time is 5.4 ms, both well within the frame interval (33 ms in our setting), with a prediction window size of 10. Furthermore, this results in only a 1.3% increase in absolute CPU utilization per session, making it suitable for multiple concurrent sessions in mobile cloud rendering scenarios. In practice, a rendering node already limits itself to about ten sessions per GPU, so server capacity remains GPU-bound; *MARC*'s extra CPU demand is well below the point where CPU becomes the scaling bottleneck.

5.5 Video Quality Evaluation

We assessed *MARC*'s impact on video quality by comparing original renderer output with received frames on a local testbed. The test uses 500-frame videos under various network traces. We employed three video quality metrics: PSNR, Structural Similarity Index (SSIM), and VMAF *phone model*. Tab. 2 shows that *MARC* has only a 1% decrease in PSNR (48.13 vs. 47.74) and VMAF (96.4 vs. 95.7) compared to WebRTC, while its average throughput demand differs by less than 3% from the baseline, indicating minimal perceived quality loss. *MARC* has a higher standard deviation in PSNR because *MARC* adjusts the bitrate for motion frames, resulting in minor quality fluctuations during the adjustment process.

Table 2: Video quality of *MARC* vs. WebRTC.

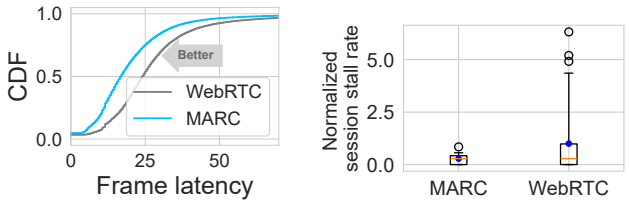
	PSNR	SSIM	VMAF
WebRTC	48.13(±0.82)	0.9937	96.4
MARC	47.74(±0.96)	0.9933	95.7

5.6 Large Scale Evaluation in the Wild

We deployed *MARC* in the online environment as described in § 5.1. *MARC* was compared with WebRTC in an A/B test. The test was conducted from April 4, 2024, to April 11, 2024, encompassing over 1 million sessions.

Metrics. Both latency and user engagement are evaluated. Latency measurements include the *send duration* of motion frames, the *proportion* of motion frames with an MTP latency exceeding 150ms, and the *session freeze rate*, which is the ratio of sessions that experience at least one freeze event. The freeze event count is provided by WebRTC [59]. User engagement is assessed using two behavioral interaction metrics [42]: *session duration*, which measures the total time each user spends on the service, and the *ratio* of interaction time to the session duration, which measures the user activity level.

Reducing latency. *MARC* reduces the frame latency by 20% at the tail (P99) and 29% at the median, as shown in Fig. 18a. Consequently, the proportion of motion frames with an MTP latency exceeding 150ms decreased by 20%. Additionally, the session freeze rate variance decreased, as illustrated in Fig. 18b, where the whiskers represent the 5th and 95th percentile values, and the average session freeze rate (denoted by a blue dot in the box) is reduced by 71%.



(a) Frame latency (b) Session freeze rate

Figure 18: Latency evaluation in online environment.

Improving user engagement. The latency reduction translates to improved user engagement. The session duration increased by 9%. The ratio of interaction time relative to the session duration increased by 20%, indicating that users were more willing to interact with the service under the *MARC* strategy. Such improvements in user engagement can potentially lead to increased profitability for e-commerce platforms [5].

Table 3: Comparison of user engagement metrics.

	Session Duration	Interaction Ratio
WebRTC	1	1
MARC	1.09	1.20

6 Discussion

Generalizability. *MARC*'s motion-aware bitrate adaptation extends beyond mobile e-commerce to interactive RTC workloads that alternate between active manipulation and passive viewing, such as cloud desktops, and remote VR/AR streaming. Porting *MARC* merely requires recalibrating the QoE mapping (Fig. 2, Eq. (7)) with engagement studies from the target application.

QoE function. In this study, we adopt a simplified QoE model that emphasizes two core elements of user experience: video quality and frame latency. Our results in Appendix B show that, even with this simplified model, the specific QoE formulation notably influences algorithm performance. Expanding beyond this formulation, a more sophisticated QoE model could integrate additional factors such as content complexity [47] and user expectations [19,31]. Metrics like bitrate switching could be explored under multi-dimensional QoE objectives to refine the framework further.

Controlling frame size. Our system uses existing encoder APIs to regulate encoding parameters, deliberately avoiding alterations to internal logic [17]. Certain frameworks offer the possibility of fine-grained frame size control by adjusting QP values [11] or through iterative re-encoding [12,40], *MARC* can extend its action space to incorporate these techniques. However, even with a perfect encoder that can encode frames at the exact target bitrate, users may still have different QoE requirements for motion and non-motion frames, which existing rate controllers cannot meet.

Paced sending and state predictors. Paced sending was employed to mitigate risks of queue buildup and network congestion when handling oversized frames [56]. Although pacing smooths network load, our simplified prediction of pacing rates may introduce latency estimation errors under extreme network bandwidth variations or prolonged queues. Several prediction modules, including pacing rate prediction, can still be improved by incorporating advanced time-series forecasting methods, such as those based on transformer [30]. We leave this part for future work.

Integrating other optimizations. Other techniques exist that target different components of the end-to-end pipeline in RTC systems, including loss recovery [36,51] and last-mile latency optimization [35]. While these techniques are promising to be combined with *MARC* to improve QoE performance, applying them in e-commerce cloud rendering systems may require further investigation.

QoE fairness. QoE fairness concerns dividing shared network resources so that users experience comparable satisfaction. Minerva [38] explores this for video-on-demand streams, a non-interactive setting where visual quality dominates and latency is largely secondary. In contrast, *MARC*'s measurements show that in interactive RTC workloads QoE priorities shift with the user's interaction phase § 2.2. This state-dependent insight provides a useful lever for future studies on fairness in multi-user interactive systems.

7 Related Work

QoE-related factors and metrics in video streaming. A substantial body of work has examined the impact of network conditions and video quality metrics on user engagement in streaming applications [10,27,53]. Building on these findings, *MARC* firstly correlates user engagement with user behavior and derives dynamic QoE preferences to guide rate control.

Rate control methods. Various rate control methods have been developed to optimize different aspects of Quality of Experience (QoE) in video streaming. For example, research has focused on dynamically adjusting bitrate based on factors such as dynamic network [52,64,65,68], content complexity [47], quality sensitivity [43,66], smoothness requirement [7] and visual attention [16,37,46,54]. However, these rate control methods typically rely on a fixed QoE model or predefined trade-offs. In contrast, *MARC* addresses intra-session QoE preference changes, continuously adjusting bitrate according to user behavior within the same session.

Congestion control for RTC. In RTC systems, congestion control algorithms (CCAs) aim to maximize bandwidth utilization and avoid network overloading [6,24,48,53]. CCAs typically probe bandwidth and adjust video sending rates based on delay, loss, or other congestion signals, but do not directly optimize for QoE objectives. Therefore, existing CCAs cannot solve the observed issue in § 2.

8 Conclusion

We have conducted a large-scale QoE study and identified that user behavior in mobile e-commerce cloud rendering exhibits unique QoE requirements, particularly in motion phases. Existing rate controllers often overlook the rapid QoE preference shifts in motion and non-motion phases, resulting in excessive latency exactly when users most value interactive responsiveness. To tackle this, we developed *MARC*, a dynamic QoE-driven motion-aware rate control framework that integrates user-behavior modeling and network dynamics, adjusting the bitrate at the frame level to optimize QoE. We implemented *MARC* and deployed it on Taobao's mobile cloud rendering platform. Millions of sessions demonstrated the efficiency of *MARC* in improving tail latency and user engagement.

Acknowledgments

We thank our shepherd, Jaehong Kim, and the anonymous reviewers for their insightful feedback. This work is supported by the National Key R&D Program of China (2022YFB2901800) and Alibaba Group through Alibaba Research Fellowship Program.

References

[1] Rfc 4960: Stream control transmission protocol. https://datatracker.ietf.org/doc/html/rfc4960, September 2007.

[2] NVIDIA NVENC Video Encoder API Programming Guide: Constant Bitrate (CBR) Mode. https://docs.nvidia.com/video-technologies/video-codec-sdk/11.1/nvenc-video-encoder-api-prog-guide/index.html#rate-control, 2024. Accessed: 2024-08-20.

[3] Venkat Arun and Hari Balakrishnan. Copa: Practical {Delay-Based} congestion control for the internet. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 329–342, 2018.

[4] Ali C. Begen and Jonathan Lennox. Rtp payload format for flexible forward error correction (fec). https://datatracker.ietf.org/doc/html/rfc8627, July 2019. Internet Engineering Task Force.

[5] Abdelsalam H Busalim, Fahad Ghabban, et al. Customer engagement behaviour on social commerce platforms: An empirical study. *Technology in society*, 64:101437, 2021.

[6] G. Carlucci, L. De Cicco, S. Holmer, and S. Mascolo. Analysis and design of the google congestion control for web real-time communication (webrtc). In *Proceedings of the 7th International Conference on Multimedia Systems*, pages 1–12, May 2016.

[7] Tianyu Chen, Yiheng Lin, Nicolas Christianson, Zahaib Akhtar, Sharath Dharmaji, Mohammad Hajiesmaili, Adam Wierman, and Ramesh K Sitaraman. Soda: An adaptive bitrate controller for consistent high-quality video streaming. In *Proceedings of the ACM SIGCOMM 2024 Conference*, pages 613–644, 2024.

[8] Chromium Project. WebRTC Video Encoder Bitrate Adjuster Source Code. https://source.chromium.org/chromium/chromium/src/+/main:third_party/webrtc/video/encoder_bitrate_adjuster.h, April 2024. Accessed: 2024-04-02.

[9] Mark Claypool and Kajal Claypool. Latency and player actions in online games. *Communications of the ACM*, 49(11):40–45, 2006.

[10] Florin Dobrian, Vyas Sekar, Asad Awan, Ion Stoica, Dilip Joseph, Aditya Ganjam, Jibin Zhan, and Hui Zhang. Understanding the impact of video quality on user engagement. *ACM SIGCOMM computer communication review*, 41(4):362–373, 2011.

[11] FFmpeg Developers. libavcodec. https://github.com/FFmpeg/FFmpeg/tree/master/libavcodec, 2024. Accessed: 2024-05-06.

[12] Sadjad Fouladi, John Emmons, Emre Orbay, Catherine Wu, Riad S Wahby, and Keith Winstein. Salsify:{Low-Latency} network video through tighter integration between a video codec and a transport protocol. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 267–282, 2018.

[13] Zhengqing Fu, Goulin Liu, and Lanlan Guo. Sequential quadratic programming method for nonlinear least squares estimation and its application. *Mathematical problems in engineering*, 2019(1):3087949, 2019.

[14] Epic Games. Unreal engine. https://www.unrealengine.com, 2019.

[15] Epic Games. Unreal engine pixelstreaming docs. https://docs.unrealengine.com/5.0/en-US/pixel-streaming-in-unreal-engine/, 2024. Accessed: 2024-02-03.

[16] Yu Guan, Chengyuan Zheng, Xinggong Zhang, Zongming Guo, and Junchen Jiang. Pano: Optimizing 360 video streaming with a better understanding of quality perception. In *Proceedings of the ACM Special Interest Group on Data Communication*, pages 394–407, 2019.

[17] Mohamed Hegazy, Khaled Diab, Mehdi Saeedi, Boris Ivanovic, Ihab Amer, Yang Liu, Gabor Sines, and Mohamed Hefeeda. Content-aware video encoding for cloud gaming. In *Proceedings of the 10th ACM multimedia systems conference*, pages 60–73, 2019.

[18] Chun-Ying Huang, Cheng-Hsin Hsu, Yu-Chun Chang, and Kuan-Ta Chen. Gaminganywhere: An open cloud gaming system. In *Proceedings of the 4th ACM multimedia systems conference*, pages 36–47, 2013.

[19] Tianchi Huang, Rui-Xiao Zhang, Chenglei Wu, and Lifeng Sun. Optimizing adaptive video streaming with human feedback. In *Proceedings of the 31st ACM International Conference on Multimedia*, pages 1707–1718, 2023.

[20] Tianchi Huang, Rui-Xiao Zhang, Chao Zhou, and Lifeng Sun. Qarc: Video quality aware rate control for real-time video streaming based on deep reinforcement learning. In *Proceedings of the 26th ACM international conference on Multimedia*, pages 1208–1216, 2018.

[21] IETF. Real-time communication in web-browsers (rtcweb). https://datatracker.ietf.org/wg/rtcweb/about/, 2023.

[22] IKEA. Ikea planners and design tools. https://www.ikea.com/us/en/planners/, 2024. Accessed: 2024-05-21.

[23] Zenja Ivkovic, Ian Stavness, Carl Gutwin, and Steven Sutcliffe. Quantifying and mitigating the negative effects of local latencies on aiming in 3d shooter games. In *Proceedings of the 33rd annual acm conference on human factors in computing systems*, pages 135–144, 2015.

[24] Zhidong Jia, Yihang Zhang, Qingyang Li, and Xinggong Zhang. Burstrtc: Harnessing variable bit-rate of rtc through frame-bursting congestion control. In *Proceedings of the 8th Asia-Pacific Workshop on Networking*, pages 213–214, 2024.

[25] Steven G. Johnson. NLopt: Nonlinear-optimization package. https://github.com/stevengj/nlopt.

[26] Pantea Karimi, Sadjad Fouladi, Vibhaalakshmi Sivaraman, and Mohammad Alizadeh. Vidaptive: Efficient and responsive rate control for real-time video on variable networks. *arXiv preprint arXiv:2309.16869*, 2023.

[27] S Shunmuga Krishnan and Ramesh K Sitaraman. Video stream quality impacts viewer behavior: inferring causality using quasi-experimental designs. In *Proceedings of the 2012 Internet Measurement Conference*, pages 211–224, 2012.

[28] Insoo Lee, Jinsung Lee, Kyunghan Lee, Dirk Grunwald, and Sangtae Ha. Demystifying commercial video conferencing applications. In *Proceedings of the 29th ACM international conference on multimedia*, pages 3583–3591, 2021.

[29] Zhi Li, Anne Aaron, Ioannis Katsavounidis, Anush Moorthy, Megha Manohara, et al. Toward a practical perceptual video quality metric. *The Netflix Tech Blog*, 6(2):2, 2016.

[30] Bryan Lim, Sercan Ö Arık, Nicolas Loeff, and Tomas Pfister. Temporal fusion transformers for interpretable multi-horizon time series forecasting. *International Journal of Forecasting*, 37(4):1748–1764, 2021.

[31] Shengmei Liu and Mark Claypool. The impact of latency on navigation in a first-person perspective game. In *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems*, pages 1–11, 2022.

[32] Gerui Lv, Qinghua Wu, Weiran Wang, Zhenyu Li, and Gaogang Xie. Lumos: towards better video streaming qoe through accurate throughput prediction. In *IEEE INFOCOM 2022 - IEEE Conference on Computer Communications*, pages 650–659, 2022.

[33] H. Mao, R. Netravali, and M. Alizadeh. Neural adaptive video streaming with pensieve. In *Proceedings of the conference of the ACM special interest group on data communication*, pages 197–210, August 2017.

[34] Z. Meng, T. Wang, Y. Shen, B. Wang, M. Xu, R. Han, et al. Enabling high quality Real-Time communications with adaptive Frame-Rate. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 1429–1450, 2023.

[35] Zili Meng, Yaning Guo, Chen Sun, Bo Wang, Justine Sherry, Hongqiang Harry Liu, and Mingwei Xu. Achieving consistent low latency for wireless real-time communications with the shortest control loop. In *Proceedings of the ACM SIGCOMM 2022 Conference*, SIGCOMM '22, page 193–206, New York, NY, USA, 2022. Association for Computing Machinery.

[36] Zili Meng, Xiao Kong, Jing Chen, Bo Wang, Mingwei Xu, Rui Han, Honghao Liu, Venkat Arun, Hongxin Hu, and Xue Wei. Hairpin: Rethinking packet loss recovery in edge-based interactive video streaming. In *USENIX NSDI*, 2024.

[37] Omar Mossad, Khaled Diab, Ihab Amer, and Mohamed Hefeeda. Deepgame: Efficient video encoding for cloud gaming. In *Proceedings of the 29th ACM International Conference on Multimedia*, pages 1387–1395, 2021.

[38] Vikram Nathan, Vibhaalakshmi Sivaraman, Ravichandra Addanki, Mehrdad Khani, Prateesh Goyal, and Mohammad Alizadeh. End-to-end transport for video qoe fairness. In *Proceedings of the ACM Special Interest Group on Data Communication*, pages 408–423. 2019.

[39] NVIDIA. Video codec sdk. https://developer.nvidia.com/video-codec-sdk, 2024. Accessed: 2024-05-21.

[40] NVIDIA Corporation. *NVIDIA NVENC Video Encoder API Programming Guide*, 2024. Accessed: 2024-05-21.

[41] Obsess. Obsess - virtual store platform. https://obsessar.com/, 2024. Accessed: 2024-05-21.

[42] Jeeyun Oh, Saraswathi Bellur, and S Shyam Sundar. Clicking, assessing, immersing, and sharing: An empirical model of user engagement with interactive media. *Communication Research*, 45(5):737–763, 2018.

[43] Mirko Palmer, Malte Appel, Kevin Spiteri, Balakrishnan Chandrasekaran, Anja Feldmann, and Ramesh K Sitaraman. Voxel: Cross-layer optimization for video streaming with imperfect transmission. In *Proceedings of the 17th International Conference on emerging Networking EXperiments and Technologies*, pages 359–374, 2021.

[44] Michael JD Powell. *A direct search optimization method that models the objective and constraint functions by linear interpolation*. Springer, 1994.

[45] WebRTC project. https://webrtc.org/, 2024. Accessed: 2024-02-03.

[46] Feng Qian, Bo Han, Qingyang Xiao, and Vijay Gopalakrishnan. Flare: Practical viewport-adaptive 360-degree video streaming for mobile devices. In *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking*, pages 99–114, 2018.

[47] Yanyuan Qin, Shuai Hao, K. R. Pattipati, Feng Qian, Subhabrata Sen, Bing Wang, and Chaoqun Yue. Abr streaming of vbr-encoded videos: characterization, challenges, and solutions. In *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '18, page 366–378, New York, NY, USA, 2018. Association for Computing Machinery.

[48] D. Ray, C. Smith, T. Wei, D. Chu, and S. Seshan. Sqp: Congestion control for low-latency interactive video streaming. arXiv preprint arXiv:2207.11857, 2022.

[49] Devdeep Ray, Jack Kosaian, KV Rashmi, and Srinivasan Seshan. Vantage: optimizing video upload for time-shifted viewing of social live streams. In *Proceedings of the ACM Special Interest Group on Data Communication*, pages 380–393. 2019.

[50] rooom. rooom - 3d solutions for e-commerce, 2024. Accessed: 2024-05-21.

[51] M. Rudow, F. Y. Yan, A. Kumar, G. Ananthanarayanan, M. Ellis, and K. V. Rashmi. Tambur: Efficient loss recovery for videoconferencing via streaming codes. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 953–971, 2023.

[52] Haiping Wang, Zhenhua Yu, Ruixiao Zhang, Siping Tao, Hebin Yu, and Shu Shi. Twinstar: A practical multi-path transmission framework for ultra-low latency video delivery. In *Proceedings of the 31st ACM International Conference on Multimedia*, pages 9234–9242, 2023.

[53] Shibo Wang, Shusen Yang, Xiao Kong, Chenglei Wu, Longwei Jiang, Chenren Xu, Cong Zhao, Xuesong Yang, Jianjun Xiao, Xin Liu, et al. Pudica: Toward {Near-Zero} queuing delay in congestion control for cloud gaming. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 113–129, 2024.

[54] Shibo Wang, Shusen Yang, Hailiang Li, Xiaodan Zhang, Chen Zhou, Chenren Xu, Feng Qian, Nanbin Wang, and Zongben Xu. Salientvr: Saliency-driven mobile 360-degree video streaming with gaze information. In *Proceedings of the 28th Annual International Conference on Mobile Computing And Networking*, pages 542–555, 2022.

[55] Zhou Wang, Alan C Bovik, Hamid R Sheikh, and Eero P Simoncelli. Image quality assessment: from error visibility to structural similarity. *IEEE transactions on image processing*, 13(4):600–612, 2004.

[56] WebRTC. Pacing module, 2024. Accessed: 2024-04-29.

[57] WebRTC Project. Peerconnection interface. https://source.chromium.org/chromium/chromium/src/+/main:third_party/webrtc/api/peer_connection_interface.h. Accessed: 2024-05-21.

[58] Wikipedia contributors. Webrtc - wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/WebRTC#Support, 2024. Accessed: 2024-02-02.

[59] World Wide Web Consortium. Webrtc statistics api: Freeze count definition. https://www.w3.org/TR/webrtc-stats/#dom-rtcinboundrtpstreamstats-freezecount, 2024. Accessed: 2024-08-13.

[60] M. Xu, Z. Fu, X. Ma, L. Zhang, Y. Li, F. Qian, et al. From cloud to edge: a first look at public edge platforms. In *Proceedings of the 21st ACM Internet Measurement Conference*, pages 37–53, November 2021.

[61] Xiaokun Xu and Mark Claypool. Measurement of cloud-based game streaming system response to competing tcp cubic or tcp bbr flows. In *Proceedings of the 22nd ACM Internet Measurement Conference*, pages 305–316, 2022.

[62] Francis Y. Yan, Hudson Ayers, Chenzhi Zhu, Sadjad Fouladi, James Hong, Keyi Zhang, Philip Levis, and Keith Winstein. Learning in situ: a randomized experiment in video streaming. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 495–511, Santa Clara, CA, February 2020. USENIX Association.

[63] Xiaoqi Yin, Abhishek Jindal, Vyas Sekar, and Bruno Sinopoli. A control-theoretic approach for dynamic adaptive video streaming over http. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, pages 325–338, 2015.

[64] H. Zhang, A. Zhou, Y. Hu, C. Li, G. Wang, X. Zhang, et al. Loki: improving long tail performance of learning-based real-time video adaptation by fusing rule-based models. In *Proceedings of the 27th Annual International Conference on Mobile Computing and Networking*, pages 775–788, October 2021.

[65] H. Zhang, A. Zhou, J. Lu, R. Ma, Y. Hu, C. Li, et al. Onrl: improving mobile video telephony via online reinforcement learning. In *Proceedings of the 26th Annual International Conference on Mobile Computing and Networking*, pages 1–14, September 2020.

[66] Xu Zhang, Yiyang Ou, Siddhartha Sen, and Junchen Jiang. SENSEI: Aligning video streaming quality with dynamic user sensitivity. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 303–320. USENIX Association, April 2021.

[67] Yuankang Zhao, Qinghua Wu, Gerui Lv, Furong Yang, Jiuhai Zhang, Feng Peng, Yanmei Liu, Zhenyu Li, Ying Chen, Hongyu Guo, et al. Jitbright: towards low-latency mobile cloud rendering through jitter buffer optimization. In *Proceedings of the 34th edition of the Workshop on Network and Operating System Support for Digital Audio and Video*, pages 36–42, 2024.

[68] A. Zhou, H. Zhang, G. Su, L. Wu, R. Ma, Z. Meng, et al. Learning to coordinate video codec with transport protocol for mobile video telephony. In *The 25th Annual International Conference on Mobile Computing and Networking*, pages 1–16, October 2019.

[69] Yuhan Zhou, Tingfeng Wang, Liying Wang, Nian Wen, Rui Han, Jing Wang, Chenglei Wu, Jiafeng Chen, Longwei Jiang, Shibo Wang, et al. Augur: Practical mobile multipath transport service for low tail latency in real-time streaming. In *USENIX NSDI*, 2024.

Appendix

A Finding the Best Prediction Horizon

In this section, we evaluate how different prediction horizon sizes (N) affect the performance of *MARC*. We vary N from 1 to 15 frames, measuring both the QoE gains (Fig. 19) and the additional computational overhead (Fig. 17). As shown in Fig. 19, using a horizon smaller than 10 frames can lead to suboptimal decisions that degrade tail latency (99.9 percentile), whereas increasing N beyond 10 frames yields only marginal QoE improvements at the cost of higher runtime overhead. Therefore, we select $N = 10$ as the default prediction horizon for *MARC*, striking a good balance between performance and complexity.

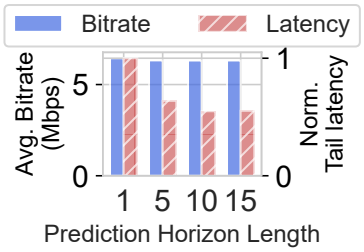
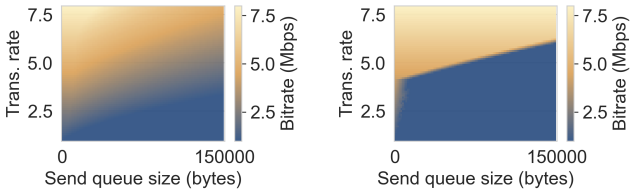


Figure 19: Impact of different prediction horizon sizes on overall QoE. Increasing N from 1 to 10 significantly reduces tail latency. Further increasing N yields diminishing returns.

B QoE Object Sensitivity

Sensitivity to QoE object. The QoE function defines the optimization objective and plays a crucial role in influencing *MARC*'s decisions. We investigate the impact of different QoE functions q_R on *MARC*'s bitrate selection by comparing the concave QoE function Eq. (10) with the linear QoE function Eq. (12). In our controlled experiments, we varied the send queue size and transport rate as follows: the transport rate ranged from 0.1 to 8 Mbps in increments of 0.1 Mbps, and the send queue size ranged from 0 to 150 KB in increments of 1500 bytes. The one-way delay was kept at 0, focusing solely on the impact of queuing delay on *MARC*'s decision-making. The delay QoE function q_L (Eq. (11)) remains unchanged with $\lambda_s = 0.1$, and motion frames are not considered.

$$q_R(R) = \frac{R}{R_{\max}} \tag{12}$$



(a) q_R (Eq. (10)). (b) Linear q_R (Eq. (12)).

Figure 20: The impact of different QoE object functions q_R on *MARC*'s bitrate decision (represented by color).

As shown in Fig. 20, both objective functions reduce the bitrate as the send queue increases. However, the algorithm using q_R makes more continuous bitrate decisions (Fig. 20a), while the algorithm using a linear q_R exhibits abrupt changes in bitrate (Fig. 20b). Such abrupt decision changes negatively impact the algorithm's overall performance. We evaluate the performance of these QoE objective functions on frame latency in our trace-driven evaluation environment. As shown in Fig. 21, the Pareto frontier of q_R outperforms that of Linear-q_R.

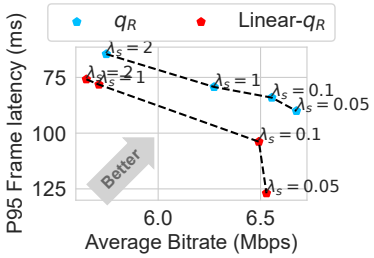


Figure 21: Comparison of the P95 frame latency with different q_R objective functions.

Accelerating Distributed Graph Learning by Using Collaborative In-Network Multicast and Aggregation

Zhaoyi Li[1,3], Jiawei Huang[1], Yijun Li[1], Jingling Liu[1], Junxue Zhang[2], Hui Li[1]
Xiaojun Zhu[1], Shengwen Zhou[1], Jing Shao[1], Xiaojuan Lu[1], Qichen Su[1]
Jianxin Wang[1], Chee Wei Tan[3], Yong Cui[4], Kai Chen[2]
[1]*Central South University,* [2]*Hong Kong University of Science and Technology*
[3]*Nanyang Technological University,* [4]*Tsinghua University*

Abstract

Distributed GNN training systems typically partition large graphs into multiple subgraphs and train them across multiple workers to eliminate single-GPU memory limitations. However, the graph propagation in each iteration involves numerous one-to-many multicast and many-to-one aggregation operations across workers, resulting in massive redundant traffic and severe bandwidth bottlenecks. Offloading multicast and aggregation operations into programmable switches has the potential to reduce the traffic volume significantly. Unfortunately, the complex dependencies among graph data and the limited switch-aggregator resources lead to performance degradation. The graph-agnostic sending order results in excessive traffic in multicast operations, leading to a severe backlog. Additionally, a small number of vertices may consume the major part of aggregator resources, while most traffic misses the opportunity for in-network aggregation.

To tackle these challenges, we propose SwitchGNN, which accelerates graph learning through coordinated in-network multicast and aggregation. First, to alleviate the link underutilization and queue backlog, we design a graph-aware multicast reordering algorithm, which prioritizes the upload of multicast vertices with the higher number of neighbors to reduce the communication time. Second, to prevent aggregator overflow, SwitchGNN employs a multi-level graph partitioning mechanism that further partitions boundary vertices into independent blocks to perform in-network aggregation in batches while ensuring the correctness of the graph propagation. We implement SwitchGNN using P4 programmable switch and DPDK host stack. The experimental results of the real testbed and NS3 simulations show that SwitchGNN effectively reduces the communication overhead and speeds up the training time by up to 74%.

1 Introduction

In recent years, Graph Neural Networks (GNNs) are widely applied to graph-related downstream tasks such as recommendation systems [1], drug discovery [2], public health surveillance [3, 4], and knowledge graphs [5] due to their ability to capture structured information from graph-based data. However, real-world graph data is becoming increasingly large. For instance, ByteDance observes over 2 billion vertices and 2 trillion edges with 100TB of graph data in production [6]. Training such massive graphs on a single worker or a single GPU is challenging because of memory limitations. To address this problem, a series of distributed GNN training frameworks [7–9] is proposed to partition the large graph into multiple subgraphs for parallel training on multiple GPUs.

Unfortunately, distributed GNN training generates massive communication traffic during the iterative graph propagation phase, which can account for up to 80% of the epoch time [10, 11] and become a training bottleneck. Specifically, the feature of each vertex is multicast to its remote neighbors located on different workers. Meanwhile, the worker receives features of the vertex's remote neighbors in a many-to-one fashion, aggregates them, and applies neural-network operations to compute the new vertex embeddings. Several solutions [6, 12–14] use sampling methods to reduce traffic volume. However, the absence of global graph information leads to low model quality. Existing full-graph training solutions alleviate communication overhead through efficient pipeline scheduling [15] or optimized graph partitioning algorithms [8]. However, these approaches rely on host-based multicast and aggregation, which still generate significant one-to-many redundant traffic and cause substantial many-to-one bandwidth bottlenecks, resulting in long epoch time.

With the emergence of programmable switches, a potential optimization for GNN training is to offload multicast and aggregation operations into the switches, thereby eliminating one-to-many and many-to-one bottlenecks, respectively. Specifically, each vertex is sent only once from the worker to the switch, and the switch then multicasts the vertex to the aggregators corresponding to its neighbors. Each aggregator, upon receiving all the required features, sends the aggregated result to the corresponding vertex. As a result, for each vertex, the worker receives only one single aggregated feature

from its neighbors. In the ideal case, in-network multicast and aggregation can significantly reduce the communication volume from $O(EM^2)$ to $O(EM)$, where E and M represent the number of boundary vertices and workers, respectively.

Existing in-network multicast and aggregation techniques are typically used in distributed DNN training [16–18] and word count [19], where the data is independent and can be multicast and aggregated in any order. However, graph-agnostic in-network multicast and aggregation are far from the ideal case. First, it is challenging to synchronize the multicast of these dependent vertices. Any straggling machine will cause aggregators to wait for long time until all necessary data arrives for aggregation. Second, the number of vertices with dependency is significantly large, creating an enormous demand for switch aggregator resources. For instance, the Reddit dataset with 32 workers needs approximately 500MB-sized aggregators for aggregation, while programmable switches typically have only around 10-100MB of memory [20, 21]. As a result, when a small portion of the graph data occupies all the switch's aggregator resources during multicast, the majority of the data must still be aggregated at the host, leading to substantial traffic.

To tackle these issues, we propose SwitchGNN, which leverages graph-aware in-network multicast and aggregation to reduce network traffic during graph propagation. First, to minimize communication time, we employ a graph-aware multicast reordering algorithm to determine the sending order of multicast vertices at the host. Vertices with a larger number of neighbors are prioritized for sending, ensuring a more efficient forwarding pipeline. Moreover, SwitchGNN addresses the limited aggregator resources by breaking a part of dependencies and partitioning boundary vertices from a large connected graph into multiple independent blocks. SwitchGNN aggregates vertices of each block in batches to ensure that switch-aggregator constraints are met.

In summary, the contributions of the paper are as follows:

- We identify that the one-to-many and many-to-one traffic generated during graph propagation in full-graph training is the key performance bottleneck. We reveal that though in-network multicast and aggregation techniques can eliminate this bottleneck, the complex dependencies between graph vertices still degrade performance.
- We propose SwitchGNN, a graph-aware in-network multicast and aggregation solution for GNN training. First, we introduce a graph-aware multicast reordering algorithm to reorder the sending sequence of multicast vertices, minimizing communication time. Then, it uses a multi-level graph partitioning algorithm to break the dependency within large connected boundary vertices and perform batch aggregation, preventing aggregator overflow and reducing traffic volume.
- We implement SwitchGNN using P4-programmable switches and DPDK-based host protocol stacks. Experimental results of testbed and NS3 simulations show that compared to the state-of-the-art GNN systems, we reduce epoch time by up to 74% without losing accuracy.

2 Background and Motivation

2.1 Distributed GNN

Graph Neural Networks are special neural networks designed to learn from graph-structured data. For example, Graph Convolutional Networks (GCNs) [22] extend convolution operations to graph domains. In each GCN layer, an aggregation operation (e.g., sum, mean, or max) collects features from each vertex's neighbors, followed by a neural network transformation that generates a new embedding for the vertex. This process is known as graph propagation. After multiple layers of graph propagation, the resulting embeddings, which encode structural information, are used for downstream tasks such as node classification and link prediction. Formally, a graph is defined as $G = (V, E, F)$, where V, E, and F represent the set of vertices, the set of edges, and the feature associated with each vertex, respectively. The graph propagation can be expressed as follows:

$$a_v^{(k)} = \text{AGGREGATE}^{(k)}\left(\left\{x_u^{(k-1)} \mid u \in N(v)\right\}\right), \quad (1)$$

$$x_v^{(k)} = \text{UPDATE}^{(k)}\left(a_v^{(k)}, x_v^{(k-1)}\right), \quad (2)$$

where $N(v)$ denotes neighbor vertices of vertex v, $x_u^{(k)}$ is the feature vector of vertex u at the k-th layer, $\text{AGGREGATE}^{(k)}$ is a function that aggregates features from neighbors of vertex v to produce the aggregated result $a_v^{(k)}$, and $\text{UPDATE}^{(k)}$ computes the new feature vector for vertex v.

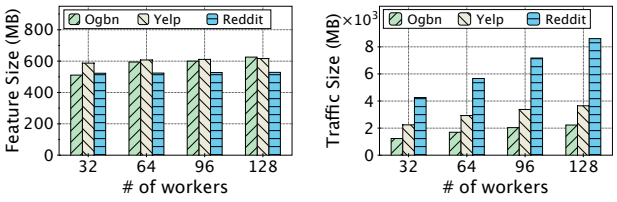
GNN training can be categorized into two approaches: mini-batch and full-graph training. Mini-batch training (e.g., DGL [7] and PyG [12]) samples a subset of vertices and their neighbors in each iteration, addressing the GPU memory limitation issue. However, this approach lacks global graph information, leading to unstable convergence and low accuracy. Therefore, we focus on full-graph training, which leverages the entire graph's features for training, achieving higher convergence accuracy [15].

In distributed GNN training, large graphs are partitioned into multiple subgraphs and distributed across different workers for parallel training. During each iteration's graph propagation phase, when a vertex's neighbors are located on remote workers, the features are transmitted over the network, resulting in significant communication overhead. Existing frameworks commonly employ the graph partitioning algorithm such as METIS [23] to balance the number of vertices in each

subgraph while minimizing the number of edges across subgraphs. This algorithm balances computational load across workers and reduces inter-worker communication volume.

2.2 All-to-all Communication

In distributed full-graph training, each epoch involves all-to-all communication between vertices and their remote neighbors. This can be viewed from two perspectives: sending and receiving. From the sending perspective, for the vertex v with d_v neighbors, the host-based multicast must copy its data d_v times and send it to d_v remote neighbors, causing large redundant traffic. From the receiving perspective, d_v features from d_v remote neighbors are pulled for host-based aggregation, leading to many-to-one bandwidth contention. The more remote neighbors the vertices have, the greater the all-to-all traffic and communication overhead.



(a) Feature Size (b) Traffic Size

Figure 1: Feature size and traffic size of different datasets.

We next analyze the boundary vertex size and traffic volume in realistic datasets. We use METIS [23] to partition the graph across varying numbers of workers. Three different datasets Ogbn-products [24], Reddit [25], and Yelp [26] are partitioned. Figure 1 shows that, while the boundary vertex size remains relatively stable as the number of workers increases, the traffic volume grows significantly. This is because, although the number of vertices per worker decreases, the average number of neighbors per vertex increases. Since the Reddit graph is denser than the others, it has more average neighbor partitions, resulting in the highest traffic volume. Specifically, in Reddit with 128 partitions, the traffic volume is 16 times the boundary size, meaning that, on average, the worker generates 1-to-16 copies of each vertex when sending and causes 16-to-1 bandwidth contention when receiving. Thus, host-based multicast and aggregation will induce significant one-to-many redundant traffic and many-to-one bandwidth contention.

2.3 Promise of In-network Multicast and Aggregation

2.3.1 Strength and limitation of programmable switch

Programmable switches, such as Intel Tofino, offer flexible packet processing capabilities, allowing users to define custom packet-handling logic. These switches utilize memory (stateless metadata and stateful registers) to store packet states and perform basic calculations. Many existing works leverage programmable switches to improve transmission performance. For instance, flexible in-network multicast mechanisms [27–29] enable stateless or hybrid multicast by parsing routing information carried in packet headers. In-network aggregation protocols [16–19, 30] offload key-value aggregation functions to switches to accelerate distributed machine learning and typical MapReduce jobs. Specifically, the switch memory is organized as an array of aggregators. The values with the same key from different workers are stored and aggregated in the aggregator with the same index. Once aggregation is complete, the results are sent back to the host via packets, and the aggregators are reallocated for other keys. These approaches significantly reduce traffic volume and network latency by using the programmability of switches. This provides an opportunity to leverage in-network multicast and aggregation to accelerate distributed GNN training.

However, programmable switches still face several limitations. Limited memory resources (e.g., 10MB to 100MB [20,21]) restrict the switch's ability to buffer and process large-scale traffic. The limited number of pipeline stages constrains the deployment of complex dependency logic. A packet can only access a stateful register once during a single pipeline pass. These constraints introduce challenges for offloading more advanced network functions to the switch.

2.3.2 Strawman solution

Indeed, offloading the multicast and aggregation operations into the switch for sending and receiving data can significantly reduce traffic volume. Specifically, each vertex's feature only needs to be sent once to the switch. The switch then multicasts this feature to the corresponding workers. For any vertex, all its neighbor features are aggregated on the switch, mitigating the many-to-one bottleneck. Assuming the graph is partitioned across M workers, with each worker having E boundary vertices on average, in the extreme case where boundary vertices are fully connected, the host-based multicast and aggregation will generate $2EM^2$ traffic, with EM^2 for sending and EM^2 for receiving. However, the in-network multicast and aggregation will only generate $2EM$ traffic, EM for sending, and EM for receiving, thus reducing the traffic volume to $\frac{1}{M}$ of the former one.

Thus, we design a strawman solution that leverages in-network multicast and aggregation to accelerate GNN training. First, each worker sends the features of boundary vertices to the switch using a best-effort approach. On the switch, each feature is copied d_v times into d_v aggregators corresponding to its d_v neighbors. For each aggregator, the received features are accumulated and then dropped until it receives all required features from the neighbors of the destination vertex. Once the aggregation is complete, the result is forwarded to the

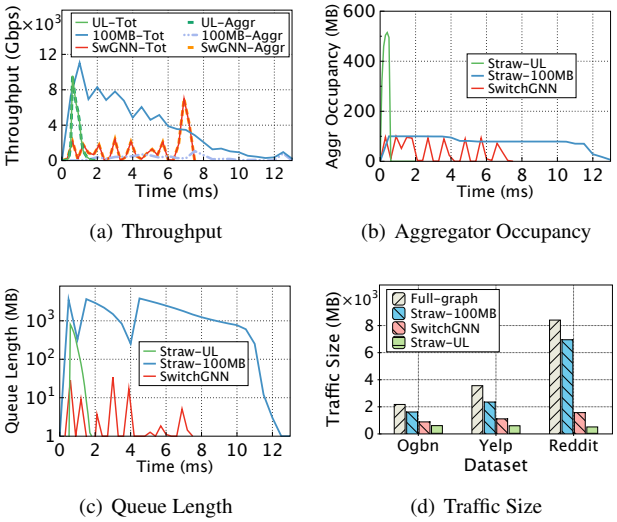
corresponding worker, and the aggregator is released for reuse by other vertices. If no aggregators are available when new data arrives at the switch, the current feature will be directly sent to the worker for aggregation.

2.4 Problem Insight

The strawman solution appears the significant potential to reduce traffic volume. However, directly applying in-network multicast and aggregation introduces two key problems: (1) The graph-agnostic multicast order results in link under-utilization and severe queue backlog. (2) The huge and inter-dependent vertex features are difficult to fit into the limited aggregators, leading to aggregator overflow and nullifying the traffic reduction.

2.4.1 Experimental observations

We conduct the NS3 simulations to reveal the problem of the strawman solution. We set up a star topology with 128 workers connected to one switch, and each link has 100Gbps bandwidth. We partition the Ogbn-product, Yelp, and Reddit datasets across 128 workers using the METIS [23] algorithm.



(a) Throughput (b) Aggregator Occupancy

(c) Queue Length (d) Traffic Size

Figure 2: The performance of strawman solution.

We evaluate the performance of the strawman solution with 100MB switch memory (aggregators) and with unlimited switch memory, respectively. We measure the real-time aggregation throughput, aggregator occupancy, and queue length under the Reddit dataset with 128 workers. To clearly observe the buffer requirements, we set the buffer size to 10GB to avoid the drastic performance fluctuations caused by packet loss and timeout retransmissions. As shown in Figure 2(a), although the total throughput under the 100MB constraint (100MB-Tot) is very high, the aggregation throughput (100MB-Aggr) is significantly lower than that of Straw-UL (UL-Aggr). During this process, as shown in Figure 2(b), the 100MB aggregator has been nearly fully occupied. Moreover, we observe that both the 100MB and unlimited cases experience significant queue backlog, as shown in Figure 2(c).

We further measure the traffic volume under the full-graph and strawman solutions. As shown in Figure 2(d), under the 100MB (Straw-100MB) aggregator constraint, the generated traffic volume of strawman is only slightly lower than that of the full-graph approach, but significantly higher than the unlimited case (Straw-UL). Specifically, for the Reddit dataset, the traffic volume in the Straw-UL is reduced by 94% and 92% compared to the full-graph and Straw-100MB, respectively.

2.4.2 Deep dive

Next, we dive into the causes of performance degradation in the strawman multicast and aggregation approach. Existing in-network multicast and aggregation techniques are commonly used to accelerate distributed machine learning jobs where data is independent. However, in GNNs, graph data dependencies are complex. The graph-structure-agnostic manner of the strawman solution leads to inefficient forwarding pipeline and severe aggregator overflow.

The graph-agnostic multicast order results in an inefficient forwarding pipeline. During the aggregation process, switch aggregators must wait for data from all participating vertices. Meanwhile, if a vertex completes multiple aggregations after being multicast to aggregators, multiple aggregation results are generated and sent to the workers. The strawman solution multicasts vertices to aggregators in random order. Due to the complex dependencies in graph data, some vertices are uploaded without timely completion of aggregation, leading to link under-utilization. Conversely, other vertices quickly complete a large number of aggregations after being uploaded, causing high queuing delays.

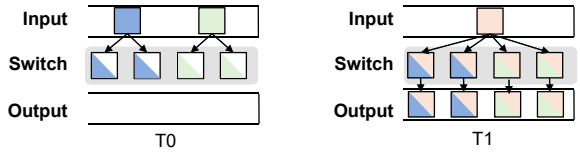


Figure 3: Graph-structure agnostic sending order leads to link under-utilization and queue backlog.

Figure 3 illustrates the link under-utilization and queue backlog due to graph-agnostic multicast order. At time T0, two vertices with a degree of 2 arrive to the switch and multicast to two aggregators, each waiting for aggregation to complete. During this time, the output throughput remains at 0. At time T1, although only one vertex is uploaded, it completes the aggregation for 4 aggregators simultaneously, however, causing a queue buildup at the output port. For instance, as shown in Figure 2(a) and Figure 2(c), under both Straw-UL and Straw-100MB, the queue backlog is severe. Specifically, from time 0ms to 1ms, the queue length rapidly increases from 0MB to 1000MB.

The huge and interdependent vertex features lead to aggregator overflow. A vertex's multicast on the switch consumes a number of aggregators equal to the number of its neighbors. However, the vertex required for these neighbors' aggregation also depends on their own neighbors. Due to the large number of dependent vertices (i.e., large connected subgraphs), multicasting these dependent vertices consumes far more aggregator resources than the programmable switch can support. As a result, a massive traffic is sent directly to the workers due to aggregator collisions, missing the in-network aggregation opportunities.

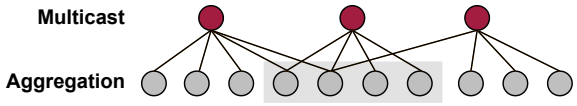


Figure 4: The memory demand of the large number of dependent vertices exceeds the switch memory size.

For example, as shown in Figure 4, when three dependent red vertices are simultaneously multicast at the switch, 10 aggregators are required, but the switch can only provide 4 aggregators, leaving the remaining 6 features unable to use in-network aggregation. The experimental results in Figure 2(b) show that when aggregator resources are unlimited, the aggregation resource usage is approximately 500MB. Therefore, under the 100MB resource constraint, severe aggregator overflow occurs. With a 100MB aggregator limit, the aggregation throughput is significantly reduced to only about 10% of the total throughput.

2.5 Summary

In distributed full-graph training, the graph propagation with host-based multicast and aggregation causes large redundant traffic and severe bandwidth contention, respectively. Using in-network multicast and aggregation can significantly reduce traffic volume, but the strawman solution suffers from queue backlog and aggregator overflow due to complex vertex dependencies. Therefore, we need to carefully design a collaborative in-network multicast and aggregation based on graph-structured data to fully leverage the potential of in-network multicast and aggregation.

3 SwitchGNN Design

3.1 Architecture and Workflow

We propose a solution to accelerate full-graph training using programmable switches. This solution performs collaborative in-network multicast and aggregation during graph propagation, thereby reducing network traffic volume.

Figure 5 shows SwitchGNN's architecture. We assume that the full graph has already been partitioned into multiple workers. First, the host reschedules the sending order of vertices to optimize the forwarding pipeline, alleviating the link underutilization and queue backlog. Second, to prevent aggregator overflow, we employ a multi-level graph partitioning mechanism to break the dependencies among boundary vertices. Moreover, we still transmit the traffic from cut edges to guarantee the correct graph propagation. The workers follow the predefined sending order to minimize communication time, while the switch stores the connectivity information between vertices for in-network multicast and aggregation. Third, we design the reliability and congestion control mechanisms to guarantee the correct aggregation and avoid network congestion in all-to-all transmission. Finally, the switch receives the feature of each vertex and multicast it to the aggregators corresponding to its neighbors. The aggregators aggregate the features from multiple workers and the aggregation results are sent back to the corresponding workers.

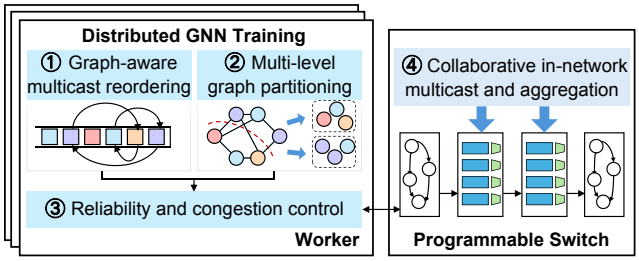


Figure 5: SwitchGNN's architecture.

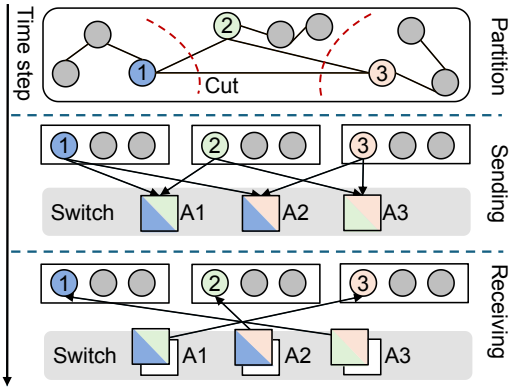


Figure 6: SwitchGNN's workflow.

We use an example to illustrate SwitchGNN workflow. As shown in Figure 6, we assume that vertices 1, 2, and 3 are interdependent and partitioned into three different workers. During the graph propagation, they need to exchange features over the network. In the feature sending step, vertices 1, 2, and 3 send their features to the switch, which multicasts them to the corresponding aggregators A{1,2}, A{1,3}, and A{2,3}, where aggregators A1, A2, and A3 correspond to destination vertices 3, 2, and 1, respectively. Once the aggregators have

received all the required neighbor features for each destination vertex, vertices 1, 2, and 3 receive the aggregated results from aggregators A3, A2, and A1. As a result, each of the three vertices sends only one feature and receives one aggregated feature. When all vertices have completed their transmissions, the communication for the current GNN layer is finished.

3.2 Graph-Aware Multicast Reordering

We reorder the multicast sequences of vertices according to the graph structure to minimize the communication time.

3.2.1 Problem statement

Given a graph $G(V,E)$ comprising n vertices, e edges, along with a designated vertex sequence $S = (v_1, v_2, ..., v_n)$ to be sent to a switch, assume that A_t denotes the set of vertices that arrive to the switch at time t. At most k packets can enter the switch and k packets leave the switch in each time slot. When $t \leq n/k$, the switch receives $v_{(t-1)k+1}$ to v_{tk}; otherwise, it receives $v_{(t-1)k+1}$ to v_n. Therefore, we have

$$A_t = \begin{cases} \bigcup_{i=(t-1)k+1}^{tk} \{v_i\}, & t \leq n/k, \\ \bigcup_{i=(t-1)k+1}^{n} \{v_i\}, & t > n/k. \end{cases} \tag{3}$$

We define $g_t(v_i)$ as the number of aggregation results completed when vertex v_i arrives at switch at time t. Let $f(t)$ denote the number of aggregators that complete the aggregation at time t, with the resulting packets queued at the switch. Then we have

$$f(t) = \sum_{v_i \in A_t} g_t(v_i). \tag{4}$$

Let $Q(t)$ represent the number of packets waiting for transmission after time t, and assume $Q(0) = 0$. Then we have

$$Q(t) = max\{f(t) + Q(t-1) - k, 0\}. \tag{5}$$

The job completion time z is given by

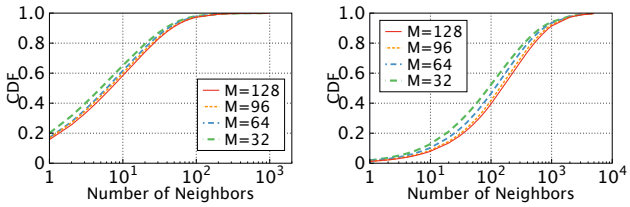
$$z = \lceil \frac{n}{k} \rceil + \lceil \frac{Q(\lceil \frac{n}{k} \rceil)}{k} \rceil, \tag{6}$$

where $\lceil \frac{n}{k} \rceil$represents the time required for all vertices to arrive at the switch, and $\lceil \frac{Q(\lceil \frac{n}{k} \rceil)}{k} \rceil$ represents the time required to drain the remaining queued packets at the switch after the last vertex has arrived. The objective is to find an optimal vertex sequence S to minimize z.

3.2.2 Lightweight solution

Considering the high complexity and substantial time cost in deriving the optimal solution, we propose a heuristic algorithm that maintains a manageable level of complexity for graph-aware multicast reordering.

First, we aim to minimize the aggregation time for features to avoid a large number of features waiting for aggregation at the switch, which leads to link under-utilization. When the vertices multicast into the aggregators, completing these aggregations quickly requires that other vertices destined for the same aggregators are also multicast immediately. These vertices are connected through a common one-hop neighbor, which links them together in the aggregation process. Thus, we use a priority-based breadth-first search (BFS) scheduling approach. Starting from a randomly selected vertex, its neighboring vertices are added to the sending queue, ensuring that closely related vertices are transmitted in close time. This reduces the waiting time of aggregators.



(a) Ogbn-products (b) Reddit

Figure 7: The CDF of neighbor counts.

Second, to alleviate the queue backlog, we introduce priority scheduling based on vertex degree. We observe that vertex degrees are skewed. For example, Figure 7 shows the cumulative distribution function (CDF) of neighbor counts. In the Reddit dataset, with 32 workers, 20% of vertices account for 80% of the total neighbors. If high-degree vertices are sent later, there is a higher probability of triggering a large number of aggregations at once, leading to traffic bursts. However, by sending higher-degree vertices earlier, as smaller-degree vertices are uploaded, aggregations are completed gently, enabling a more efficient forwarding pipeline and reducing the queue backlog. Thus, we enhance the BFS approach by assigning priority based on vertex degree, where higher-degree vertices are transmitted earlier in the sequence.

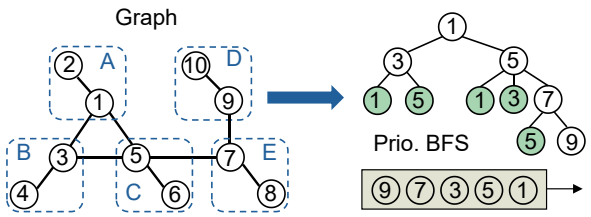


Figure 8: Priority-based BFS.

As illustrated in Figure 8, the vertex sequence is generated according to our priority-based BFS. We randomly select vertex 1 from the graph to enqueue. Vertex 1 is then dequeued, and its neighbor vertices 3 and 5 are enqueued. Since vertex 5

has more neighbors than vertex 3, vertex 5 will be dequeued first. Then, vertex 5's neighbor, vertex 7, is enqueued. As vertex 3 and vertex 7 have the same number of neighbors, the earlier enqueued vertex 3 will be dequeued before vertex 7. Vertex 9, with only one neighbor, will be dequeued last. Note that we track each enqueued vertex to ensure that each vertex is enqueued only once. The final vertex sequence is the order in which the vertices are dequeued. In non-in-network aggregation methods, high-degree vertices will generate more redundant traffic. However, in SwitchGNN, high-degree vertices uploaded to the switch do not immediately generate multicast traffic. Instead, they first occupy the aggregator's memory and trigger data transmission to the worker only after other required neighboring vertices arrive and complete aggregation. This multicast reordering strategy optimizes pipeline efficiency, mitigates queue backlog.

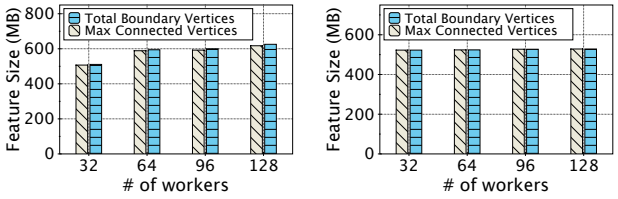
Next, to demonstrate the effectiveness of the graph-aware multicast reordering mechanism, we quantitatively analyze two typical graph structures of star graph (non-uniform) and ring graph (uniform), and discuss the performance difference between the priority-based BFS strategy (PB) and the random ordering (RO). In the star topology, consider the undirected graph $G_1(V,E)$, where vertex set is $V=\{v_1,v_2,...,v_n\}$, and edge set is $E=\{(v_1,v_2),(v_1,v_3),...,(v_1,v_n)\}$. The degree of the central vertex is $d(v_1)=n-1$, while all other vertices have degree 1, i.e., $d(v_2)=\cdots=d(v_n)=1$.. PB selects the vertex v_1 as the root of BFS, resulting in a traversal sequence $\{v_1,v_2,...v_n\}$. According to the derivation in 3.2.1, we assume that $k=1$, and the total execution time of PB is n. RO randomly selects vertices, and the total execution time of RO depends on the position of vertex v_1. The optimal case of RO is that v_1 is selected first, and the total execution time is n. In the worst case, the last choice is v_1, and the total execution time is $2n-1$; The average execution time is $\frac{3n-1}{2}$.

In the ring topology, we construct an undirected graph $G_2(V,E)$, where vertices is $V=\{v_1,v_2,...,v_n\}$, and edge set is $E=\{(v_1,v_2),(v_2,v_3),...,(v_{n-1},v_n),(v_n,v_1)\}$. All vertices have a degree of 2. When $n=3$, both strategies take the same execution time. When $n\geq 4$, the total execution time under PB is $n+2$, the best execution time under RO is $n+1$, and the worst execution time under RO is $n+2$.

The degree distribution characteristics of graph structures impact the performance of heuristic ordering strategies. SwitchGNN outperforms random ordering in graphs with non-uniform degree distributions, while both strategies show similar performance in graphs with uniform degree. Thus, SwitchGNN achieves greater benefits in skewed graphs, which is common in real-world scenarios [31].

3.3 Multi-level Graph Partitioning

The required number of aggregators is directly related to the maximum number of dependent vertices. Here, we analyze the size of related boundary vertices across different datasets.



(a) Ogbn-products (b) Reddit

Figure 9: The total boundary and max. connected feature size.

Figure 9 shows the total boundary and maximum connected vertices size under different datasets. The results show that nearly 99% of boundary vertices exhibit dependency relationships, thus the aggregator demand far exceeds the switch's memory capacity. To address this issue, in addition to partition the full graph across multiple workers, we further partition the connected boundary vertices into multiple blocks by cutting edges, thereby breaking their dependencies. These blocks are then transmitted one by one. The next block is processed only after the transmission of the previous one is complete. By ensuring that the number of vertices in each block does not exceed the available aggregator resources, we eliminate aggregator overflow. However, breaking the dependencies changes the original graph structure, leading to incorrect graph propagation and low model quality.

To maintain correct aggregation, data exchange is still required between cut edges. Thus, SwitchGNN groups the cut edges into a new block, and the partitioning continues until all blocks are small enough to fit within the aggregator limits. To fully utilize aggregator resources while minimizing the extra traffic caused by cut edges, we use the METIS algorithm for boundary vertices partitioning.

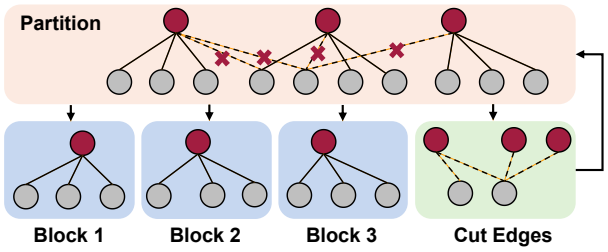


Figure 10: Multi-level graph partitioning.

Figure 10 shows an example of multi-level graph partitioning. We assume that a connected subgraph with 13 boundary vertices is to be multicast and aggregated on a switch with a limit of 4 aggregators. First, through graph partitioning, 4 edges are cut, resulting in 3 blocks, each with a size of 4 vertices. The subgraph formed by the cut edges has a size of 5, so it will continue to undergo the next level of partitioning until all blocks are smaller than 4. Then, by batch-processing the multicast and aggregation of vertices within each block, SwitchGNN completely avoids aggregator overflow. The time

complexity of METIS is $O(|E|+|V|)$, while the multicast reordering method has a time complexity of $O(|E|+|V|log|V|)$, where $|E|$ and $|V|$ denote the number of edges and vertices, respectively. For large graphs, Par-METIS method can be used to partition the graph in parallel. After multi-level partitioning is completed, vertices within each partitioned block can also be reordered in parallel, significantly reducing pre-processing time. Besides, SwitchGNN is still compatible with any advanced graph partitioning techniques to improve performance.

In the pre-processing phase, Graph-Aware Multicast Reordering and Multi-level Graph Partitioning mechanisms determine the multicast order and blocks of vertices, and this information is distributed to the corresponding hosts. Each host only needs to send its local vertices to the switch according to the predefined *block_id* and vertex order. Note that, each host must wait until the current block's transmission is completed before sending the next block. Once a host receives all the aggregation results in the current block, it sends the current *block_id* and its *host_id* to the switch. The switch keeps track of the current *block_id* and the number of hosts that have completed transmission for that block. When all hosts have completed the transmission of vertices for the current block, the switch multicasts the current *block_id* to all workers. Upon receiving the block completion signal from the switch, the workers send the vertices of next block.

3.4 Reliability and Congestion Control

In-network multicast and aggregation break the traditional end-to-end connections, where a single packet at the switch is replicated into multiple packets, and multiple packets are aggregated back into one packet by the switch. Therefore, we need to redesign the reliability guarantee and congestion control mechanisms to ensure correct and efficient transmission.

For reliability guarantee, when a vertex is sent, SwitchGNN maintains two bitmaps for each vertex to track the status of its neighbors at the host. One bitmap records the acknowledgment status of the sent vertex features, while the other tracks the reception status of each vertex's neighbor features. Upon receiving the aggregation result, the corresponding reception bitmap is set to all ones, and the worker multicasts ACKs to the vertex's neighbors. When the sending bitmap becomes all ones, it indicates that the vertex's transmission has successfully completed. When receiving a data packet of a vertex, the worker first determines whether the packet is a complete aggregation result from the switch. If a non-aggregated feature is received, the non-aggregated feature is cached in the worker. When other boundary vertices in the worker also need to participate in aggregation with this non-aggregated feature, they can share it from the local cache instead of fetching it again from remote workers, thereby reducing network traffic. Additionally, SwitchGNN employs a timeout mechanism to handle packet loss caused by network congestion. When a worker exceeds the timeout threshold without receiving features from the current block, it pulls the required features of the unaggregated vertices from other workers. Once the pulled packets arrive at the switch, the corresponding aggregators are released. The features pulled from the worker are marked as "bypass", meaning they no longer participate in switch-based aggregation but are directly forwarded to the host for aggregation.

For congestion control, SwitchGNN continuously sends vertices according to the scheduling order and adjusts the sending rate according to the Explicit Congestion Notification (ECN) marking, similar to DCQCN [32]. Upon receiving an ECN marking, the sending rate is halved. If the network is not congested, the sending rate increases additively.

4 Implementation

We implement SwitchGNN using Data Plane Development Kit (DPDK) protocol stack and P4-Programmable switch.

Host. On the host side, SwitchGNN is used as a plugin integrated with the DGL framework. We modify the communication context of DGL by using SwitchGNN with DPDK host stack. In the DPDK, we use $rte_pktmbuf_mtod_offset$ to add SwitchGNN's header fields after the IP header, which includes 32bit *Src_id*, 32bit *Dst_id*, 16bit *Block_id*, 16bit *Count*, 1bit *Is_ACK*, 1bit *Is_Fetch*, 1bit *ECN*, and 1bit *Resend* fields. The *Src_id* and *Dst_id* fields carry the IDs of source vertex and destination vertex, respectively. The *Block_id* field is used to notice the transmission batch, ensuring that the same block's features participate in in-network aggregation in the same batch. The *Count* field carries the number of boundary neighbors for the current vertices. The *Is_ACK* and *Is_Fetch* fields are used to identify the type of packet, indicating whether it is an acknowledgment or a fetch request, respectively. The *ECN* field is marked when the switch queue length exceeds a certain threshold, while the *Resend* field is set to 1 for retransmitted packets.

Switch. On the switch, we store the mapping relationship between each vertex ID and its corresponding aggregator indexes in the table. When a packet arrives, it can directly query the table using the source vertex ID to determine which aggregators should be aggregated at. In SwitchGNN, each aggregator is allocated 128 bytes for aggregation. The aggregator format is similar to ATP [16]. The difference is that instead of using a bitmap to track which vertices have been added in the aggregator, we determine aggregation completion according to the aggregation count. To avoid errors from duplicate accumulation due to retransmitted packets, we mark these packets with a *Resend* flag. When such a packet reaches the switch, it releases the corresponding aggregator and is directly forwarded to the designated worker for aggregation.

To multicast the packet into multiple aggregators, the same register needs to be accessed multiple times. However, Tofino switch allows a packet to access a register only once per pipeline pass. Therefore, after accumulating the payload into the first aggregator, we loop the packet back to the ingress

pipeline for subsequent aggregator operations until all multicast aggregators are accessed.

Moreover, SwitchGNN adopts in-network aggregation only for traffic that crosses hosts or racks, while GPUs within the same host use NVLink or PCIe for communication. When workers are distributed across multiple switches, multi-level aggregation is required. SwitchGNN uses a mechanism similar to existing in-network aggregation solutions [16, 18]. SwitchGNN needs to extend packet headers and aggregator fields to record the level of each switch. For example, in a two-level aggregation with three switches, two level-1 switches perform partial aggregation for directly connected workers, and their partial aggregation results are then forwarded to a level-2 switch for final aggregation.

5 Testbed Evaluation

5.1 Setup

Topology. We evaluate SwitchGNN using a star topology, which contains one switch and eight GPU servers. All servers connect to the switch via 100Gbps links. The switch is an Intel Wedge 100BF-32X programmable switch with 10MB memory. Each server has an RTX3090 GPU with CUDA 11.2, Ubuntu 20.04, 100GbE Mellanox CX5 NIC, 20 CPU cores and 64GB memory. We deploy one worker per server.

Datasets and models. We use three popular graph datasets in our evaluation, including Ogbn-products [24], Yelp [26], and Reddit [25]. The graph details are shown in Table 1. In addition, we train a 4-layer GCN with 128 hidden units and set the initial learning rate to 0.01 on the above datasets.

Table 1: Datasets Details.

Dataset	Vertices	Edges	Feature	Classes
Ogbn-products	2.4M	62M	100	47
Yelp	0.72M	7M	300	100
Reddit	0.23M	114M	602	41

Baselines. We compare SwitchGNN with state-of-the-art full-graph GNN training systems, such as BNS-GCN [8] and G3 [15]. BNS-GCN reduces communication overhead during graph propagation by sampling neighbors with a sampling rate p. To ensure full-graph training, we set p in BNS-GCN to 1. For BNS-GCN and SwitchGNN, we use METIS for the graph partitioning. G3 uses locality-aware iterative partition [15] to balance the communication load among workers. For a fair comparison, we implement the transport protocols of all baselines using DPDK. The transport protocols of BNS-GCN and G3 are implemented according to DCQCN.

Metrics. We adopt three evaluation metrics, including training throughput, real-time loss and GPU utilization in our experiments. The training throughput is the number of processed vertices per second of all workers. Real-time loss is the curve of training loss changing with time. We select one of the workers to observe the real-time GPU utilization.

5.2 Training Throughput

We evaluate the training throughput (measured in 10^4 vertices per second) of SwitchGNN under varying numbers of workers (e.g., 5∼8) with 4 GNN layers, and under different numbers of GNN layers using 8 workers, respectively. We compare SwitchGNN with baselines such as BNS-GCN and G3.

As shown in Figure 11 and 12, BNS-GCN achieves the lowest training throughput since it suffers from severe one-to-many and many-to-one bottlenecks. G3 outperforms BNS-GCN by balancing traffic volume among workers, avoiding the straggler problem caused by skewed traffic. However, G3 does not reduce the overall traffic volume, as there are still many redundant packets and significant many-to-one bandwidth contention in the network. SwitchGNN achieves the highest training throughput across all three datasets under different numbers of workers and GNN layers. This is because SwitchGNN eliminates the one-to-many and many-to-one bottlenecks by leveraging collaborative in-network multicast and aggregation, effectively reducing traffic volume. Moreover, as the number of workers increases, the benefits of SwitchGNN grow since the proportion of communication time relative to computation time increases. Specifically, SwitchGNN improves training throughput by up to 54% and 24% compared to BNS-GCN and G3, respectively.

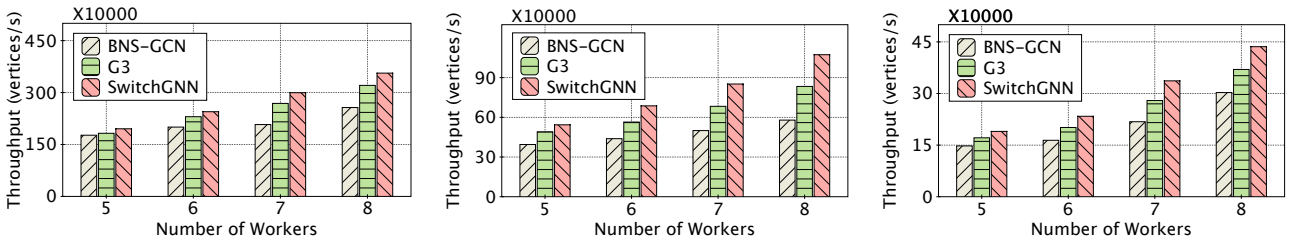
5.3 Real-time Loss

Next, we compare SwitchGNN with baselines in performance of time-to-accuracy. All training systems are trained in an 8-worker cluster using three datasets: Ogbn-products, Reddit and Yelp. We record the real-time loss during the training.

As shown in Figure 13, SwitchGNN achieves the fastest convergence across different datasets, without any loss in accuracy compared to BNS-GCN and G3. This is because, although SwitchGNN partitions boundary vertices into multiple blocks for in-network aggregation using a multi-level graph partitioning algorithm, it still transmits features associated with cut-edges, ensuring the correctness of graph propagation. The performance improvement of SwitchGNN is more significant on the denser datasets such as Yelp and Reddit than on the relatively sparse Ogbn-products dataset, as dense graphs generate more redundant traffic and experience more severe many-to-one bandwidth contention under BNS-GCN and G3.

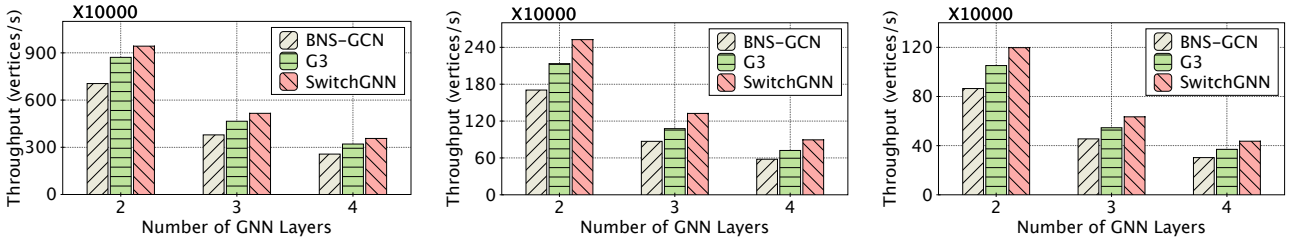
5.4 GPU Utilization

We evaluate the GPU utilization of one randomly selected worker from an 8-worker cluster. We use the Ogbn-products, Yelp and Reddit datasets to train the GCN model under SwitchGNN and our baselines. We show the GPU utilization over 10 seconds. Figure 14 shows the GPU utilization of SwitchGNN compared with full-graph training systems. We observe that SwitchGNN utilizes more GPU resources than BNS-GCN and G3. The reason is that SwitchGNN directly



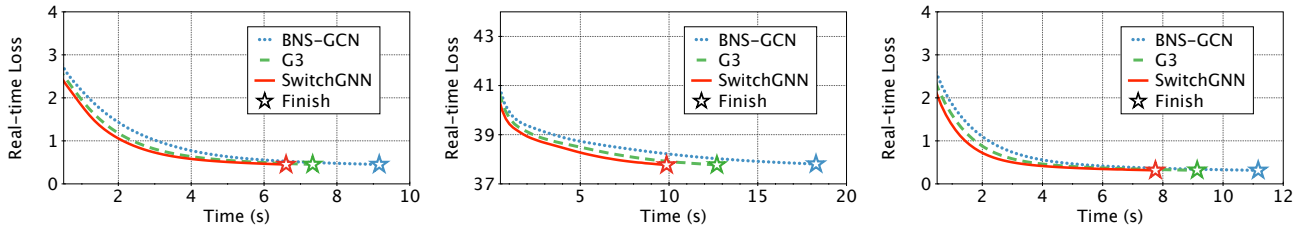
(a) Ogbn-products (b) Yelp (c) Reddit

Figure 11: Training throughput under varying number of workers.



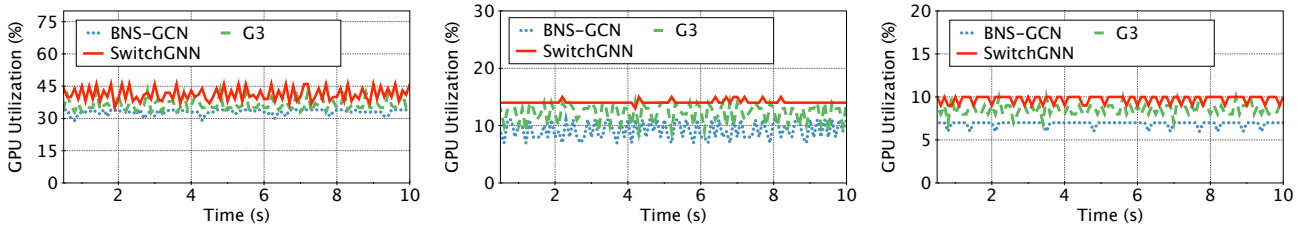
(a) Ogbn-products (b) Yelp (c) Reddit

Figure 12: Training throughput under varying number of GNN layers.



(a) Ogbn-products (b) Yelp (c) Reddit

Figure 13: Time to accuracy.



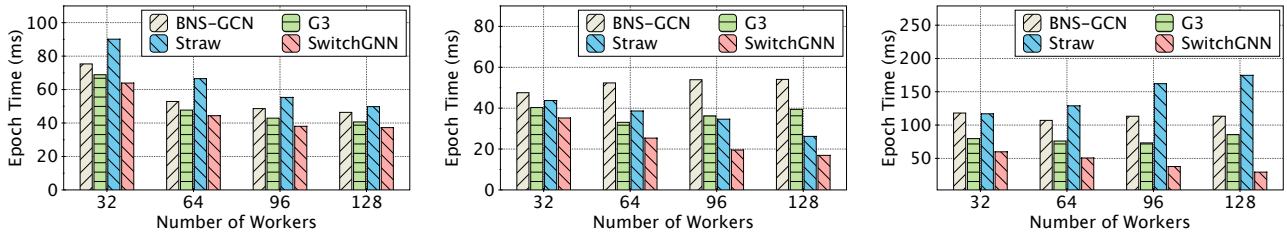
(a) Ogbn-products (b) Yelp (c) Reddit

Figure 14: Real-time GPU utilization.

reduces the traffic volume, reducing the communication time and contributing to higher GPU utilization.

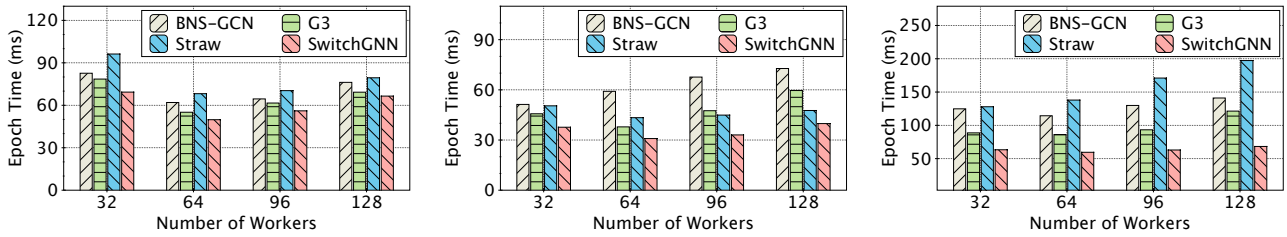
6 Simulation Evaluation

We conduct NS3 simulations to evaluate the performance of SwitchGNN in the large-scale scenarios. We use the star topology with 128 hosts and each host connects to the switch with 100Gbps bandwidth link and 2μs propagation delay. Based on the configuration of mainstream ASIC switches, the total aggregator size is set from 10MB to 100MB [20, 21]. By default, the switch memory size is 100MB. We compare the performance of BNS-GCN, G3, strawman solution (Straw), and SwtichGNN. We use three realistic graph workloads, including Ogbn-products, Yelp and Reddit. When scaling up the number of workers to 128, the dataset is re-divided into 128 partitions. The evaluation metrics include epoch time, traffic volume, queue length, and aggregation throughput. The epoch time consists of the communication time for a 4-layer GCN propagation in NS3 network combined with the average computation time per epoch during training in the testbed. Traffic volume refers to the data generated by one layer of GCN propagation.



(a) Ogbn-products (b) Yelp (c) Reddit

Figure 15: Epoch time under the varying number of workers with star topology.



(a) Ogbn-products (b) Yelp (c) Reddit

Figure 16: Epoch time under the varying number of workers with leaf-spine topology.

6.1 Basic Performance

First, we evaluate the basic performance of SwitchGNN using the same experimental setup as in the motivation section. As shown in Figure 2(d), across all three datasets, SwitchGNN significantly reduces traffic volume compared to the strawman approach. This reduction is achieved through multi-level graph partitioning, which effectively prevents aggregator overflow and thus substantially reduces traffic volume. However, to maintain correct aggregation after graph partitioning, SwitchGNN introduces some additional traffic, making its traffic size slightly larger than the UL case. Figures 2(a) and 2(b) demonstrate that SwitchGNN utilizes aggregator resources without overflow, maintaining high aggregation throughput. Moreover, due to the priority-based BFS algorithm that reorders the multicast sequence, SwitchGNN achieves a more efficient pipeline and avoids queue backlog, as illustrated in Figure 2(c).

Note that, under SwitchGNN, the synchronization process of each block between hosts and switches can be affected by straggling workers. As shown in Figures 2(a) and 2(b), the observed fluctuations in throughput and aggregator occupancy are caused by per-block synchronization. However, since SwitchGNN significantly reduces traffic volume, it still achieves low communication time.

6.2 Performance under Different Conditions

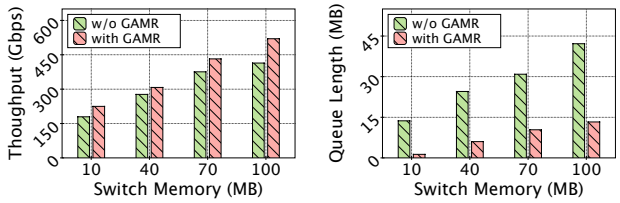
Then, we measure the epoch time of BNS-GCN, G3, Strawman solution, and SwitchGNN under varying numbers of workers and different workloads. As shown in Figure 15, across all three datasets, the epoch time of BNS-GCN, G3, and the strawman solution increases rapidly as the number of workers grows. This is because more workers result in each vertex needing to be replicated to more neighbors, introducing the one-to-many and many-to-one bottlenecks. Although G3 balances the communication load across workers, it does not reduce the overall traffic volume, still leading to high epoch time. In contrast, SwitchGNN leverages in-network multicast and aggregation to reduce traffic volume, effectively eliminating the one-to-many and many-to-one bottlenecks, and preventing significant traffic increases as the number of workers grows. Specifically, under Reddit dataset, SwitchGNN reduces epoch time by up to 74%, 65%, and 83% compared to BNS-GCN, G3, and strawman solution, respectively.

Besides, we conduct the experiments on an 8×8 leaf-spine topology in a typical datacenter, consisting of 8 leaf switches and 8 spine switches. Each leaf switch connects to 16 hosts, resulting in a total of 128 hosts. Each leaf switch connects the same number of workers, with at most one worker per host. For SwitchGNN, the in-network multicast and aggregation operations are deployed on the leaf switches. As a result, the traffic undergoes up to two levels of aggregation. The experimental results are shown in Figure 16. As the number of workers increases, cross-rack communication traffic also grows. Once cross-rack communication becomes the primary bottleneck, the epoch time increases rapidly. SwitchGNN reduces traffic in each rack through hierarchical aggregation, achieving the lowest epoch time.

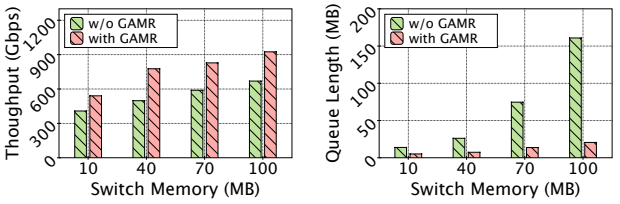
6.3 Effectiveness of GAMR

Moreover, we evaluate the effectiveness of the graph-aware multicast reordering (GAMR). We measure the maximum

queue length and throughput for Ogbn-products and Reddit datasets under varying switch memory from 10MB to 100MB. We set the switch buffer size to a very large value to observe the buffer requirements of different methods.



(a) Aggr. throughput under Ogbn (b) Queue length under Ogbn



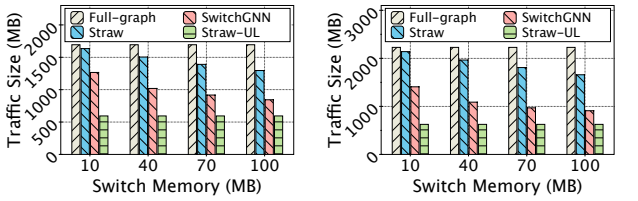
(c) Aggr. throughput under Reddit (d) Queue length under Reddit

Figure 17: Performance under different workloads.

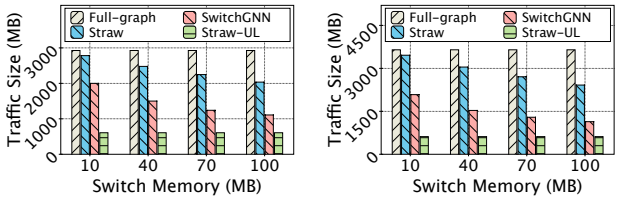
As shown in Figure 17, the results show that the performance of SwitchGNN with GAMR is significantly better than that without GAMR. GAMR enhances the pipeline efficiency for each block, improving the overall aggregation throughput. Moreover, as the switch memory increases, the maximum queue length under SwitchGNN without GAMR grows rapidly. This is because larger switch memory can hold more features waiting for aggregation, and the graph-agnostic sending order may cause a small number of packets to trigger larger aggregation results, leading to a huge traffic volume and severe queue backlog in a short time. Without GAMR, SwitchGNN experiences more severe queue backlog and throughput loss under the denser Reddit graph than that under Ogbn-products. This is because each vertex in Reddit has a higher degree, resulting in longer wait time for neighbors to arrive at the switch. By prioritizing high-degree boundary vertices for transmission, GAMR improves the throughput and alleviates the queue backlog.

6.4 Traffic Size under Varying Switch Memory

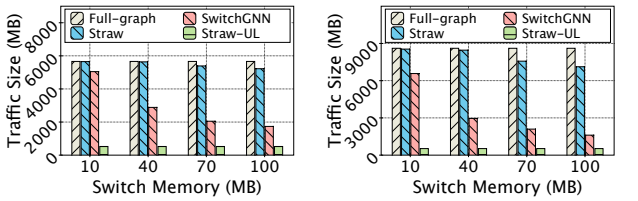
In this section, we evaluate the traffic size under different switch memory with 64 and 128 workers, respectively. As shown in Figure 18, the benefits of SwitchGNN become more significant as the number of workers increases. This is because a higher number of workers results in more copies of features being generated; SwitchGNN eliminates this redundant traffic through in-network multicast and aggregation. Moreover, Figure 18(f) shows that with only 10 MB memory, the traffic reduction is limited to 23%, whereas with 100 MB memory, the traffic size is reduced by 81%. This is because, under the smaller aggregator size, the graph is divided into more partitions during the multi-level partitioning process, thereby increasing traffic due to cut edges.



(a) Ogbn-products under 64 workers (b) Ogbn-products under 128 workers



(c) Yelp under 64 workers (d) Yelp under 128 workers



(e) Reddit under 64 workers (f) Reddit under 128 workers

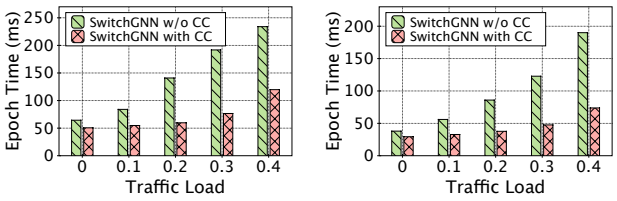
Figure 18: Traffic volume under varying switch memory.

As shown in Figures 18(b), 18(d) and 18(f), the dense graph (Reddit) generates more redundant traffic compared to the sparse graphs (Ogbn-products and Yelp). Thus, with larger memory size (100 MB), SwitchGNN delivers greater improvements for dense graphs. However, under limited memory (10MB), it is hard to partition dense graphs to eliminate inter-vertex dependencies, resulting in more cut edges. SwitchGNN shows better improvement for sparse graphs under limited memory.

6.5 Effectiveness of Congestion Control

Additionally, we validate the effectiveness of SwitchGNN's congestion control. We inject the background traffic under varying loads across 64 and 128 workers and observe the epoch time under the Reddit dataset with and without congestion control (CC). The background flows are randomly generated across all workers. The flow size follows the realistic datacenter workload of Hadoop and their arrival rate follows the Poisson distribution.

As shown in Figure 19, as the background traffic load increases, the SwitchGNN without CC suffers from significant



(a) 64 workers (b) 128 workers

Figure 19: Epoch time under varying traffic load.

performance degradation, whereas the SwitchGNN with CC can maintain a short epoch time.

7 Related Work

Mini-batch GNN training. DGL [7] distills several generalized sparse tensors from the GNNs' computation for massive parallelization. AGL [33] adopts a message transform method and is implemented on MapReduce [34] to support mini-batch GNN training. BGL [6] co-designs the caching policy and sampling order, reducing the communication overhead in feature retrieving. PaGraph [35] utilizes spare GPU memory as a cache for storing hot data to reduce the communication between CPU and GPU. GNNLab [36] proposes a pre-sampling cache policy for efficient storage and robust performance. P3 [37] combines model and data parallelism to reduce the communication overhead. However, the above approaches with mini-batch training mechanisms still face challenges such as sampling variance, neighbor explosion, and significant sampling overhead.

Full-graph GNN training. Full-graph training methods can overcome the above problems of mini-batch training methods. However, these methods suffer from large communication overhead. In recent years, numerous works have been proposed to reduce the communication overhead. Neugraph [38] combines dataflow-based deep learning frameworks to accelerate graph training through parallel computation. ROC [39] designs an online-learning-based graph partition algorithm to achieve a balanced workload and employs dynamic programming to optimize memory management. DGCL [40] proposes a shortest path spanning tree algorithm to schedule traffic, fully utilizing NVLink, PCIe, and other link bandwidths while balancing workloads and avoiding contention. BNS-GCN [8] randomly chooses the neighbors of boundary vertices to decrease the communication overhead. PipeGCN [41] achieves an efficient pipeline to overlap communication and computation while employing a smoothing algorithm to reduce errors incurred by stale features. Sancus [11] reduces communication overhead by adaptively caching historical embeddings. G3 [15] further balances the communication load among multiple workers while introducing finer-grained pipelining through partitioned vertices into smaller bins. This approach overlaps the communication and computation time. To address the high CPU-GPU communication overhead caused by duplicated traffic fetching during large-scale graph training, HongTu [9] employs a deduplication communication framework, which transfers the redundant CPU-GPU traffic on high-bandwidth inter-GPU links, accelerating the training.

Unfortunately, existing solutions still face the challenge of massive traffic volume. Our design introduces in-network multicast and aggregation for GNN training, significantly reducing traffic without sacrificing training accuracy.

In-network aggregation. DAIET [42] is the first INA algorithm and aggregates gradients at the switch to reduce network traffic. Sharp [43] and Sharpv2 [44] reduce the traffic by aggregating it in an aggregation tree deployed within the switches. However, Sharp does not consider the packet loss, which makes it unsuitable for lossy networks. SwitchML [17] proposes a packet loss recovery mechanism to aggregate gradients while providing a static and fair aggregator allocation mechanism. PANAMA [30] supports floating-point value for INA by using FPGA. Nevertheless, the static resource allocation of SwitchML and PANAMA leads to aggregator underutilization. ATP [16] addresses this problem with dynamic hash allocation. To further improve aggregation efficiency, A^2TP [18] decouples the control of link bandwidth and in-switch aggregators, and independently allocates the resources according to the job and network status.

However, none of the existing INA solutions are designed for GNN. SwitchGNN coordinates in-network multicast and aggregation according to the graph-structure relationships, achieving more efficient transmission.

8 Conclusion

In this paper, we propose an in-network accelerated framework SwitchGNN for distributed GNN training. SwitchGNN coordinates the in-network multicast and aggregation to reduce the traffic volume. First, to minimize the communication time and alleviate queue backlog, SwitchGNN uses graph-aware multicast reordering to schedule the multicast sequence, achieving an efficient pipeline. Then, SwitchGNN employs the multi-level partitioning method to avoid aggregator overflow. The experiment results show that SwitchGNN reduces the epoch time by up to 74% compared to the state-of-the-art full-graph training systems.

Acknowledgments

We thank our shepherd Rajan M. A. and the anonymous reviewers for their helpful comments. This work was supported by the National Natural Science Foundation of China (U24A20245, 62132022, 62402407, 62132009, 62302524), the Science and Technology Innovation Program of Hunan Province (2024RC1005, 2024JJ6531), the Singapore Ministry of Education Tier 2 AcRF (MOE-T2EP20224-0009), and the China Scholarship Council Scholarship.

References

[1] Rex Ying, Ruining He, Kaifeng Chen, Pong Eksombatchai, William L Hamilton, and Jure Leskovec. Graph convolutional neural networks for web-scale recommender systems. In *Proc. ACM SIGKDD*, pages 974–983, 2018.

[2] Shuangli Li, Jingbo Zhou, Tong Xu, Liang Huang, Fan Wang, Haoyi Xiong, Weili Huang, Dejing Dou, and Hui Xiong. Structure-aware interactive graph neural networks for the prediction of protein-ligand binding affinity. In *Proc. ACM SIGKDD*, pages 975–985, 2021.

[3] Ching Nam Hang, Pei-Duo Yu, Siya Chen, Chee Wei Tan, and Guanrong Chen. Mega: machine learning-enhanced graph analytics for infodemic risk management. *IEEE Journal of Biomedical and Health Informatics*, 27(12):6100–6111, 2023.

[4] Chee Wei Tan, Pei-Duo Yu, Siya Chen, and H Vincent Poor. Deeptrace: Learning to optimize contact tracing in epidemic networks with graph neural networks. *IEEE Transactions on Signal and Information Processing over Networks*, 11:97–113, 2025.

[5] Takuo Hamaguchi, Hidekazu Oiwa, Masashi Shimbo, and Yuji Matsumoto. Knowledge transfer for out-of-knowledge-base entities: a graph neural network approach. In *Proc. IJCAI*, pages 1802–1808, 2017.

[6] Tianfeng Liu, Yangrui Chen, Dan Li, Chuan Wu, Yibo Zhu, Jun He, Yanghua Peng, Hongzheng Chen, Hongzhi Chen, and Chuanxiong Guo. Bgl:gpu-efficient GNN training by optimizing graph data i/o and preprocessing. In *Proc. USENIX NSDI*, pages 103–118, 2023.

[7] Minjie Wang, Lingfan Yu, Da Zheng, Quan Gan, Gai Yu, Zihao Ye, Mufei Li, Jinjing Zhou, Qi Huang, Chao Ma, Ziyue Huang, Qipeng Guo, Hao Zhang, Haibin Lin, Junbo Zhao, Jinyang Li, Alexander Smola, and Zheng Zhang. Deep graph library: Towards efficient and scalable deep learning on graphs. In *Proc. ICLR*, 2019.

[8] Cheng Wan, Youjie Li, Ang Li, Nam Sung Kim, and Yingyan Lin. Bns-gcn: Efficient full-graph training of graph convolutional networks with partition-parallelism and random boundary node sampling. In *Proc. MLsys*, volume 4, pages 673–693, 2022.

[9] Qiange Wang, Yao Chen, Weng-Fai Wong, and Bingsheng He. Hongtu: Scalable full-graph GNN training on multiple gpus. In *Proc. ACM SIGMOD*, pages 1–27, 2023.

[10] Zhe Zhang, Ziyue Luo, and Chuan Wu. Two-level graph caching for expediting distributed GNN training. In *Proc. IEEE INFOCOM*, pages 1–10. IEEE, 2023.

[11] Jingshu Peng, Zhao Chen, Yingxia Shao, Yanyan Shen, Lei Chen, and Jiannong Cao. Sancus: staleness-aware communication-avoiding full-graph decentralized training in large-scale graph neural networks. In *Proc. VLDB*, pages 1937–1950. VLDB Endowment, 2022.

[12] Matthias Fey and Jan Eric Lenssen. Fast graph representation learning with pytorch geometric. *arXiv preprint arXiv:1903.02428*, 2019.

[13] Swapnil Gandhi and Anand Padmanabha Iyer. P3: Distributed deep graph learning at scale. In *Proc. USENIX OSID*, pages 551–568, 2021.

[14] Tim Kaler, Nickolas Stathas, Anne Ouyang, Alexandros-Stavros Iliopoulos, Tao Schardl, Charles E Leiserson, and Jie Chen. Accelerating training and inference of graph neural networks with fast sampling and pipelining. In *Proc. MLSys*, pages 172–189, 2022.

[15] Xinchen Wan, Kaiqiang Xu, Xudong Liao, Yilun Jin, Kai Chen, and Xin Jin. Scalable and efficient full-graph GNN training for large graphs. In *Proc. ACM SIGMOD*, pages 1–23, 2023.

[16] ChonLam Lao, Yanfang Le, Kshiteej Mahajan, Yixi Chen, Wenfei Wu, Aditya Akella, and Michael Swift. Atp: In-network aggregation for multi-tenant learning. In *Proc. USENIX NSDI*, pages 741–761, 2021.

[17] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan Ports, and Peter Richtárik. Scaling distributed machine learning with in-network aggregation. In *Proc. USENIX NSDI*, pages 785–808, 2021.

[18] Zhaoyi Li, Jiawei Huang, Yijun Li, Aikun Xu, Shengwen Zhou, Jingling Liu, and Jianxin Wang. A2tp: Aggregator-aware in-network aggregation for multi-tenant learning. In *Proc. ACM Eurosys*, pages 639–653, 2023.

[19] Yongchao He, Wenfei Wu, Yanfang Le, Ming Liu, and ChonLam Lao. A generic service to provide in-network aggregation for key-value streams. In *Proc. ACM ASPLOS*, pages 33–47, 2023.

[20] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. Silkroad: Making stateful layer-4 load balancing fast and cheap using switching ASICs. In *Proc. ACM SIGCOMM*, pages 15–28, 2017.

[21] Mariano Scazzariello, Tommaso Caiazzi, Hamid Ghasemirahni, Tom Barbette, Dejan Kostić, and Marco Chiesa. A high-speed stateful packet processing approach for tbps programmable switches. In *Proc. USENIX NSDI*, pages 1237–1255, 2023.

[22] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. In *Proc. ICLR*, 2017.

[23] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on scientific Computing*, 20(1):359–392, 1998.

[24] Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. Open graph benchmark: Datasets for machine learning on graphs. In *Proc. NIPS*, pages 22118–22133, 2020.

[25] Will Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. In *Proc. NIPS*, volume 30, 2017.

[26] Hanqing Zeng, Hongkuan Zhou, Ajitesh Srivastava, Rajgopal Kannan, and Viktor Prasanna. Graphsaint: Graph sampling based inductive learning method. *arXiv preprint arXiv:1907.04931*, 2019.

[27] Khaled Diab and Mohamed Hefeeda. Yeti: Stateless and generalized multicast forwarding. In *Proc. USENIX NSDI*, pages 1093–1114, 2022.

[28] Khaled Diab, Parham Yassini, and Mohamed Hefeeda. Orca: Server-assisted multicast for datacenter networks. In *Proc. USENIX NSDI*, pages 1075–1091, 2022.

[29] Muhammad Shahbaz, Lalith Suresh, Jennifer Rexford, Nick Feamster, Ori Rottenstreich, and Mukesh Hira. Elmo: Source routed multicast for public clouds. In *Proc. ACM SIGCOMM*, pages 458–471, 2019.

[30] Nadeen Gebara, Manya Ghobadi, and Paolo Costa. In-network aggregation for shared machine learning clusters. In *Proc. MLSys*, pages 829–844, 2021.

[31] Rong Chen, Jiaxin Shi, Yanzhe Chen, Binyu Zang, Haibing Guan, and Haibo Chen. Powerlyra: Differentiated graph computation and partitioning on skewed graphs. *ACM Transactions on Parallel Computing*, 5(3):1–39, 2019.

[32] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. Congestion control for large-scale rdma deployments. In *Proc. ACM SIGCOMM*, pages 523–536, 2015.

[33] Dalong Zhang, Xin Huang, Ziqi Liu, Jun Zhou, Zhiyang Hu, Xianzheng Song, Zhibang Ge, Lin Wang, Zhiqiang Zhang, and Yuan Qi. Agl: A scalable system for industrial-purpose graph machine learning. *VLDB Endow.*, 13(12), 2020.

[34] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[35] Zhiqi Lin, Cheng Li, Youshan Miao, Yunxin Liu, and Yinlong Xu. Pagraph: Scaling GNN training on large graphs via computation-aware caching. In *Proc. ACM SOCC*, pages 401–415, 2020.

[36] Jianbang Yang, Dahai Tang, Xiaoniu Song, Lei Wang, Qiang Yin, Rong Chen, Wenyuan Yu, and Jingren Zhou. Gnnlab: a factored system for sample-based GNN training over gpus. In *Proc. ACM Eurosys*, pages 417–434, 2022.

[37] Swapnil Gandhi and Anand Padmanabha Iyer. P3: Distributed deep graph learning at scale. In *Proc. USENIX OSDI*, pages 551–568, 2021.

[38] Lingxiao Ma, Zhi Yang, Youshan Miao, Jilong Xue, Ming Wu, Lidong Zhou, and Yafei Dai. Neugraph: Parallel deep neural network computation on large graphs. In *Proc. USENIX ATC*, pages 443–458, 2019.

[39] Zhihao Jia, Sina Lin, Mingyu Gao, Matei Zaharia, and Alex Aiken. Improving the accuracy, scalability, and performance of graph neural networks with roc. In *Proc. MLSys*, pages 187–198, 2020.

[40] Zhenkun Cai, Xiao Yan, Yidi Wu, Kaihao Ma, James Cheng, and Fan Yu. Dgcl: An efficient communication library for distributed GNN training. In *Proc. ACM Eurosys*, pages 130–144, 2021.

[41] Cheng Wan, Youjie Li, Cameron R Wolfe, Anastasios Kyrillidis, Nam Sung Kim, and Yingyan Lin. Pipegcn: Efficient full-graph training of graph convolutional networks with pipelined feature communication. *arXiv preprint arXiv:2203.10428*, 2022.

[42] Amedeo Sapio, Ibrahim Abdelaziz, Abdulla Aldilaijan, Marco Canini, and Panos Kalnis. In-network computation is a dumb idea whose time has come. In *Proc. ACM HotNets*, pages 150–156, 2017.

[43] Richard L Graham, Devendar Bureddy, Pak Lui, Hal Rosenstock, Gilad Shainer, Gil Bloch, Dror Goldenerg, Mike Dubman, Sasha Kotchubievsky, Vladimir Koushnir, et al. Scalable hierarchical aggregation protocol (sharp): A hardware architecture for efficient data reduction. In *Proc. IEEE COMHPC*, pages 1–10. IEEE, 2016.

[44] Richard L Graham, Lion Levi, Devendar Burredy, Gil Bloch, Gilad Shainer, David Cho, George Elias, Daniel Klein, Joshua Ladd, Ophir Maor, et al. Scalable hierarchical aggregation and reduction protocol (sharp) tm streaming-aggregation hardware design and evaluation. In *Proc. HiPC*, pages 41–59. Springer, 2020.

Opening Up Kernel-Bypass TCP Stacks

Shinichi Awamoto and Michio Honda
University of Edinburgh

Abstract

We have seen a surge of kernel-bypass network stacks with different design decisions for higher throughput and lower latency than the kernel stack, but how do they perform in comparison to each others in a variety of workload, given that modern stacks have to handle both bulk data transfers over multi-hundred gigabit Ethernet and small request-response messages that require low latency? We found that even representative kernel-bypass stacks have never been compared for a set of basic workloads, likely because of difficulty to run their implementation. This paper takes the first step towards answering that question by comparing six in-kernel or kernel-bypass stacks. We show that existing stacks cannot handle those workloads at the same time or lack generality. We then use those observations to discuss possible pathways towards practical kernel-bypass stacks.

1 Introduction

TCP has been widely used in both the Internet and datacenters. It is not just an interconnect, but a part of the network; TCP interact with applications, physical and virtual networks often operated by SDN controllers, and middleboxes, such as load balancers, firewalls, network observability and telemetry systems, prevalent in cloud infrastructure. Being part of an operating system kernel, TCP has performed reasonably well with the aid of hardware offloading for checksum computation and segmentation. Most of issues raised over time have been remedied by protocol or stack extensions; for example, TFO [63] enables data transmission over SYN packets, MPTCP [65] supports simultaneous use of multiple paths, and Minion [53] preserves application message boundaries. To support modern ML workloads, MLTCP [66] incorporates the application behavior in the TCP congestion control.

Meanwhile, some applications have suffered from slow kernel network stack performance evolution at the *system level* to meet tight latency or throughput requirement over high speed datacenter network fabric. Major criticisms are low throughput and high latency when the server handles a large number of connections [59, 79], often cited as the C10K/C10M problem [37], low throughput scaling to multiple CPU cores due to contentions at CPU caches or locks [24, 32], and high software overheads posed by generality that supports many features and abstractions [23, 45, 21].

Some hyperscalers thus have attempted to deploy a *kernel-bypass* TCP stack built on top of a fast raw packet I/O engine like DPDK [46, 82, 16] and we have seen a surge of such stacks also from academia. Indeed, kernel stacks are primarily optimized for high throughput of bulk data transfers, but at the same time, generality and patch-based development cycle have bloat software complexity and data structure footprint. As a result, the stack cannot drive 100 Gb/s or higher bandwidth networks, respond to millions of RPCs (request-response messages) per second, or save enough CPU cycles for application-level tasks or energy efficiency.

Operators and application developers now face a problem with what network stack they would build or adopt with an understanding of performance characteristics and trade-offs of design options. As it turns out, it has been increasingly apparent that no one knows which stack design or implementation makes a difference in a range of workloads, or even whether they really need a kernel-bypass stack. When a new stack is proposed in the literature, it typically compares against the regular Linux stack and one existing kernel-bypass stack or two. The comparison is thus limited, and the metrics often highlight their strengths or main focus.

Although a new stack is usually designed to solve problems that appear in a specific workload, it is rare that operators can completely ignore performance in another workload—especially in networks or applications where the highly generic TCP protocol is used, rather than in those where specialized non-TCP transport can be adopted for specific workloads [34, 44, 55, 26, 74]. Also, even for the workloads that a proposed stack focuses on, we found that several stacks exhibit opposite results from their reports. This is unsurprising, because the performance of stacks depend on many factors, including application architecture, workload, and hardware or software configuration whose optimal one differs between the stacks.

The implication of adopting or building a new stack is not just moving the TCP implementation out of the kernel and adapting the applications to the new stack (although that is already a substantial engineering effort). Today's systems heavily depend on the kernel stack in both the *software* and *protocol* aspects. The software aspect includes system-wide and application-level resource accounting, fault isolation between the application and stack, protection against application or network protocol misbehavior, and support for queuing disciplines or packet filters controlled by familiar tools or orchestration frameworks. The protocol aspect includes extensions, which mostly rely on TCP options negotiated with the remote

endpoint, and new algorithms for congestion control and loss recovery, which interplay with other traffic in the network.

A new kernel-bypass stack would need to support those features when they are used in production. It is not just an addition of features, but requires development and operation cycles over time, perhaps with a continual community effort, as we have seen in the evolution of the Linux kernel stack that has continually changed by 5–25% LoC modification in each component, each year, at least 2010–2020 [61]. Given that relatively few stacks could afford such an effort for a long term, what would those stacks look like?

Once a high quality, general enough kernel-bypass stack supported by a wider community is available, the clear implication is accelerated evolution of it both in development and deployment, much like what we have seen in success of QUIC often shipped with web browsers. Some users would fork it and some would contribute back the enhancement they made, and new features could be used without OS upgrade, which has lead to slow stack evolution and option deployment in the Internet [48, 28].

We present performance measurement results of six different in-kernel or kernel-bypass network stacks—regular Linux, mTCP [32], TAS [36], F-Stack [16], IX [7] and Demikernel [80]—running on the same hardware, serving the same workloads and using the same application whose I/O is optimized for individual stacks. Selection of the stacks is made based on their design decisions (Figure 1 and Table 1) and availability, although we still needed to implement new features, fix bugs or figure out optimal configuration without document in some to enable fair comparison with their best possible performance.

We examine the effect of network stack design decisions on *basic* workloads, such as simple bulk data transfers and RPCs over different degrees of parallel connections with varying number of CPU cores, which surprisingly have not been studied. As we will describe in § 4–§ 7, those workloads stress varying aspects of the stack. We discuss untested stacks and workloads in § 8.5.

This paper makes three contributions.

- The first is a snapshot of the performance of different network stacks on modern hardware as of 2024, showing none of the stacks perform well for both large and small data transfers at a range of concurrency levels.
- The second contribution is our measurement methodology that allows us to make fair comparison between different stacks for variety of workloads with simple hardware requirements. It also shares our experience of writing an application over multiple stacks with largely different APIs or even programming models.
- Last but not least, we enable third-party software artifact of five representative kernel-bypass stacks for all the basic workloads under a unified application; this would enable others to start research or development on those stacks. We believe this is a crucial step. Alibaba has reported that fixing problems with existing stacks is more expensive than building a stack from scratch due to limited community support [82], but such an approach is only affordable for hyperscalers with a large amount of engineering resources.

2 Motivation

Kernel-bypass stacks are designed and implemented to outperform kernel stacks, but we observe two problems to understand the status quo as a whole. First, most of those are evaluated in a limited set of workloads, disregarding one or more of basic yet salient stack properties that we detail in § 2.1; for example, none of the stacks we test are evaluated for large data transfer[1]. Although datacenter traffic contain many small RPCs, large transfers are also common in storage traffic [4] and machine learning [66]. Second, those stacks are compared only against one or two other stacks, likely due to the lack of practical availability of other stacks.

There exists a growing trend of releasing the artifact to ensure reproducibility of research output, but artifact does not necessarily mean software release supposed to be used by others in their own way. We first review how the kernel stacks have evolved towards better performance by addressing emerging challenges. We then describe the representative stacks we evaluate in this paper, if any, including our modifications required to evaluate their basic stack properties.

The rest of this section describes the key properties used in the kernel TCP stack to serve various workloads. We then review the kernel-bypass stacks we test in terms of their design highlights and usage by the original authors and others, which we also summarize in Table 1.

2.1 Kernel Network Stack

Large sends. Kernel network stacks have been optimized primarily for throughput of bulk data transfer. It is motivated by file transfers over the Internet, which thus has been the common networking performance metric measured by popular tools, including prominent `iperf` [52].

The basic primitive of the kernel stack is the packet data structure (`sk_buff` in Linux and `mbuf` in FreeBSD). It represents arbitrary-sized packet data that can span across multiple packets or kernel pages and can be moved between the layers without data copy, while maintaining shared metadata like pointers to packet headers or sockets and reference counts. When the application writes data on the TCP socket, it is copied to the kernel socket buffer. The kernel then allocates a packet metadata, which is passed down through the stack.

The TCP implementation creates a larger segment than the MTU size preceded by a single TCP and IP header so as to reduce the number of network layer operations required to

[1] Only brief report is provided in [38].

send the same amount of data. This segment is split into MTU-sized packets while adjusting their sequence numbers, packet length and checksum fields at the NIC with TSO or the bottom of the software stack with GSO. The kernel stack can send the next data in the socket buffer, which would have been held due to the lack of the available window, as soon as it receives a positive ACK, because the ACK packets are processed in the interrupt context invoked by the NIC (`softirq` in Linux and `isr` in FreeBSD).

Multi-hundred gigabit Ethernet fabric requires large transfers also to cope with high packet rates, whose problems are discussed next. BigTCP [18] enables TSO for larger segments than 65 K B to further reduce the network layer processing per unit data. High bandwidth networks also create congestion in the host interconnect and memory subsystems, and [2] mitigates it.

Small messages. It is easy for kernel stacks to fill a 10 Gb/s link with large transfers [24], because the transmission time of a 1.5KB packet is 1.2 µs, which is sufficient for the sender to prepare the next segment and push it to the NIC. However, that of a 64 byte packet is 67 ns, which is too short to prepare the next packet, because of software overheads in the protocol stack [24, 32, 72]. Those overheads include system call, packet metadata (de)allocation, socket manipulation such as state update, file descriptor lookup on receive, many of which rely on atomic operations or locking. Short messages are common in datacenters due to RPCs, for example, to retrieve remote in-memory caches, commit database queries and run consensus protocols [70].

Concurrent connections. Concurrent TCP connections are common in the Internet servers, whose challenges are cited as C10K or C10M problem [37, 42, 49]. Those are also common in datacenters due to parallel RPCs [70]. In addition to request processing throughput, tail latency also matters, because it is common that a server accesses many other servers to generate the response to a single request, meaning that one server-server communication delay blocks the response to the client.

Concurrent connections reduce CPU cache locality (e.g., of the connection data structure). Another problem is that the application (thread) needs to process many (per-connection) file descriptors that receive a request. In the conventional `epoll`-based programming model, each thread processes those descriptors in turn, reading the request and writing a response using two syscalls, whose overheads are significant for small data. Concurrent connections thus create request backlogs at the application, leading to queuing delay [79]. The new I/O multiplexing APIs, `io_uring`, reduces the request handling syscalls.

Multiple CPU cores. Modern NICs are equipped with multiple queues processed by different CPU cores to improve request handling throughput over concurrent connections. On receive, Receive Side Steering (RSS) distributes the packets to different queues based on the source-destination 4-tuples to avoid packet reordering in each connection. Those queues

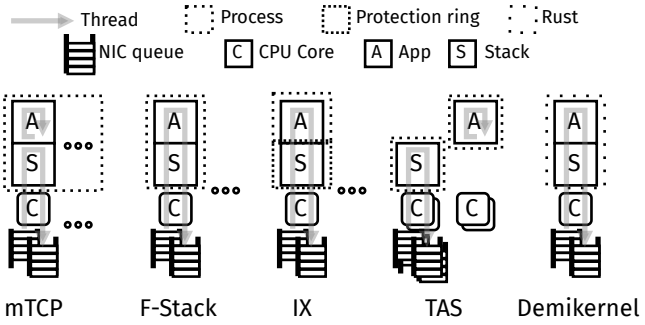


Figure 1: Stack architectures. Details in § 2 and Table 1.

interrupt the corresponding CPU cores (e.g., same index) to parallelize the subsequent packet processing. The send syscall context typically chooses the NIC queue based on the current CPU core to avoid lock contention. However, locks, atomic operations and large memory footprint of data structures impair parallelism across multiple CPU cores, limiting the whole-server throughput. Also, multiple queues do not increase the parallelism inside the NIC; packets across the queues are still serialized at the NIC hardware [73].

2.2 mTCP

mTCP implements the TCP/IP protocol suite from scratch and runs it in user space on top of a raw packet I/O framework (ShaderIO [23], DPDK or netmap [67]) that achieves packet I/O at a very high rate by enabling *batch* packet processing. Although mTCP provides socket-like APIs, scanning connections to process the requests needs just lightweight library calls for every connection, instead of expensive real `read`/`write` syscalls.

The most unique aspect of the mTCP architecture is extensive resource partitioning between the CPU cores. It allocates a *stack thread*, which performs packet I/O and protocol processing, to every application thread, and co-locates the thread pair on the same CPU core to enable lock-free data sharing between the stack and application, as illustrated in the leftmost part of Figure 1. mTCP stack threads even do not share the TCP connection instances; if a stack thread receives a packet that belongs to a connection handled by another thread, it simply discards the packet.

The implementation of mTCP is relatively stable. We have seen its usage or extension in other research projects, some with the mTCP authors [31, 50, 38], but many without [3, 7, 36, 81, 41]. However, the workload is limited; for example, the maximum transfer size per request is 8 K B and the number of CPU cores is at most 8 in the original paper and 24 without application-stack thread pinning in other work [36]; [41] uses 24 cores, but at low request rates (up to 0.2M requests per second). We test mTCP with transfer size up to 2 MB (§ 4) and up to 24 CPU cores (§ 7).

	Architecture	API	TCP impl.	Use by author(s)	Use w/o author(s)
mTCP [32]	App-stack thread pair on the same core	Socket-like (no semantics)	Custom	Up to 8 K B data and 8 cores [32, 31, 50, 38]	Up to 8 K B data or 24 cores [3, 41, 7, 36, 81]
F-Stack [16]	App-level processing in the stack thread	Event callback to the stack	FreeBSD	–	Up to 8 K B data [57, 81], 8 cores [10] or 64 conns. [58]
IX [7]	App-level processing in the stack thread	Packet-level TX/RX buffers	`lwIP` [19]	Up to 8 K B data [38]	Up to 64 B data [36], 4 K B w/ 8 cores [81] or low data rate [41]
TAS [36]	Dedicated threads for TCP data path	Socket-like	Custom	Up to 2 K B data and 24 cores [36, 72, 75]	Up to 0.3 MReqs with high overhead apps [41]
Demikernel [80]	App-level processing in the stack thread in Rust	Packet-level TX/RX buffers	Custom	Up to 16 conns. [64] and 256 K B data [80, 15]	Up to 64 conns. [58]

Table 1: Design and usage summary of the network stacks we test. Their details are in § 2.

mTCP has a significant deployment issue. When it initiates the connection (i.e., used as a TCP client), returning packets must arrive at the same CPU core by carefully choosing the local port number based on the RSS hash function. This restricts the number of usable local ports per destination or compromises the flexibility of choosing the local port. Further, its specific RSS hash seed required at both the client and server also results in poor connection distribution among the cores, which would be problematic when a large number of cores is available. Further, mTCP does not support applications that use different threads for sending and receiving [36].

In our measurement, we needed to simulate the distribution with various seeds and modify mTCP to use the pre-calculated seed. In the real world, the user may need to implement a dynamic load balancing or thread migration mechanism. Therefore, mTCP is easy to use in terms of I/O APIs because those are similar to socket APIs, but its shared nothing architecture complicates the application model and system configuration. We also needed to select, with mTCP code modification, busy-polling or interrupt to handle packet input for each workload to achieve best possible performance.

2.3 F-Stack

F-Stack [16] is a production-quality kernel-bypass stack developed and used by Tencent Cloud. It runs on top of DPDK but ports the FreeBSD TCP implementation to benefit from its advanced TCP features, including RACK [14], TFO [63], ABC [5] and SACK extensions [47, 8], to name a few.

This is a stark contrast to mTCP, which implements optimized TCP from scratch at the expense of supporting only basic features. It should be noted that, since the current TCP has a number of extensions widely used, just implementing RFC793 [77] and RFC1323 [9] does not mean being a *fully-fledged* TCP implementation.

F-Stack employs the architecture where the stack and application run in the per-core process (see Figure 1). Unlike mTCP, both stack processing and application-level tasks are executed in the same thread. Although F-Stack intends to provide socket-like APIs to minimize the porting effort of existing applications, we found that it is not as easy as it sounds due to their core architecture. Since the address space is not shared between processes by default, application-level contexts would need to be relocated to explicit shared memory. Further, event handlers that are registered by the application as callbacks must return control to the stack to restart the stack activity, further hardening the application port. This is a stark contrast to the applications that are running on top of the kernel stack where the OS appropriately schedules the application and stack tasks.

F-Stack is used by several external research projects for either bulk data transfers briefly [38], small messages up to 8 K B [57, 81], 8 CPU cores with up to parallel 1120 connections [10], up to 64 parallel connections [58] or functional test [83]. However, it is unclear how F-Stack works for variety of workloads in the same setup.

2.4 IX

IX runs the stack in a protected CPU ring domain usually used for virtualization. It runs on top of DPDK and employs `lwIP` [19], a portable TCP/IP protocol implementation, thereby inheriting its protocol features and extensions. IX provides a low-level APIs based on event conditions, and operates based on the run-to-completion execution model throughout the stack and application; when a series of packets (in the same or different connections) arrive at a NIC ring, the busy-polling thread processes those packets in the stack, executes the application-level request processing, and then sends the responses through the stack (Figure 1). The thread restarts waiting for new packets after consuming all the packets. IX [7] reports much lower average and tail latency, as well as higher request throughput, than mTCP.

Since IX does not support socket-like APIs, porting existing applications to IX requires significant effort, including changes in both I/O APIs and the application execution model.

IX is the basis of following projects by the original authors, including ZygOS [62], which reduces the tail latency by work stealing, and Shinjuku [33], which further cuts latency with preemptive scheduling. IX has also been used by several research projects without the authors [81, 36, 41]. However, their use of IX is limited to 64 B messages [36], 4 K B messages [81] or low request rates of up to 0.4M per second [41].

In our experiment, since the original IX implementation is unstable, we use the one included in Reflex [39] which is also the work of the common authors. IX in Reflex removes the dependency on Dune and thus the use of protected CPU domain [6]. This actually prevents IX from being disadvantaged by the overheads caused by extra protection that other stacks do not offer, enabling fair comparison in our study.

2.5 TAS

TAS takes a unique approach that replaces the *datapath* of the TCP stack, while leaving control to the kernel. The TAS datapath runs dedicated *acceleration* user-space threads, which execute the packet I/O using DPDK and TCP/IP protocol processing and post the data to the application that runs in separate threads (Figure 1). Therefore, TCP datapath functionalities, such as congestion control, loss recovery and packet filtering, are not inherited from the kernel stack and must be implemented in TAS.

Application porting effort required to use TAS is relatively low. TAS preloads the POSIX I/O library to move the data between the acceleration and application threads with ease, meaning that the application can still use the POSIX system call. While some socket options are not supported, no major modification to the application source code was needed. Unlike the stacks mentioned earlier, it does not use the shared-nothing architecture or run-to-completion, which significantly contributes to the low application porting effort. On the other hand, it does not support modern TCP extensions.

TAS has been used in following projects by its original authors [72, 75], but to the best of our knowledge, without the TAS authors, it was used only at low request rates [41].

TAS had a significant problem to test. First, we noticed that TAS does not implement the Window Scaling (WS) option when we observed very low bulk data transfer throughput. The lack of large window usage does not only limit the throughput, but leads to incompatibility with stacks that do not support WS-incapable remote endpoint, which is the case for mTCP. Even after implementing this option, we found that TAS cannot use the resulting large window; we further added support for that.

Moreover, TAS is sensitive to configuration. Since its default internal buffer (equivalent to kernel socket buffer) size is very small, the application needs to call write commands repeatedly until the data is fully written, causing inefficiency. TAS limits the bandwidth usage by default to optimize for load balancing among multiple connections. However, it is harmful for bulk data transfers. Finally, the user can manually assign CPU cores to the application and stack threads. This means that there exist a large search space for optimal core assignment where the optimal setting depends on workload and application and can be verified only based on measurement of the resulting performance. This is a stark contrast to the kernel stack that shares cores between the stack and applications. We searched for the optimal split of cores between the stack and application in multi-core experiment (§ 7).

2.6 Demikernel

Demikernel augments various kernel-bypass I/O frameworks like DPDK and RDMA, and protects the stack and application by Rust. It is designed to achieve ultra-low latency. Demikernel supports generating Rust coroutine, a lightweight context switching mechanism, which is advantageous compared to the POSIX thread-based context management. It provides a new asynchronous I/O API and employs the run-to-completion execution model. Its TCP/IP protocol implementation is built from scratch, similar to mTCP and TAS.

Demikernel has been used by the original authors for their following projects [64, 15], but also by other authors [58]. Demikernel has been tested only with small numbers of concurrent TCP connections, up to 16 by the original authors [64] or 64 by others [58].

Porting existing applications to use Demikernel is difficult in several aspects. Demikernel assumes that the application has already segmented their data into MTU-sized chunks. This would be troublesome for bulk data transfers. Its I/O event model also differs from that of regular kernel stacks, which is particularly relevant when handling concurrent TCP connections. For example, in contrast to Linux `epoll_wait` that notifies the caller of ready-to-process events, the equivalent function of Demikernel sends event completions. To follow this model, the developer might not only just replace API calls but restructure the application. Finally, due to its single-threaded design, it is hard to utilize stack-level multi-core parallelism. Therefore, although the I/O interfaces could be wrapped to resemble POSIX APIs, it would be hard to use Demikernel in terms of application threading and event handling.

3 Methodology

Flexible yet simple measurement methodology is crucial to characterize stack performance. Existing stacks typically use their own custom application for basic measurement, which utilizes the underlying stack most efficiently with minimum overheads. However, since those applications use their own application-level protocol, one cannot be used to evaluate another stack. Further, those applications do not support all the workloads we would like to test.

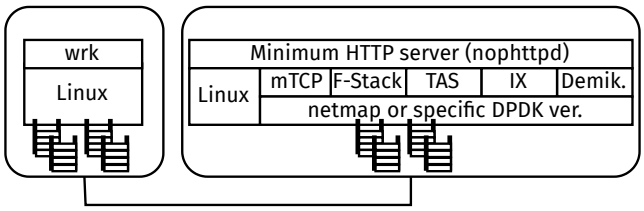


Figure 2: Experiment software architecture in the client and server machine connected back-to-back.

Some of the stacks are ported to an existing HTTP server application, such as `lighttpd` or `nginx`, but it includes significant application-level overheads, which is undesirable to highlight performance characteristics of the stack itself. Further, different stacks are ported to different HTTP servers, whereas those HTTP servers have different architectures and complexity, meaning that we cannot compare "HTTP server X on stack A" against "HTTP server Y on stack B". Memcached has been ported to all the stacks, but they are different in versions and exhibit high application-level overheads, similar to HTTP servers.

Server. We thus implement `nophttpd` (Figure 2 right), a custom TCP server that simply serves a static in-memory data in response to an HTTP request. Exploiting the advantage of building the application by ourselves, `nophttpd` is optimized for individual stacks—Linux, mTCP, F-Stack, IX, TAS, and Demikernel (Catnip TCP stack)—to handle all of bulk data transfer, small RPCs, concurrent connections as efficiently as possible and well utilize multiple CPU cores, while addressing most of the API challenges posed by those stacks, described in § 2.2–§ 2.6. When running on regular Linux, `nophttpd` uses `epoll`-based event loop. For Demikernel, we implement `nophttpd` in Rust to use its native APIs directly.

We do not opt for implementing a wrapper that unifies the interfaces of all the stacks and the minimum HTTP server logic on top of it. This is because the stack's execution model largely differs between the stacks, not just the function names or arguments. Therefore, such an unified wrapper approach would advantage or disadvantage particular stacks, making the comparison unfair. Our approach of tightly coupling the application logic with the native API of each stack might be impractical when building an application in practice, but this approach is suitable for the purpose of fair comparison between the stacks.

Client. Another challenge is the client. Some existing work [36, 32] employ multiple client machines that run regular Linux stack to saturate the kernel-bypass server, because the kernel-bypass stack is faster than in-kernel one by a large margin particularly for small messages. However, the use of multiple clients complicates the measurement workflow and requires additional hardware resources.

We thus use a single client machine. When we test multi-core scalability of the server stack, we *downclock* the server CPU clock frequency, as done in [67, 27], to prevent the client from becoming the bottleneck. We use the `wrk` benchmark tool for its useful stat report and straightforward implementation. However, we found it has multi-core scalability issue in its stat management mechanism, which creates a contention between threads when their number is high. We thus modified `wrk` to aggregate the stats across the threads in a more coarse granularity and use thread-local buffers to store intermediate records. `wrk` establishes one or more persistent TCP connections to the server and repeats sending a request and receiving a response in every connection. Our methodology might seem limited, but as we will show in the next sections, we can characterize stacks with variety of workloads.

Workload. We measure bulk data transfer throughput, small message latency with unloaded client and server, and concurrent message throughput and latency over multiple connections that stress the server. Those workloads are simple, but stress different parts of the server network stack, including per-byte or per-message transfer efficiency, connection scalability, and multi-core scalability. We believe our measurement is worthwhile, because existing work does not test the stacks across all these aspects simultaneously in a consistent setup, as discussed in § 2.

Hardware and system configuration. We use a pair of identical machines connected back-to-back. Each machine is equipped with Intel Xeon Gold 5418N (Sapphire Rapids) clocked at 1.8Ghz and Intel XXV710-DA2 25GbE NIC. Both machines run Linux kernel 5.18 unless otherwise stated. We deactivate unused cores with CPU hotplug in Linux to prevent the kernel from performing relevant activity to the test on those cores. We disable segmentation offload (TSO) for all the stacks for fair comparison, as mTCP and Demikernel do not support it; if it was enabled, it would provide larger degree of improvement in stacks with higher per-packet processing costs (i.e., lower small message throughput), because TSO reduces the packet rate in the stack. We run each benchmark for 8 seconds.

4 Bulk Data Transfer

We first measure bulk data transfer performance of the stacks. Since we test each stack running at the server, the client issues a small request that triggers a large response to test the ability of the server stack to send large packets continually, which is clocked by receiving ACK packets. Many kernel-bypass stacks do not consider this workload much, although large transfer still happen in many scenarios; storage, search and web systems are a few of the examples [56, 1].

On the regular socket `write`, the kernel copies up to as much application data as the available kernel socket buffer. Partial write is not a problem, because the socket informs the caller of how much the data is written, and TCP semantics is a byte stream, like a file. In other words, the application can `write` arbitrary sized data into the socket. Kernel-bypass

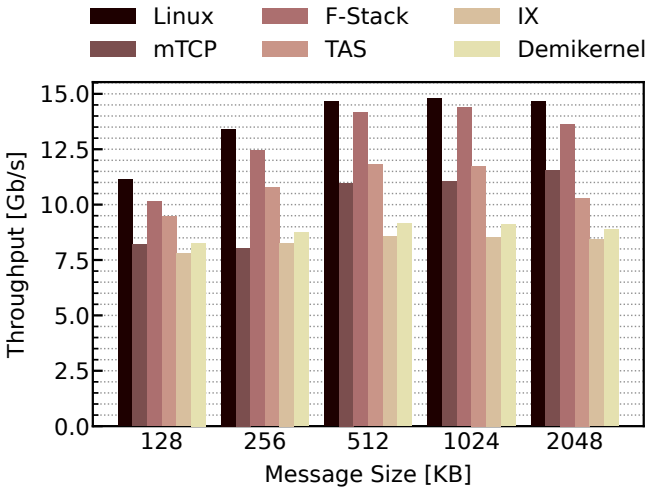


Figure 3: Bulk transfer throughput over a single connection.

TCP stack APIs vary. IX and Demikernel require the application to split the data so that each fits into a single packet, as described in § 2.4 and § 2.6, respectively. We omit the large receive performance measurement, because it is more lightweight than sends for large data. Receive-side processing performance matters for small messages due to high packet arrival rates; our experiment with small messages in the next sections stresses the receive path of the stacks.

Figure 3 shows the throughput of Linux stack, mTCP, TAS, F-Stack and Demikernel. `nophttpd` sends up to 2MB of a message over a single connection as the response to a small request issued by the client. All the stacks use a single CPU core, except for TAS, which requires minimum two cores, one for the application and the other for the accelerated datapath.

The Linux stack achieved the highest throughput, peaking at 14.69 Gb/s with 2 MB messages over a single connection. This is unsurprizing, because it has long been optimized for it. F-Stack achieved 13.66 Gb/s, following the Linux stack.

To shed light on this observation, we profiled mTCP and F-Stack using Linux `perf`. The Linux stack consumes 50.5% of the CPU time in TCP, IP and Ethernet protocol processing, whereas F-Stack consumes 52.5%; `memcpy`, whose overheads are dominant in large buffer processing, is not included in those overheads. Since the Linux stack and F-Stack can process 880 and 735 requests per second, respectively, this translates that the Linux stack can process TCP packets 19.6% more efficiently than F-Stack $(1-(50.5/880)/(52.5/735))$.

Given that the FreeBSD TCP implementation is optimized as much as Linux [71], our results suggest that the overhead of emulating the kernel environment to run the FreeBSD stack could offset the I/O efficiency of DPDK, at least for this workload. Further, it is also possible that the run-to-completion execution model of F-Stack fails to push the next packets fast enough in response to the ACK packets.

Other stacks largely fall behind. mTCP and TAS exhibit 10–12Gb/s in throughput. Those stacks are not particularly optimized for bulk data transfer, but significant performance penalty for such workload would prevent operators from adoption of them. Demikernel and IX exhibit the lowest throughput. This is partly due to their inefficient APIs that the application must fill the packet-sized data fragments, resulting in poor interaction between the application and network I/O for multi-packet data. Also, similar to mTCP and TAS, their run-to-completion model could fail to ACK-clock the large transfer efficiently.

5 Small Messages

Latency of small messages has been crucial in datacenters, because servers handle many RPCs, such as in-memory key-value cache access for small items and API calls. In this section, we discuss *unloaded* latency, where the message or packet is never queued in the network, host interconnect or the host software stack, to directly highlight basic software overheads of the stack; we discuss latency in the presence of concurrent requests and connections in the next section.

To take the lowest possible RTT of each stack whose performance is determined by the software and context switch (if any) overheads, the client triggers a small response (64 B payload preceded by the HTTP header) for a small request over a single connection and CPU core. We disable interrupt moderation at the NIC in this experiment, so that the receive interrupt can be caught quickly, which is crucial for mTCP and Linux stack. For the Linux stack, the application busy-polls the event descriptor (i.e., on `epoll_wait`) to avoid the wake-up procedure of the application that is blocking.

Figure 4 plots the P50 request RTT over a single connection with varying response size; we indicate P99 RTT with a number on each bar. Surprisingly, the kernel stack latency is comparable to other kernel-bypass network stacks, achieving 17 µs for 64 B messages. However, it is still higher than Demikernel, which achieves 13 µs, the lowest among all the stacks we test.

mTCP's high latency is expected, although the paper [32] does not report the latency. This is because one of its design decisions is to aggressively exploit packet-level batching. Further, the mTCP stack and application threads run on the same CPU core in user space, which would exhibit high context switching overheads, although maximizing cache locality and enabling lock-less stack-application communication for high multi-core scalability. F-Stack's high latency could come from the same reason with the lower bulk transfer throughput in comparison to Linux—the kernel emulation overheads to run the FreeBSD TCP/IP implementation would incur high software overheads. Low-latency stacks such as IX and Demikernel start showing higher P50 and P99 latency than Linux when the message sizes are larger than 32 K B, perhaps due

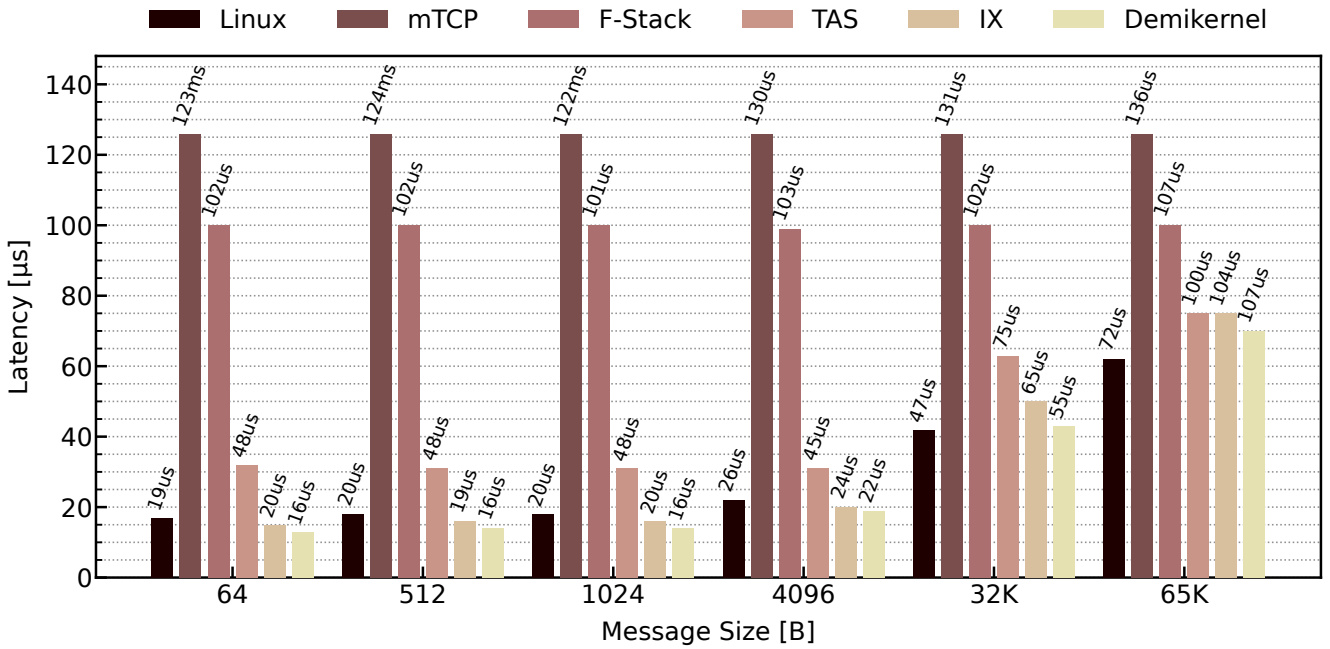


Figure 4: RTT at P50 (bars) and P99 (numbers on the bars) in a single connection.

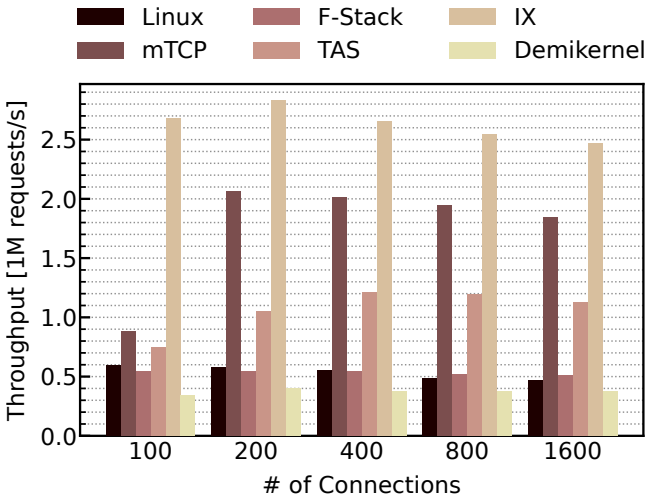


Figure 5: Throughput over concurrent TCP connections.

to the same reason with their low bulk transfer performance discussed in § 4.

We observe similar characteristics of P99 latency to P50 overall, but mTCP exhibits extremely high tail latency, for example, as much as 124 ms.

6 Concurrent Connections

We now turn our attention to concurrent requests over multiple connections, as it is a common pattern for datacenter RPCs (§ 2.1). In the stack that executes receive packet processing in a separate thread from the application (Linux, mTCP and TAS), when the stack drains the NIC queue more slowly than the packet arrival rate, the queue forms at the NIC; when the application consumes the requests more slowly than the stack, the queue forms at the application. In the stack that processes both receiving packets and application-level task in the same thread in a run-to-completion manner (F-Stack, IX and Demikernel), the backlog forms at the NIC.

Figure 5 plots the throughput with the single CPU core, except for TAS which requires at least two cores; we discuss the multi-core cases in the next section. Figure 6 plots request latency at P50 (bars) and P99 (numbers on the bars).

In throughput, IX consistently outperforms others by a large margin, 30–684%. mTCP outperforms TAS by 17–95% although TAS uses one more core in total. Those results differ from the report of the TAS paper. For those numbers of TCP connections, the TAS paper reports similar throughput between TAS and IX, which also differs in our experiment. Those observations are likely due to the different CPU core allocation, where we allocate the minimum number of cores in this experiment.

Interestingly, F-Stack and Linux perform almost the same, again implying the overheads of kernel emulation that offsets the effect of efficient DPDK packet I/O.

TAS achieves low P50 latency, close to IX, but it could be due to almost only half of the IX throughput. Linux exhibits low throughput and high latency, which exacerbates with increasing numbers of concurrent connections. This could be attributed to both high stack processing overheads and application-level per-request processing costs with syscalls.

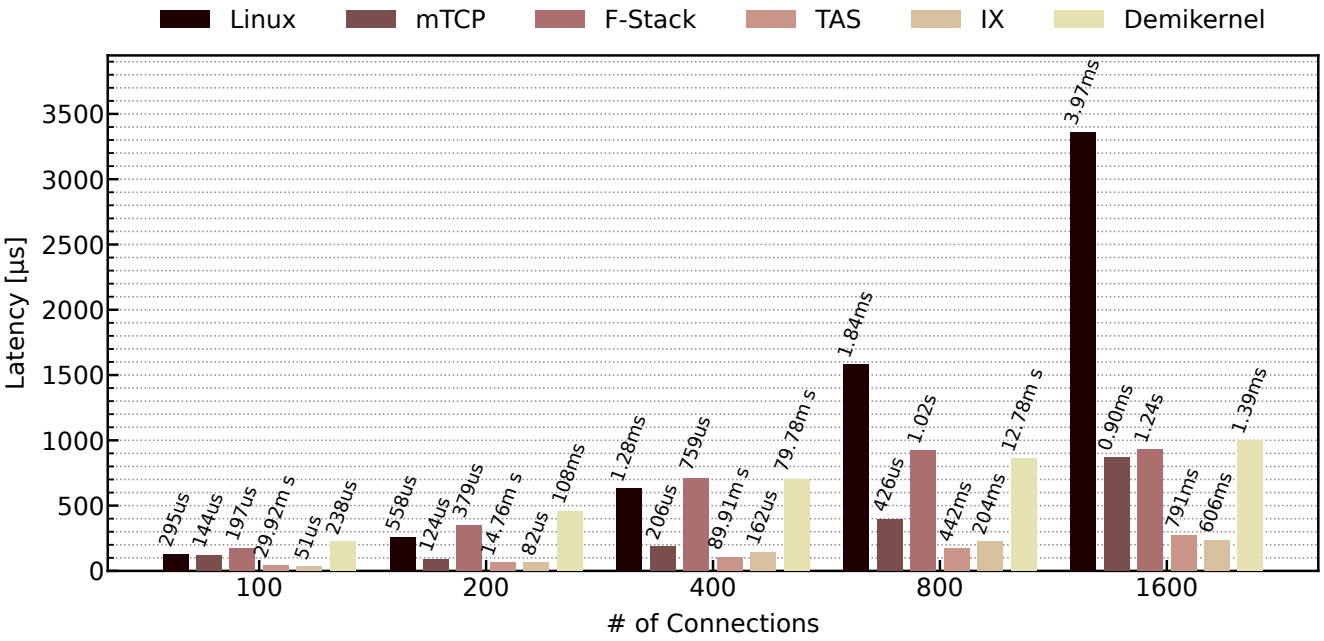


Figure 6: RTT at P50 (bars) and P99 (numbers on the bars) over concurrent connections.

IX achieves 73–85% lower P50 latency than Demikernel, which was introduced later than IX for low latency, while achieving much higher throughput; IX also achieves lower tail latency than Demikernel for up to 400 connections, where the results invert. Note that Demikernel is not compared against IX in their paper. F-Stack, TAS and Demikernel exhibit high tail latency when the connection count is high.

7 Multicore Scalability

Finally, we test how the single-core performance of each stack scales over multiple CPU cores.

Figure 7 plots the throughput over increasing numbers of CPU cores; since the server is unsaturated in many cases (i.e., the client becomes the bottleneck), we also plot the results with the server whose CPU cores are *downclocked* in Figure 8, which we use for discussion in this section. We use 4800 concurrent connections for all the cases. We were unable to test Demikernel, because it does not support multi-core parallelism in the stack. The low throughput of IX at 24 cores is due to a bug in IX, which we were unable to fix. For TAS, we report the numbers with the best-performed stack/application core assignment, which we manually identified; as a result, we assigned half of the total CPU cores to the stack and the rest to the application. More complex applications would need more cores in the application.

IX performs the best in many cases up to 16 CPU cores, outperforming TAS by 16–384% and mTCP by up to 456%. Note that the IX paper [7] reports multi-core scalability up to 8 CPU cores. TAS [36] reports slightly higher throughput than IX in their test with up to 16 CPU cores, but our result shows the opposite. The possible reason is that [36] uses their own custom key-value store, so it might be more optimized for their proposal, or that they could have used different IX implementation, where we use Reflex (§ 2). On the other hand, TAS indicates a sign of the best multi-core scalability, exhibiting 67% throughput increase when the number of cores increases from 16 to 24.

mTCP scaling decays at 24 cores, increasing throughput only by 8.7% when the number of cores increases from 16 to 24 cores, although it improves the throughput by 71% when the cores increase from 8 to 16 cores. The original mTCP paper [32] measured up to 8 cores and scaled well. Therefore, the scaling limit we observed could be due to higher degree of CPU cache contention, which is insignificant when the number of cores is small.

F-Stack scales relatively well, but always stays at a lower side in comparison to IX, TAS and mTCP.

8 Takeaway Lessons

In this section we discuss lessons learned in our work and possible directions towards practical network stacks.

8.1 Application Development Effort

Although mTCP, TAS and F-Stack argue compatibility with existing applications or minor porting effort, our experience suggests significant engineering effort for their adoption. F-Stack enforces a multi-process model for concurrency. Although our application is multi-threaded, it was straightfor-

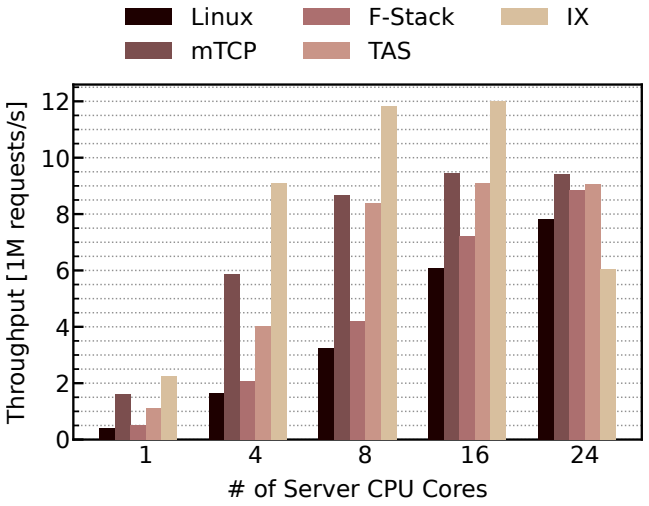


Figure 7: Throughput over multiple CPU cores.

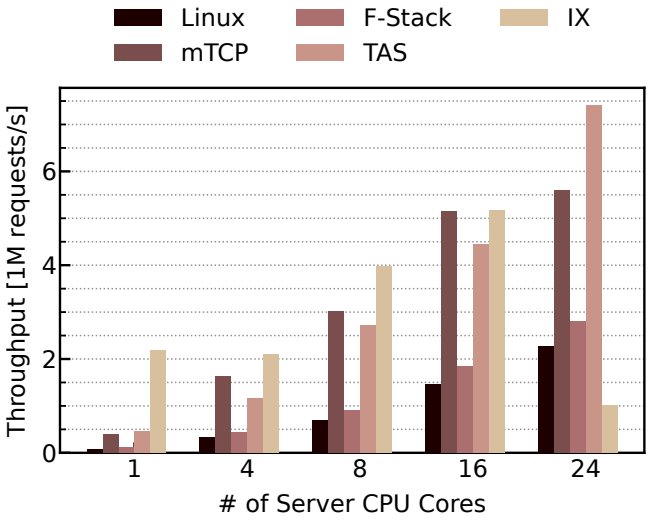


Figure 8: Throughput over multiple CPU cores downclocked to 800MHz.

ward to switch to multi-process model, because it does not have shared state. However, this would be a significant burden for most of the real-world, more complicated applications.

mTCP uses a context structure to identify the connection, instead of an integer descriptor for the kernel stack. Since it is a common practice for applications to index connections by descriptors, which also increase from the lowest value, those applications could need to redesign connection management.

TAS transparently redirects system calls issued by the application to its own library using LD_PRELOAD. However, the lack of many kernel TCP features led to runtime errors, for example, for unsupported `setsockopt`s, which would be problematic real-world, more complicated applications than ours. Nevertheless, its application porting effort is still relatively low.

Demikernel uses asynchronous calls to communicate with applications, requiring many existing applications to be rewritten. Although it provides C interfaces alongside the Rust interfaces, manual data segmentation and unusual event handling require a significant effort to port existing applications. We experienced similar issues in IX. Moreover, since Demikernel is single threaded and enforces Rust co-routines to exploit parallelism, the application cannot exploit I/O-level parallelism, which is the reason why we were not able to test multi-core scalability of Demikernel.

8.2 Configuration and Parameter Tuning

Kernel-bypass stacks fail to keep up with updates of their depending library. Although Demikernel, F-Stack, and TAS rely on DPDK, since those stacks require different versions of DPDK or other libraries, we were unable to reuse the same kernel or I/O-related library installation. We also found that some hard-coded parameters in the stack's source code negatively impact performance. Most stacks also require specific NICs, and porting those stacks to another (newer) NICs could require unpredictable engineering effort, as reported in [82]. In particular, IX requires the custom compilation and link process, using a custom linker script. This further complicates the stack porting process to another environment. Also, stack traces taken by `perf` were collapsed in IX, making our performance analysis harder. The authors might have their in-house updates for their newer projects, but we were unable to find those. Research community would significantly benefit as a whole if we somehow incentivize software maintenance and update of research output.

A sub-optimal stack configuration also leads to lower performance. As discussed in § 2.2, mTCP needs a specific RSS hash seed for both the client and the server just to function due to its resource partitioning; likewise, as discussed in § 2.5, TAS required significant configuration effort for buffer tuning, rate limit deactivation, and stack/application CPU core ratio.

We conclude that there is no one-size-fit-all configuration for kernel-bypass stacks. The optimization methodology for the Linux stack is relatively well-known, but for kernel-bypass stacks, their distinct architectures (e.g., Figure 1) make prediction of their performance characteristics difficult. Therefore, we needed to employ some degree of *brute-force* approach to find a combination of optimal configuration options.

8.3 Understanding Stacks Better

It is essential for operators and application developers to understand their stacks and debug or optimize them when opportunities are found. Two challenges have been observed in this work.

Profiling and code knowledge dependency. The Linux stack has a range of tools, such as `perf`, Kernelshark [68] and PacketDrill [13], but the kernel-bypass stacks do not have any

of those. Those tools require the users to know where to look in the source code, because those are based on trace points. However, they are still useful, as kernel stacks are reasonably well documented and code organization is well known. This is not the case for kernel-bypass stacks. Therefore, even if the equivalent tools were available, it is questionable how practical they are.

NSight [22], although not publicly available at the time of this writing, would remedy this problem. It traces application-level messages between the application and NIC even across multiple threads (e.g., stack thread and application thread in TAS or mTCP), whose interplay is difficult to track. Crucially, NSight can diagnose the source of latency without understanding of the code, except for the API layer of the stack for message identification.

Fine-grained performance analysis. We used Linux `perf` to generate CPU flame graphs that visualize the amount of CPU resources that specific functions in the kernel or user space have been consuming and to see the function call graphs. It was helpful to find the performance bottleneck, or identify unexpected idle states in the stack. Note that the live per-core CPU usage stat is useless particularly for kernel bypass stacks, because most of them busy-poll the NIC I/O queues and shared memory regions and thus exhibit 100% utilization irrespective of request arrivals.

Since it is sampling based, `perf` has several limitations. First, since its flame graph only shows the average processing time of each function, not its variation, it cannot identify the source of tail latency. Further, context switches can not be traced, even though recent stacks have more complex thread models or scheduling policy that affect the performance directly. Those problems would be remedied by NSight.

However, since the time granularity of NSight is a few 100s of ns, it would not be suitable for fine-grained software overhead analysis. Although it is common to measure stack software overheads by directly taking timestamps in the kernel stack by source code modification (e.g., [60] and [11]), NSight would not ease such measurement. SimBricks [43], a network and host hardware simulator, could be useful for fine-grained performance profiling, if the problem can be reproduced in the simulation environment and accommodated by the single-core host simulation, which is a limitation of SimBricks. For example, it could be used to understand interaction between mTCP's stack and application thread co-located on the same core, which we struggled during this work.

8.4 Optimizing Kernel or Kernel-Bypass Stack

We observed that the kernel stack performed the best for bulk data transfers, outperforming IX by 1.7×, whereas IX achieved the best for the rest, for example, outperforming the kernel stack in small message throughput by 5.2×. If a larger community wants a general-purpose, fast TCP stack for datacenters or cloud, should we optimize the kernel stack for small messages, or IX for large transfers?

Our results show that the most serious issue with the Linux stack is scalability to large numbers of connections. Linux stack latency under unloaded resource is not the same but still comparable to kernel-bypass stacks as long as it employs busy-polling like kernel-bypass stacks (§ 5). It scales relatively well over multiple CPU cores, at least up to 8M requests per second (§ 7). Multi-core scalability and its stability under mixed workloads will be further improved with flexible resource allocation techniques like NetChannel [12].

Connection scalability of the kernel stack would be improved by new APIs like `io_uring` that reduce the request backlog at the application, but it is still questionable, because F-Stack, which uses the FreeBSD TCP implementation but eliminates the syscall overheads, still experience limited connection scalability (§ 6). We thus expect similar issues in other kernel enhancement techniques, like MegaPipe [24] (tested with up to 100 connections) and Stackmap [79] (tested with up to 250 connections in [29]).

An operational implication of low connection scalability is that alleviating this problem requires reducing the number of connections per sever, which necessitates installing more servers at the expense of additional space and power footprint, in addition to the server hardware cost.

Since connection scalability and multi-core scalability of kernel-bypass stacks—particularly IX and TAS—are promising, enhancing them for bulk transfer could enable general kernel-bypass TCP stacks. However, there are several challenges. First, it is unclear whether IX's run-to-completion model suits bulk transfers, because the stack's activity to push new data packets is paused while the application processing is happening. Finer-grained scheduling, as done in Shenango [54], ZygOS [62], Shinjuku [33] and Junction [21], could be repurposed for this task, but the actual effect is an open question, because those are designed for low tail latency of small RPCs. Optimizing TAS would be easier in this aspect, because the stack uses dedicated threads separated from the application. It also tolerates skewed loads between the connection. On the other hand, this architecture compromises the advantage of the run-to-completion model that can partition the core resource between the application and stack in a work-conserving manner.

Operators would save hardware, space and energy costs with those stacks, but it comes at the expense of additional stack software development and maintenance, application porting effort, and configuration at deployment. To see real-world deployment of kernel-bypass stacks outside hyperscalers with a large amount of internal engineering resources, we believe that a stack needs to attract heavy community involvement for a long term, much like what we have seen in kernel stacks and other network software, like XORP [25].

We may want multiple stacks in the same address and port number space and on the NIC, as in MultiStack [28],

but offloading expensive (de)multiplexing between the stacks into the NIC hardware, which is trivial today (e.g., Linux `tc-flower` offload). This approach could enable application-specific specialization, as in Sandstorm [45].

8.5 Untested Stacks and Workloads

Since primary motivation is to explore the kernel-bypass stacks that are viable in the real world and evolve long-term, this work has not tested kernel-enhancement approaches. Those approaches include MegaPipe [24], which improves short-lived connection performance and multicore scalability, Stackmap [79], which improves small message performance, and NetChannel [12], which improves various aspects of the stack performance. All of those approaches define new APIs.

We did not cover stacks that utilize specific hardware, including AccelTCP [50], which needs flow processing units at the NIC, and FlexTOE [72], which requires SmartNICs equipped with general purpose CPU cores, as we do not have. However, FlexTOE is based on TAS, and AccelTCP is based on mTCP, which we tested. iip [78] is a portable TCP/IP implementation, potentially replaces `lwip` and thus improves IX. Warpcore [20] is a kernel-bypass UDP/IP stack designed for QUIC, which could also be used to implement kernel-bypass TCP on top of it. LKL [76] runs the Linux kernel stack in user-space, but reports very low performance; once optimized, we expect similar performance to F-Stack at best.

We did not test the short-lived TCP connection performance, because many applications today reuse the connections over many RPCs. We leave the latency measurement under capped throughput as future work. This workload is tricky, because that needs to make the load on each stack, rather than the request rate, identical to make fair comparison. Finally, we did not test the parallel bulk transfer throughput, because our NIC bandwidth is limited to 25Gbps.

9 Related Work

Stack performance characterization.. Linux kernel performance is extensively studied in [11], but no comparison to kernel-bypass stacks. [67] and [60] analyze the UDP stack overheads in FreeBSD and Linux, respectively, both highlighting the syscall overheads. [79] and [7] measure the TCP stack overheads in Linux and report latency posed by software overheads. A recent study [10] analyzes the performance of Linux and F-Stack, and attempts to bring the techniques used in the kernel-bypass stacks, such as polling. [17] reports deployment experience of (Linux kernel) DCTCP in Meta's datacenters, and discusses their improvement based on eBPF, providing useful insight about network stack features required in production systems.

Host hardware performance characterization.. A body of work characterize host interconnect behavior, which would help understand the TCP stack performance at very low latency and high throughput. [51], [30] and [40] characterize data transfer performance of PCIe bus. [69] emulates the emerging CXL-based PCIe communication performance. [35] characterizes the RDMA performance.

10 Conclusion

Summary of findings in this paper are:

- Linux kernel stack achieves highest large send performance, but its high software overheads incurs low throughput and high latency with small messages over a large number of connections. Its bulk transfer performance would be further improved by modern optimizations [18, 12], but those software overheads would not.
- mTCP [32] achieves relatively high connection scalability, only outperformed by IX. It outperforms TAS at up to 16 CPU cores. However, it exhibits high latency and limited multi-core scalability. Further, mTCP's requirement of symmetric RSS hashing would impose significant practical constraints.
- F-Stack [16] is stable, but porting existing applications is not easy, and performance improvement over Linux is small. It also underperforms on bulk data transfer workloads.
- TAS [36] indicates the best multi-core scalability with low application porting effort. Its asymmetric core assignment between the stack and applications have both pros, which allow the use of heterogeneous CPU cores like those in DPUs [72], and cons, which requires careful core assignment. Further, it does not perform as good as IX overall, despite its datapath implements TCP from scratch and thus minimum extension support.
- IX [7] performs overall the best except for large transfers. Its low-level APIs incur high application porting effort, and its implementation is unstable, as it does not work with some numbers of cores. IX is also hard to debug due to its specific compilation and linking process. IX's tail latency is relatively high, but its successor, such as ZygOS [62] and Shinjuku [33] would mitigate it.
- Demikernel [80] achieves the lowest latency but only with a small numbers of TCP connections, and its clean-slate APIs would incur significant engineering effort to run existing applications or write new ones. Its single-threaded stack design would also limit large transfer throughput even with a substantial optimization.

Network stack performance measurement is a needed area. Although detailed analysis of many results have been left for future work, we believe this work is valuable because it has opened up the first step towards studying kernel-bypass network stacks through the case study. The application and stack code used in this paper are available at https://github.com/uoenoplab/stackbench.

Acknowledgments

We are grateful to anonymous NSDI and ATC reviewers and our shepherd, Kartik Gopalan for their insightful feedback. We thank Steven W.D. Chien and Marios Kogias for initial setup to run IX and Suhas Narreddy for testing mTCP over netmap. This work was in part supported by EPSRC grant EP/V053418/1.

REFERENCES

[1] Vamsi Addanki, Oliver Michel, and Stefan Schmid. "PowerTCP: Pushing the performance limits of datacenter networks". *USENIX NSDI*. 2022.

[2] Saksham Agarwal, Rachit Agarwal, Behnam Montazeri, Masoud Moshref, Khaled Elmeleegy, Luigi Rizzo, Marc Asher de Kruijf, Gautam Kumar, Sylvia Ratnasamy, David Culler, et al. "Understanding host interconnect congestion". *ACM HotNets*. 2022.

[3] Abdul Alim, Richard G Clegg, Luo Mai, Lukas Rupprecht, Eric Seckler, Paolo Costa, Peter Pietzuch, Alexander L Wolf, Nik Sultana, Jon Crowcroft, et al. "FLICK: Developing and Running Application-Specific Network Services". *USENIX ATC*. 2016.

[4] Mohammad Alizadeh, Albert Greenberg, David A Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. "Data center tcp (dctcp)". *ACM SIGCOMM*. 2010.

[5] Mark Allman. *TCP Congestion Control with Appropriate Byte Counting (ABC)*. RFC 3465. Feb. 2003. URL: https://www.rfc-editor.org/info/rfc3465.

[6] Adam Belay, Andrea Bittau, Ali Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. "Dune: Safe user-level access to privileged CPU features". *USENIX OSDI*. 2012.

[7] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. "IX: a protected dataplane operating system for high throughput and low latency". *USENIX OSDI*. 2014.

[8] Ethan Blanton and Mark Allman. *Using TCP Duplicate Selective Acknowledgement (DSACKs) and Stream Control Transmission Protocol (SCTP) Duplicate Transmission Sequence Numbers (TSNs) to Detect Spurious Retransmissions*. RFC 3708. Feb. 2004. URL: https://www.rfc-editor.org/info/rfc3708.

[9] David A. Borman, Robert T. Braden, and Van Jacobson. *TCP Extensions for High Performance*. RFC 1323. May 1992. URL: https://www.rfc-editor.org/info/rfc1323.

[10] Peter Cai and Martin Karsten. "Kernel vs. User-Level Networking: Don't Throw Out the Stack with the Interrupts". *Proceedings of the ACM on Measurement and Analysis of Computing Systems* (2023).

[11] Qizhe Cai, Shubham Chaudhary, Midhul Vuppalapati, Jaehyun Hwang, and Rachit Agarwal. "Understanding host network stack overheads". *ACM SIGCOMM*. 2021.

[12] Qizhe Cai, Midhul Vuppalapati, Jaehyun Hwang, Christos Kozyrakis, and Rachit Agarwal. "Towards μ s tail latency and terabit ethernet: disaggregating the host network stack". *ACM SIGCOMM*. 2022.

[13] Neal Cardwell, Yuchung Cheng, Lawrence Brakmo, Matt Mathis, Barath Raghavan, Nandita Dukkipati, Hsiao-keng Jerry Chu, Andreas Terzis, and Tom Herbert. "packetdrill: Scriptable network stack testing, from sockets to packets". *USENIX ATC*. 2013.

[14] Yuchung Cheng, Neal Cardwell, Nandita Dukkipati, and Priyaranjan Jha. *The RACK-TLP Loss Detection Algorithm for TCP*. RFC 8985. Feb. 2021. URL: https://www.rfc-editor.org/info/rfc8985.

[15] Inho Choi, Nimish Wadekar, Raj Joshi, Joshua Fried, Dan RK Ports, Irene Zhang, and Jialin Li. "Capybara: μSecond-Scale Live TCP Migration". *ACM APSys*. 2023.

[16] Tencent Cloud. *F-Stack*. http://www.f-stack.org/.

[17] Abhishek Dhamija, Balasubramanian Madhavan, Hechao Li, Jie Meng, Shrikrishna Khare, Madhavi Rao, Lawrence Brakmo, Neil Spring, Prashanth Kannan, Srikanth Sundaresan, and Soudeh Ghorbani. "A large-scale deployment of DCTCP". *USENIX NSDI*. Apr. 2024.

[18] Eric Dumazet. *tcp: BIG TCP implementation*. https://lwn.net/Articles/883713/.

[19] Adam Dunkels. "Design and Implementation of the lwIP TCP/IP Stack". *Swedish Institute of Computer Science* (2001).

[20] Lars Eggert. "Towards securing the Internet of Things with QUIC". *Proc. Workshop Decentralized IoT Syst. Security (DISS)*. 2020.

[21] Joshua Fried, Gohar Irfan Chaudhry, Enrique Saurez, Esha Choukse, Íñigo Goiri, Sameh Elnikety, Rodrigo Fonseca, and Adam Belay. "Making kernel bypass practical for the cloud with junction". *USENIX NSDI*. 2024.

[22] Roni Haecki, Radhika Niranjan Mysore, Lalith Suresh, Gerd Zellweger, Bo Gan, Timothy Merrifield, Sujata Banerjee, and Timothy Roscoe. "How to diagnose nanosecond network latencies in rich end-host stacks". *USENIX NSDI*. 2022.

[23] Sangjin Han, Keon Jang, KyoungSoo Park, and Sue Moon. "PacketShader: A GPU-Accelerated Software Router". *ACM SIGCOMM*. 2010.

[24] Sangjin Han, Scott Marshall, Byung-Gon Chun, and Sylvia Ratnasamy. "MegaPipe: A New Programming Interface for Scalable Network I/O". *USENIX OSDI*. 2012.

[25] Mark Handley, Eddie Kohler, Atanu Ghosh, Orion Hodson, and Pavlin Radoslavov. "Designing extensible IP router software". *USENIX NSDI*. 2005.

[26] Mark Handley, Costin Raiciu, Alexandru Agache, Andrei Voinescu, Andrew W. Moore, Gianni Antichi, and Marcin Wójcik. "Re-Architecting Datacenter Networks and Stacks for Low Latency and High Performance". *ACM SIGCOMM*. 2017.

[27] Michio Honda, Felipe Huici, Giuseppe Lettieri, and Luigi Rizzo. "mSwitch: a highly-scalable, modular software switch". *ACM SOSR*. 2015.

[28] Michio Honda, Felipe Huici, Costin Raiciu, Joao Araujo, and Luigi Rizzo. "Rekindling network protocol innovation with user-level stacks". *ACM SIGCOMM Computer Communication Review* (2014).

[29] Michio Honda, Giuseppe Lettieri, Lars Eggert, and Douglas Santry. "PASTE: A Network Programming Interface for Non-Volatile Main Memory". *USENIX NSDI*. Apr. 2018.

[30] Wentao Hou, Jie Zhang, Zeke Wang, and Ming Liu. "Understanding Routable PCIe Performance for Composable Infrastructures". *USENIX NSDI*. Apr. 2024.

[31] Muhammad Asim Jamshed, YoungGyoun Moon, Donghwi Kim, Dongsu Han, and KyoungSoo Park. "mOS: A Reusable Networking Stack for Flow Monitoring Middleboxes". *USENIX NSDI*. 2017.

[32] Eun Young Jeong, Shinae Woo, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. "mTCP: A Highly Scalable User-level TCP Stack for Multicore Systems". *USENIX NSDI*. 2014.

[33] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. "Shinjuku: Preemptive Scheduling for {μsecond-scale} Tail Latency". *USENIX NSDI*. 2019.

[34] Anuj Kalia, Michael Kaminsky, and David Andersen. "Datacenter RPCs can be general and fast". *USENIX NSDI*. 2019.

[35] Anuj Kalia, Michael Kaminsky, and David G Andersen. "Design guidelines for high performance RDMA systems". *USENIX ATC*. 2016.

[36] Antoine Kaufmann, Tim Stamler, Simon Peter, Naveen Kr Sharma, Arvind Krishnamurthy, and Thomas Anderson. "TAS: TCP acceleration as an OS service". *ACM EuroSys*. 2019.

[37] Dan Kegel. *The C10K problem*. http://www.kegel.com/c10k.html.

[38] Taehyun Kim, Deondre Martin Ng, Junzhi Gong, Youngjin Kwon, Minlan Yu, and KyoungSoo Park. "Rearchitecting the TCP Stack for I/O-Offloaded Content Delivery". *USENIX NSDI*. 2023.

[39] Ana Klimovic, Heiner Litz, and Christos Kozyrakis. "Reflex: Remote flash $\approx$ local flash". *ACM ASPLOS*. 2017.

[40] Yohei Kuga, Ryo Nakamura, Takeshi Matsuya, and Yuji Sekiya. "NetTLP: A Development Platform for PCIe devices in Software Interacting with Hardware". *USENIX NSDI*. 2020.

[41] Ashwin Kumar, Priyanka Naik, Sahil Patki, Pranav Chaudhary, and Mythili Vutukuru. "Evaluating Network Stacks for the Virtualized Mobile Packet Core". *ACM APNet*. 2022.

[42] Jonathan Lemon. "Kqueue-A Generic and Scalable Event Notification Facility." *USENIX ATC*. 2001.

[43] Hejing Li, Jialin Li, and Antoine Kaufmann. "SimBricks: end-to-end network system evaluation with modular simulation". *ACM SIGCOMM*. 2022.

[44] Jialin Li, Jacob Nelson, Ellis Michael, Xin Jin, and Dan R. K. Ports. "Pegasus: Tolerating Skewed Workloads in Distributed Storage with In-Network Coherence Directories". *USENIX OSDI*. Nov. 2020.

[45] Ilias Marinos, Robert N.M. Watson, and Mark Handley. "Network Stack Specialization for Performance". *ACM SIGCOMM*. 2014.

[46] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C Evans, Steve Gribble, et al. "Snap: A microkernel approach to host networking". *ACM SOSP*. 2019.

[47] Matthew Mathis and Jamshid Mahdavi. "Forward acknowledgement: Refining TCP congestion control". *ACM SIGCOMM Computer Communication Review* (1996).

[48] Alberto Medina, Mark Allman, and Sally Floyd. "Measuring the evolution of transport protocols in the Internet". *ACM SIGCOMM Computer Communication Review* (2005).

[49] Jeffrey C Mogul. "The case for persistent-connection HTTP". *ACM SIGCOMM Computer Communication Review* (1995).

[50] YoungGyoun Moon, SeungEon Lee, Muhammad Asim Jamshed, and KyoungSoo Park. "AccelTCP: Accelerating network applications with stateful TCP offloading". *USENIX NSDI*. 2020.

[51] Rolf Neugebauer, Gianni Antichi, José Fernando Zazo, Yury Audzevich, Sergio López-Buedo, and Andrew W Moore. "Understanding PCIe performance for end host networking". *ACM SIGCOMM*. 2018.

[52] NLANR/DAST. *iPerf - The ultimate speed test tool for TCP, UDP and SCTP*. https://iperf.fr/.

[53] Michael F Nowlan, Nabin Tiwari, Janardhan Iyengar, Syed Obaid Amin, and Bryan Ford. "Fitting square pegs through round pipes". *USENIX NSDI*. 2012.

[54] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. "Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads." *NSDI*. 2019.

[55] John Ousterhout. "A Linux Kernel Implementation of the Homa Transport Protocol". *USENIX ATC*. Jul. 2021.

[56] Satadru Pan, Theano Stavrinos, Yunqiao Zhang, Atul Sikaria, Pavel Zakharov, Abhinav Sharma, Shiva Shankar, Mike Shuey, Richard Wareing, Monika Gangapuram, et al. "Facebook's tectonic filesystem: Efficiency from exascale". *USENIX FAST*. 2021.

[57] Federico Parola, Shixiong Qi, Anvaya B. Narappa, K. K. Ramakrishnan, and Fulvio Risso. "SURE: Secure Unikernels Make Serverless Computing Rapid and Efficient". *ACM SoCC*. 2024.

[58] Dinglan Peng, Congyu Liu, Tapti Palit, Anjo Vahldiek-Oberwagner, Mona Vij, and Pedro Fonseca. "Pegasus: Transparent and Unified Kernel-Bypass Networking for Fast Local and Remote Communication". *ACM EuroSys*. 2025.

[59] Aleksey Pesterev, Jacob Strauss, Nickolai Zeldovich, and Robert T. Morris. "Improving Network Connection Locality on Multicore Systems". *ACM EuroSys*. 2012.

[60] Simon Peter, Jialin Li, Irene Zhang, Dan RK Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. "Arrakis: The operating system is the control plane". *ACM Transactions on Computer Systems (TOCS)* (2015).

[61] Boris Pismenny, Haggai Eran, Aviad Yehezkel, Liran Liss, Adam Morrison, and Dan Tsafrir. "Autonomous NIC offloads". *ACM ASPLOS*. 2021.

[62] George Prekas, Marios Kogias, and Edouard Bugnion. "Zygos: Achieving low tail latency for microsecond-scale networked tasks". *ACM SOSP*. 2017.

[63] Sivasankar Radhakrishnan, Yuchung Cheng, Jerry Chu, Arvind Jain, and Barath Raghavan. "TCP fast open". *ACM CoNEXT*. 2011.

[64] Deepti Raghavan, Shreya Ravi, Gina Yuan, Pratiksha Thaker, Sanjari Srivastava, Micah Murray, Pedro Henrique Penna, Amy Ousterhout, Philip Levis, Matei Zaharia, and Irene Zhang. "Cornflakes: Zero-Copy Serialization for Microsecond-Scale Networking". *ACM SOSP*. 2023.

[65] Costin Raiciu, Christoph Paasch, Sebastien Barre, Alan Ford, Michio Honda, Fabien Duchene, Olivier Bonaventure, and Mark Handley. "How Hard Can It Be? Designing and Implementing a Deployable Multipath TCP". *USENIX NSDI*. 2012.

[66] Sudarsanan Rajasekaran, Sanjoli Narang, Anton A Zabreyko, and Manya Ghobadi. "MLTCP: A Distributed Technique to Approximate Centralized Flow Scheduling For Machine Learning". *ACM HotNets*. 2024.

[67] Luigi Rizzo. "netmap: A Novel Framework for Fast Packet I/O". *USENIX ATC*. 2012.

[68] Steven Rostedt. *Using KernelShark to analyze the real-time scheduler*. https://lwn.net/Articles/425583/.

[69] Henry N Schuh, Arvind Krishnamurthy, David Culler, Henry M Levy, Luigi Rizzo, Samira Khan, and Brent E Stephens. "CC-NIC: a Cache-Coherent Interface to the NIC". *ACM ASPLOS*. 2024.

[70] Korakit Seemakhupt, Brent E Stephens, Samira Khan, Sihang Liu, Hassan Wassel, Soheil Hassas Yeganeh, Alex C Snoeren, Arvind Krishnamurthy, David E Culler, and Henry M Levy. "A Cloud-Scale Characterization of Remote Procedure Calls". *ACM SOSP*. 2023.

[71] *Serving Netflix Video Traffic at 400Gb/s and Beyond*.

[72] Rajath Shashidhara, Tim Stamler, Antoine Kaufmann, and Simon Peter. "FlexTOE: Flexible TCP Offload with Fine-Grained Parallelism". *USENIX NSDI*. Apr. 2022.

[73] Brent Stephens, Arjun Singhvi, Aditya Akella, and Michael Swift. "Titan: Fair Packet Scheduling for Commodity Multiqueue NICs". *USENIX ATC*. 2017.

[74] Brent E. Stephens, Darius Grassi, Hamidreza Almasi, Tao Ji, Balajee Vamanan, and Aditya Akella. "TCP is Harmful to In-Network Computing: Designing a Message Transport Protocol (MTP)". *ACM HotNets*. 2021.

[75] Matheus Stolet, Liam Arzola, Simon Peter, and Antoine Kaufmann. "Virtuoso: High Resource Utilization and μs-scale Performance Isolation in a Shared Virtual Machine TCP Network Stack". *arXiv preprint arXiv:2309.14016* (2023).

[76] Hajime Tazaki, Ryo Nakamura, and Yuji Sekiya. *Library operating system with mainline Linux network stack*. Proceedings of netdev 0.1, https://netdevconf.org/0.1/papers/Library-Operating-System-with-Mainline-Linux-Network-Stack.pdf. 2015.

[77] *Transmission Control Protocol*. RFC 793. Sep. 1981. URL: https://www.rfc-editor.org/info/rfc793.

[78] Kenichi Yasukata. "iip: an integratable TCP/IP stack". *ACM SIGCOMM Computer Communication Review* (2024).

[79] Kenichi Yasukata, Michio Honda, Douglas Santry, and Lars Eggert. "StackMap: Low-Latency Networking with the OS Stack and Dedicated NICs". *USENIX ATC*. 2016.

[80] Irene Zhang, Amanda Raybuck, Pratyush Patel, Kirk Olynyk, Jacob Nelson, Omar S Navarro Leija, Ashlie Martinez, Jing Liu, Anna Kornfeld Simpson, Sujay Jayakar, et al. "The demikernel datapath os architecture for microsecond-scale datacenter systems". *ACM SOSP*. 2021.

[81] WL Zhang, YF Shen, H Song, Zh Zhang, K Liu, Q Huang, and MY Chen. "QStack: Re-architecting User-space Network Stack to Optimize CPU Efficiency and Service Quality". *arXiv preprint arXiv:2210.08432* (2022).

[82] Lingjun Zhu, Yifan Shen, Erci Xu, Bo Shi, Ting Fu, Shu Ma, Shuguang Chen, Zhongyu Wang, Haonan Wu, Xingyu Liao, et al. "Deploying User-space TCP at Cloud Scale with LUNA". *USENIX ATC*. 2023.

[83] Yong-Hao Zou, Jia-Ju Bai, Jielong Zhou, Jianfeng Tan, Chenggang Qin, and Shi-Min Hu. "TCP-Fuzz: Detecting memory and semantic bugs in TCP stacks with fuzzing". *USENIX ATC*. 2021.

GPREEMPT: GPU Preemptive Scheduling Made General and Efficient

Ruwen Fan[†] Tingxu Ren[†] Minhui Xie[‡] Shiwei Gao[†] Jiwu Shu[†] Youyou Lu[†*]

[†]*Department of Computer Science and Technology, BNRist, Tsinghua University*
[‡]*Renmin University of China*

Abstract

GPUs support various workloads with different peak periods and diverse service level agreements (SLA) requirements, including latency-critical tasks and best-effort tasks. Co-locating tasks with diverse SLA demands can enhance resource utilization, yet it introduces the risk of performance interference. Prior work employs preemption strategies to enforce SLAs for latency-critical tasks. These strategies can be classified into two categories: wait-based and reset-based approaches. The wait-based strategy ensures broad generality but suffers from significant preemption latency. In contrast, the reset-based strategy necessitates the idempotence of preempted kernels, limiting its generality.

This paper presents GPREEMPT, a preemption mechanism that breaks the trade-off. GPREEMPT implements a *timeslice-based yield mechanism* to facilitate context-switch preemption on GPUs. To mitigate the overhead associated with context-switching, GPREEMPT employs a *hint-based pre-preemption* technique to overlap the preemption process with the essential data-preparation phase. Our evaluation demonstrates that GPREEMPT achieves within 40 µs low-latency preemption comparable to executing only latency-critical tasks while remaining applicable to non-idempotent workloads, where reset-based mechanisms prove inadequate.

1 Introduction

GPUs have become essential for a wide range of computing tasks, including computer vision [15, 34], machine learning [14, 22], graphic rendering [5, 27], and scientific computing [23, 30]. Their large-scale parallelism provides exceptional computing throughput, making them a preferred choice in both enterprise and research environments.

These above GPU workloads often experience fluctuating peak traffic periods [9, 32]. Statically allocating dedicated GPU resources to a single task leads to under-utilization during low-traffic periods, resulting in significant resource wastage. Furthermore, GPU tasks typically exhibit varying Service Level Agreement (SLA) requirements [11, 36], including latency-critical (LC) applications (e.g., real-time recommendation, autonomous driving and virtual reality) and best-effort (BE) workloads (such as offline inference and data analytics). A common practice [12, 18, 29] is to co-locate multiple tasks with different SLA requirements on shared GPU resources to improve GPU utilization. However, co-locating also introduces a significant challenge: When LC tasks arrive, BE tasks may still occupy the GPU, interfering with the execution of LC tasks, and potentially violating SLA guarantees.

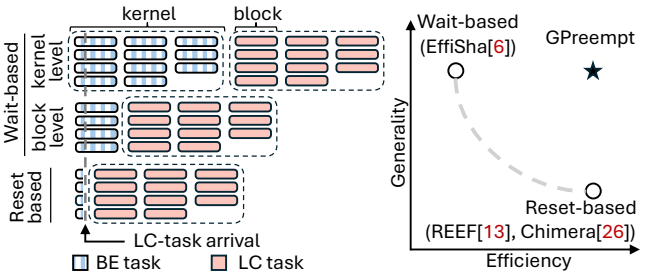


Figure 1: Existing Preemption Mechanisms and Trade-off of Them. A GPU task typically consists of multiple kernels, where each kernel is made up of several thread blocks.

Previous studies [6, 13, 17, 26] have proposed several preemption approaches to ensure the execution latency of LC tasks. However, these approaches usually make a trade-off between *efficiency* and *generality*. As Figure 1 shows, these works can be classified into two categories: wait-based and reset-based preemption strategies. Wait-based preemption, such as block-level preemption [6, 38], passively waits for the running GPU thread block to finish. These preemption methods are highly general. However, they operate at a coarse granularity, introducing a long waiting latency (e.g., up to 5 ms in block-level preemption, see §2.3 for detail). Reset-based preemption strategies [13, 26] aggressively terminate running kernels to facilitate rapid task preemption, capitalizing on the idempotency of certain DNN kernels. However,

*Corresponding author: Youyou Lu (luyouyou@tsinghua.edu.cn)

this idempotency requirement limits their applicability to non-idempotent tasks (such as graph computing and scientific computing) and modern graph kernel launch techniques (e.g., CUDA graph [8]), restricting their generality.

We aim to introduce a switch-based preemption mechanism to break the trade-off between generality and low-latency preemption. Drawing inspiration from CPU scheduling, context-switch provides a fine granularity and general preemption approach (e.g., coroutine), which is widely adopted in CPU task scheduling. Directly applying such an approach to GPUs, however, is non-trivial: (1) GPU scheduling is opaque to users, and no scheduling interface is provided. Once a kernel is launched, users are unable to *yield* the GPU computing resources allocated to that kernel; (2) The large GPU context (tens of MB) can introduce significant switching overheads.

To overcome the challenges, we designed GPREEMPT, a general-purpose GPU preemptive scheduling mechanism. To enable the *yield* primitive, GPREEMPT adopts a *timeslice-based* approach. Through a thorough analysis of the GPU open-source driver and the GPU hardware scheduling mechanism, we identified and repurposed an internal timeslice allocation mechanism in the GPU driver as the foundation for implementing the *yield* primitive, abstracting it into a general interface that enables fine-grained control over task scheduling. GPU cycles through tasks according to the timeslice of each task. By reducing the timeslice for low-priority tasks, these tasks can yield GPU resources in a short duration, making user-controlled context-switch feasible on the GPU. To mitigate the cost of context-switching, GPREEMPT integrates *Hint-based Pre-preemption*. We exploit the data-preparation phase – an unavoidable step before launching GPU kernels. During this interval, GPREEMPT can initiate the preemption procedure, thereby hiding much of the switching overhead.

We evaluate GPREEMPT on 7 workloads (including DNN, graph computing and scientific computing) on NVIDIA and AMD GPUs. Evaluation shows that GPREEMPT achieves comparable latency and throughput to reset-based methods in idempotent kernel preemption scenarios, with less than 40 µs preemption latency on the NVIDIA platform. Moreover, in non-idempotent scenarios where reset-based methods are inapplicable, our system can still achieve preemption with a latency of only 30% compared to wait-based approaches.

2 Background and Motivation

2.1 GPU Computing Tasks

Offload Computing Task to GPUs. Offloading a computing task to GPUs often involves three procedures: Data preparation, where input data is transformed into a format suitable for GPU computing; data transfer, which involves moving the processed data from the host memory to the GPU memory; and kernel execution, where the actual computing takes place on the GPU. A GPU task often consists of multiple computing kernels, which collectively require execution times ranging from a few milliseconds to several tens of milliseconds [20]. Modern GPU computing frameworks utilize CUDA graphs [8] (and AMD HIP graphs [3]) to reduce the kernel launch overhead. These graph launch approaches consolidate multiple GPU kernels into a directed structure that defines execution order and dependencies, allowing the runtime to treat the kernel sequence as a single executable unit to avoid launch overhead.

Characteristics of GPU Computing Tasks. GPU computing tasks have demonstrated remarkable versatility across diverse application domains with varying latency requirements. In latency-critical scenarios like autonomous vehicle control and real-time robotic manipulation, GPU effectively processes sensor data to make split-second decisions. Conversely, in best-effort applications like offline inference, medical image analysis and climate modeling, GPUs excel at complex pattern recognition tasks where computing time is less constrained. For example, both OpenAI and Amazon provide asynchronous batch APIs [25, 31], which will give users responses within 24 hours.

Co-locate Multiple Computing Tasks on GPU. Running multiple computing tasks on a single GPU is a prevalent demand [12, 18, 29]. Modern data center inference clusters employ strategic task co-location to maximize GPU resource utilization. The GPU workload typically fluctuates dynamically, and the gaps between inference tasks can easily lead to idle GPU resources [18]. Modern data centers typically interleave best-effort tasks with latency-critical workloads.

While this co-location strategy enhances GPU resource efficiency, it introduces complex scheduling challenges. The primary challenge lies in simultaneously satisfying the stringent SLAs of latency-critical tasks while optimizing the throughput of best-effort workloads, given their heterogeneous latency requirements and resource consumption patterns.

2.2 Existing GPU Preemptive Scheduling

With the emerging demand for running multiple tasks on GPUs, GPU scheduling [6, 13, 17, 26] becomes a widely studied research topic. Existing preemption approaches can be primarily categorized into two types: wait-based and reset-based methods. These approaches make a trade-off between general-purpose preemption and fast preemption.

Wait-based Preemption. Prior work [6, 10, 38] introduced wait-based preemption mechanisms that enforce exclusive GPU access for LC tasks by waiting for BE tasks to complete their execution. It is important to note that 'preemption' in this context often carries a different meaning than in operating systems; here, the emphasis is on achieving an effect similar to preemption (e.g., LC tasks can be executed with minimal delay), rather than an identical preemption process. These approaches operate at kernel or block boundaries. For example, SRM [10] treats the GPU as a globally shared resource

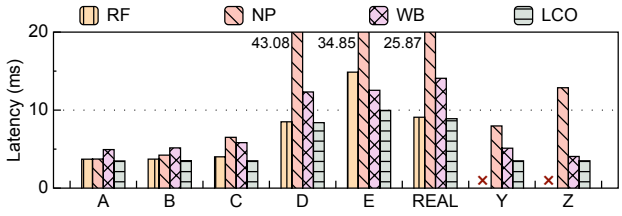


Figure 2: End-to-end Latency of LC Tasks. RF: REEF (reset-based), NP: No preemption, WB: Block-level preemption (wait-based), LCO: Execution LC task only.

protected by a semaphore, executing each kernel sequentially. Effisha [6] relinquishes GPU resources by transforming each computation kernel to check at the block boundary if any LC tasks are arriving. The LC task still needs to wait for the running thread block to complete execution. Wait-based preemption approaches are prone to long waiting times since LC tasks may still be blocked by running tasks.

Reset-based Preemption. Prior research efforts [13,26] have introduced reset-based methodologies for GPU task preemption, leveraging the idempotent characteristics inherent in specific DNN operators. These approaches enable immediate kernel termination to liberate GPU computing resources, facilitating the expedited execution of LC workloads. The idempotent nature of these operations ensures that BE tasks can be recomputed during periods of GPU inactivity without compromising computing correctness or output integrity. Nevertheless, this methodology exhibits fundamental limitations tied to the prerequisite of kernel idempotency.

2.3 Motivation

While existing GPU preemption approaches have made important strides in addressing scheduling challenges, they continue to face fundamental limitations in either generality or performance. Figure 2 illustrates the end-to-end latency of block-level wait-based and reset-based approaches. While the block-level preemption strategy significantly reduced preemption latency (reduced 30 ms on workload D) compared to a no-preemption baseline, the overhead associated with task preemption remains considerable (e.g., a 66.4% (5 ms) increase in latency for LC tasks under workload C). In workload C, the large number of BE tasks means multiple may be running when LC arrives. Although block-level preemption allows quick exits for new blocks, ongoing ones must finish, causing stalls. Moreover, LC is blocked by the slowest BE task, leading to longer average latency than in workloads with fewer BE tasks (e.g., A, B). The reset-based approach, by contrast, demonstrates lower latency. However, the idempotency constraints of kernels significantly limit the applicability of this approach. For instance, in domains such as scientific computing (Y) and graph computing (Z), the global state changes during execution, rendering the process non-idempotent. In these cases, the reset-based preemption strategy is not feasible. Additionally, the integration of advanced kernel launch mechanisms, particularly CUDA Graph, introduces further complexities in maintaining idempotent properties, which can substantially increase the overhead of reset operations.

Key Idea. Motivated by CPU scheduling, we propose a switch-based preemption mechanism to address the trade-off between generality and low-latency preemption. Context-switching is a well-established preemption technique in CPU scheduling, which ensures task transparency and fine preemption granularity. However, directly adapting context-switching to GPU scheduling poses significant challenges:

- *GPU scheduling is opaque to users.* Unlike CPU-based systems, GPUs lack a user-facing context-switch interface, making resource preemption more complex.
- *High context-switch overhead.* GPU task contexts are much larger than CPU (44 MB in NVIDIA A100 GPU vs. less than 1 KB in x86 CPU), and the overhead associated with context switching is considerable. To mitigate these challenges, a meticulously designed preemption strategy is essential to minimize the impact of this overhead.

3 GPREEMPT Design

To achieve both generality and low-overhead GPU preemption scheduling, we present GPREEMPT, a general and efficient scheduling mechanism. GPREEMPT introduces a *timeslice-based general preemption* (§3.1) to provide a fine-granularity and transparent context-switch preemption mechanism. By leveraging the common GPU computing paradigm, GPREEMPT further employs an innovative *hint-based pre-preemption* technique (§3.2), which strategically overlaps preemption operations with non-computation phases, thereby minimizing observable preemption latency.

3.1 Timeslice-based General Preemption

To facilitate rapid preemption, a fine-grained preemption mechanism is essential. While a reset-based approach can seize GPU resources at any moment, it imposes significant constraints on computational kernels. We aim to enable switch-based preemption in the GPU. Through a thorough analysis of NVIDIA open GPU kernel source [24], we discover an undocumented and little-known hardware resource allocation mechanism: timeslice allocation. GPREEMPT fully exploits the power of timeslice allocation, indirectly implementing *yield* primitive in GPU through timeslice rotation, thus enabling switch-based preemption.

Timeslice-based Yield Mechanism. When multiple independent tasks or tenants run on a GPU, the hardware cycles through them based on predetermined timeslices, ensuring fair computing time for each task. Leveraging this hardware cycling mechanism and timeslice reconfiguration, GPREEMPT

implements a passive *yield* mechanism. Specifically, GPREEMPT adjusts the timeslice of a BE task to a very short duration (e.g., 200µs) to ensure that when LC tasks arrive, BE tasks will consume the time quota quickly and *yield* GPU resources within a time frame. Correspondingly, GPREEMPT extends the timeslice for LC tasks longer than the lifespan of the LC task, ensuring that LC task execution will not be interrupted by other tasks. This approach is effective even without knowing the LC task's lifecycle in advance; thus, setting the timeslice to a very long duration (e.g., even a few seconds) is fine. When a LC task finishes, its remaining timeslice is automatically dropped, and the GPU immediately switches to the next task. This configuration is set up during system initialization, and no need to change afterwards. Although the timeslice for BE tasks is reduced, their throughput remains unaffected when no LC tasks launch. Because BE tasks are placed within the same scheduling group. When only BE tasks are present, the GPU does not forcibly switch between them, avoiding unnecessary overhead. BE tasks run uninterrupted until completion or until a LC task preempts them.

Scheduling Overhead. Scheduling overhead includes two parts, time for BE to *yield* and context-switching. According to the GPU hardware scheduling mechanism, a task relinquishes the GPU after consuming its timeslice. Let t_1 denote the timeslice for BE tasks. When a LC task arrives, the BE task will yield GPU resources within an average time of $\frac{t_1}{2}$ and a maximum time of t_1, which is about 160 µs. Context-switching overhead is fundamentally bounded by GPU context size and memory bandwidth parameters, representing a fixed cost for each GPU architecture. This overhead continues to diminish in significance as GPU memory bandwidth capabilities advance. Take NVIDIA A100 GPU as an example: each of its 108 Streaming Multiprocessors (SMs) contains 164 KB of shared memory and 64 K 32-bit registers, resulting in a per-SM context footprint of 420 KB and an aggregate GPU context of approximately 44.3 MB [1]. With the memory bandwidth of 1.1 TB/s, context-saving is completed within 40 µs. Our empirical evaluation reveals a combined overhead of approximately 100 µs (§5.3) for these operations. Additionally, we introduce a pre-preemption technique in §3.2 to further minimize preemption latency.

3.2 Hint-based Pre-preemption

Data preparation is a prerequisite for GPU computing tasks before kernel execution, as illustrated in Figure 3(a). Many tasks involve data preprocessing (such as image crop and data normalization), which can extend to hundreds of µs, and all tasks necessitate data transfer from the CPU to the GPU. Even minimal data transfers via `cudaMemcpy` introduce an overhead of approximately 80 µs [16]. **These preparatory stages offer predictive signals for the imminent arrival of LC kernels, facilitating proactive resource reservation through pre-preemption.** Typically, data preparation is executed by the

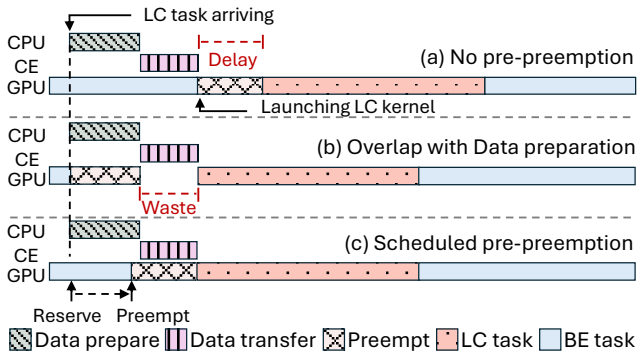


Figure 3: Hint-based Pre-preemption Example. CE: GPU copy engine.

CPU and GPU copy engines, operating independently of the GPU computing engines. In a GPU, the computing engine and copy engine can be occupied by different tasks. They will not block each other [4]. Leveraging these characteristics, GPREEMPT introduces a hint-based pre-preemption mechanism that enables LC tasks to reserve GPU computing resources in advance, seamlessly overlapping task-switching overhead with ongoing data preparation.

Overlap Preemption with Data preparation. GPREEMPT enables users to issue GPU reservation commands when LC tasks are anticipated. As shown in Figure 3(b), upon receiving such commands, GPREEMPT strategically injects a preemption kernel that consumes the timeslice of the best-effort task at a predetermined instant, concurrent with preprocessing operations. This preemption kernel orchestrates a controlled wait state until the corresponding computing kernel is ready for execution. Leveraging GPU stream semantics, the computing kernel seamlessly initiates execution upon the preemption kernel's completion. GPREEMPT leverages GDRCopy [33], which enables direct CPU-GPU memory access by mapping GPU HBM to the CPU address space, to notify the preemption kernel, achieving remarkably low latency to finish the preemption kernel (approximately 1 µs). This pre-preemption strategy effectively masks context-switching overhead by overlapping it with essential data preprocessing steps, minimizing end-to-end latency.

Scheduled Pre-preemption. Preempting GPU resources too far in advance may result in tasks remaining in the preprocessing phase for extended periods even after GPUs become available. Benefiting from §3.1, GPREEMPT provides a deterministic preemption latency. Leveraging this, GPREEMPT implements a scheduled reservation mechanism whereby users can specify their anticipated GPU resource requirements when submitting a preemption request. GPREEMPT launches the preemption kernel through a background thread based on the scheduled timing. As Figure 3(c), this avoids GPU resource waste while providing pre-preemption support.

Table 1: Workload configuration of LC and BE tasks

WL	LC Tasks	LC Rate	BE Tasks	BE Rate
A	VGG	100 (C)	ResNet	100 (C)
B	VGG	220 (C)	ResNet	220 (C)
C	VGG	100 (C)	DM*	100 (C)
D	DM*	20 (C)	DM*	100 (C)
E	DM*	20 (P)	DM*	100 (P)
REAL	DM*	Trace	DM*	Trace
Y	VGG	100 (C)	WS	150 (C)
Z	VGG	100 (C)	GM**	100 (C)

DM: {ResNet, DenseNet, VGG, Inception, BERT}
GM: {BFS, SSSP, PageRank, CC}.
(C): Constant rate, (P): Poisson rate. "100 (C)": 100 requests/sec per task.

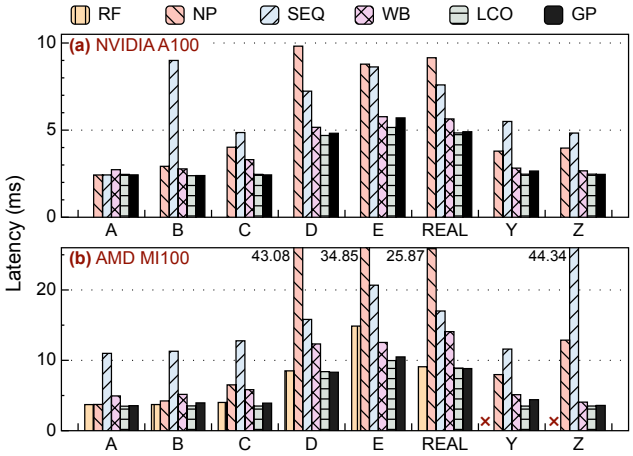


Figure 4: End-to-end Latency of LC Tasks. NP: no preemption, SEQ: sequential execution (wait-based), WB: block-level preemption (wait-based), LCO: latency-critical tasks only, RF: REEF (reset-based), GP: GPREEMPT.

4 Implementation

GPREEMPT consists of two primary components: 1) The GPU driver code modifications that enable necessary low-level functionality to support timeslice reconfiguration. 2) Building upon these driver modifications, GPREEMPT provides ready-to-use APIs that can be directly integrated into user programs without modifying the computing kernel code.

GPREEMPT on AMD GPUs. AMD has introduced the MES hardware scheduler since RDNA 3 [2]. Similar to NVIDIA, MES uses timeslices to cycle through tasks by hardware, in which the driver dedicates the timeslice for each compute context. GPREEMPT can implement this preemptive scheduling in a similar way to how it is implemented on NVIDIA GPUs. For earlier AMD GPUs, we implemented GPREEMPT by modifying the ROCm driver code and repurposing its debugging mechanism, using it to switch between contexts manually.

5 Evaluation

5.1 Evaluation Setup

Setups. Experiments are conducted on a system equipped with an Intel(R) Xeon(R) Gold 5420+ CPU, 256 GB DRAM, and an NVIDIA A100-40GB GPU. An AMD-based version of GPREEMPT was evaluated on a platform with identical CPU, memory and an AMD Instinct MI100 GPU.

Comparing Targets. On the NVIDIA platform, we compared GPREEMPT with a LC-task-only baseline (representing the ideal latency boundary), two wait-based preemption approaches (sequential execution and block-level preemption), and a no-preemption baseline (using multiple streams to maximize task parallelism, representing ideal GPU utilization). On the AMD platform, we included an additional baseline, a reset-based preemption approach (REEF [13], without dynamic kernel padding, constrained by the open-source implementation capability with specific AMD GPUs, which will not affect the preemption latency).

Workloads. As detailed in Table 1, we evaluate GPREEMPT using DISB [13] and two synthetic workloads. DISB contains 6 typical DNN inference workloads, including LC-task with uniform distribution (A, B, C, D), poison distribution (E) and real-world trace (REAL). Synthetic workloads include scientific computing [23] (Y), and graph computing tasks [21] (Z), which are co-located with LC DNN inference tasks.

5.2 Overall Performance

LC-task Latency. Figure 4 shows the end-to-end latency of LC tasks on NVIDIA A100 (4(a)) and AMD MI100 (4(b)) GPU. Reset-based approach (RF) only supports the AMD platform and idempotent workloads.

On the NVIDIA platform, compared to executing LC tasks only (LCO), no preemption (NP), sequential execution (SEQ), block-level preemption (WB), and GPREEMPT (GP) introduce average latency increases of 58.4%, 96.3%, 15.3%, and 2.4%, respectively. On the AMD platform, compared to LC, NP, SEQ, WB, reset-based preemption (RF) and GPREEMPT introduce average latency increases of 170.1%, 297.9%, 43.5%, 13.2%, and 10%, respectively.

In scenarios with high task arrival frequencies (B, E, Z), sequential execution methods are prone to resource monopolization by BE tasks, leading to the highest latency. Additionally, sequential execution fails to exploit the parallelism between tasks, further exacerbating the queuing delay. For block-level preemption, in scenarios with numerous BE tasks (C, REAL), preemption latency is influenced by the slowest BE task, resulting in higher average latency compared to scenarios with fewer BE tasks. The reset-based method achieves low preemption latency for DNN workloads (A-E, REAL) that consist entirely of idempotent kernels. However, it is un-

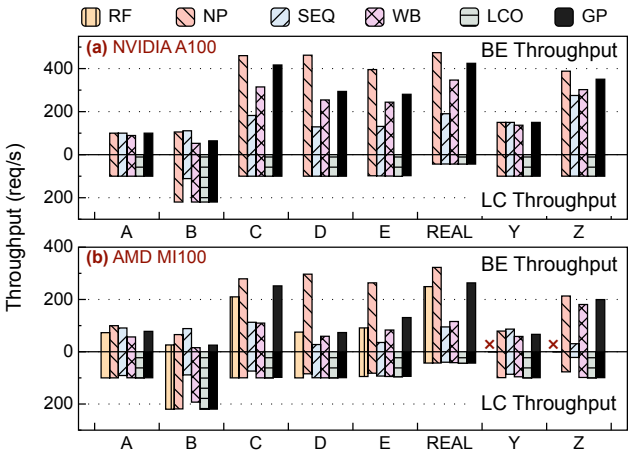


Figure 5: LC and BE Task Throughput on Various Workloads. Above the x-axis is the BE tasks throughput, and below the x-axis is the LC tasks throughput.

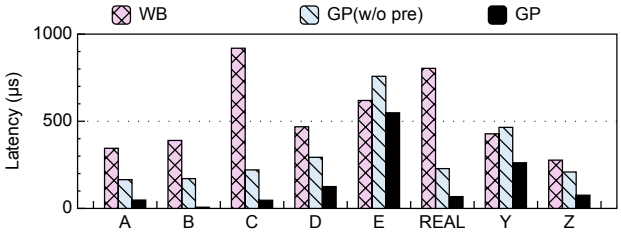


Figure 6: Additional Latency Compared to LC Task Only. GP(w/o pre): GPREEMPT without pre-preemption.

suitable for workloads Y and Z, including non-idempotent kernels. Under all workloads, GPREEMPT achieves near-ideal preemption latency while still maintaining generality.

GPU Utilization. Figure 5 illustrates the total throughput across different preemption strategies on NVIDIA (5(a)) and AMD (5(b)) GPUs. Executing all tasks without preemption (NP) achieves the highest overall throughput in all scenarios, as GPU parallelism is fully exploited when task preemption is absent. We use NP as the baseline to calculate the relative throughput of other methods.

On the NVIDIA GPU, SEQ, WB, LC, and GPREEMPT achieve throughputs of 66.0%, 79.7%, 30.3%, and 88.6%, respectively. On the AMD GPU, RF, SEQ, WB, LC, and GPREEMPT achieve throughputs of 72.4%, 53.1%, 65.7%, 38.8%, and 82.2%, respectively. GPREEMPT guarantees that the throughput of LC tasks remains unaffected across all workloads while striving to keep the throughput of BE tasks close to the ideal scenario. This demonstrates the efficient utilization of GPU resources by GPREEMPT in co-located LC and BE task scenarios. The throughput when executing only LC tasks is the lowest, as only LC tasks are unable to fully utilize GPU resources, leading to considerable idle time. While the Sequential execution strategy provides some throughput improvement, the throughput of LC tasks decreases in this configuration. This occurs because LC tasks are likely to be blocked by BE tasks, leading to high latencies. When using block-level preemption, additional checkpoints are inserted into the BE kernels to determine whether they should exit, decreasing the performance of BE tasks.

5.3 Technical Analysis

Performance Breakdown. Figure 6 shows the additional latency for different workloads. Comparing the block-level wait-based preemption, GPREEMPT without hint-based pre-preemption (GP (w/o pre)), and GPREEMPT. In most cases, enabling only switch-based preemption achieves a preemption latency with an average of just 200 µs. Only in complicated workloads like E and Y, GPREEMPT without pre-preemption shows relatively high preemption latency. After adopting the pre-preemption technique, GPREEMPT consistently achieves the lowest latency across all workloads. For single-LC-task workloads (e.g., A, B, C, Z), the average preemption latency of GPREEMPT is less than 40 µs. In multiple LC and BE tasks co-locate scenario (workload E, 10 concurrent tasks in total), the average latency increase introduced by GPREEMPT is only about 500 µs. The hint-based pre-preemption mechanism results in an average latency reduction of 160 µs.

Analysis of Hint-based Pre-preemption. As introduced in §3.2, GPREEMPT utilizes the data preparation phase of LC tasks to overlap with preemption to mitigate the preemption latency and leverage scheduled pre-preemption to further decrease the preemption influence on BE throughput. Using workload A, we varied both the data preparation time and the arrival frequency of the LC tasks to assess their effects on additional latency and BE task throughput.

As shown in Figure 7(a), the additional latency introduced by preemption decreases significantly as the data preparation time increases. When there is no data preparation time, GPREEMPT incurs an additional latency of approximately 100 µs. However, when the data preparation time exceeds 100 µs, the additional latency stabilizes to below 40 µs. The latency is mainly caused by cache interference when colocating tasks, which is negligible compared to the duration of LC tasks, which typically span several milliseconds [20].

Figure 7(b) highlights the impact of LC task frequency on BE task throughput. Compared with the block-level wait-based preemption approach, GPREEMPT improved BE task throughput by 17%-26%. To validate the effectiveness of scheduled pre-preemption, we compare GPREEMPT with a version (GP(w/o sched)) that removes this technique. This optimization results in a throughput improvement of 4%-10% across varying LC task frequencies.

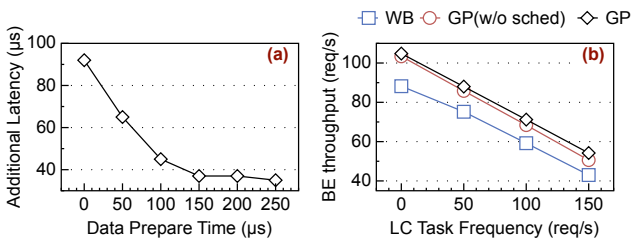


Figure 7: Analysis of Hint-based Pre-preemption: (a) Additional latency on various data prepare time and (b) BE tasks throughput on different LC task frequency. GP (w/o sched): GPREEMPT without scheduled pre-preemption.

6 Related Works

GPU Task Preemption. Previous studies [19, 35] have proposed extending GPUs to provide preemption mechanisms to reduce the context switch overhead and enable user-initiated preemption, but these were only implemented on GPU simulators. Such approaches require modifications to hardware logic, making them difficult for manufacturers to integrate into real-world GPUs. Some works [17, 37] also propose software-based preemption mechanisms by inserting checkpoints into the application to detect the arrival of high-priority tasks. Software checkpoints incur additional overhead and fail to achieve timely preemption. In contrast, GPREEMPT retrofitted the GPU internal timeslice allocation into a passive yield mechanism to achieve task preemption. GPREEMPT requires no hardware mechanism modification, making it directly applicable to commercial GPUs.

Multi-task GPU Co-location Strategies. Significant research efforts have been directed toward optimizing GPU resource utilization through strategic task co-location. DeepBoot [7] implements an adaptive scheduling mechanism that dynamically alternates between training and inference tasks, effectively recycling GPU resources. The DAG-based Scheduler [28] employs directed acyclic graphs to automatically determine inter-task data dependencies, eliminating the need for manual dependency specification and facilitating optimal parallel task execution through comprehensive resource utilization analysis. Differently, GPREEMPT focuses on preemption which is one of the foundations of multi-task scheduling.

7 Conclusion

This paper introduces GPREEMPT, a general-purpose priority scheduling approach for GPU. Unlike previous wait-based and reset-based preemption mechanisms, GPREEMPT employs a context switch-based strategy to deliver efficient and transparent preemption support on both NVIDIA and AMD GPUs. Evaluations demonstrate that GPREEMPT effectively preempts GPU resources for LC tasks while striving to keep the throughput of BE tasks across various scenarios.

Acknowledgments

We sincerely thank our shepherd Kostis Kaffes for helping us improve the paper. We also thank the anonymous reviewers for their valuable feedback. This work is supported by the National Key R&D Program of China (Grant No. 2024YFE0203300), the National Natural Science Foundation of China (Grant No. 62332011, U22B2023), and Beijing Natural Science Foundation (Grant No. L242016).

References

[1] CUDA C++ Programming Guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide/, 2025.

[2] Advanced Micro Devices. Micro Engine Scheduler (MES). https://gpuopen.com/download/documentation/micro_engine_scheduler.pdf, 2024.

[3] AMD. Graph management. https://rocm.docs.amd.com/projects/HIP/en/docs-develop/reference/hip_runtime_api/modules/graph_management.html, 2024.

[4] Joshua Bakita and James H. Anderson. Hardware compute partitioning on NVIDIA gpus. In *29th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2023, San Antonio, TX, USA, May 9-12, 2023*, pages 54–66. IEEE, 2023.

[5] Pieterjan Bartels and Takahiro Harada. Combining GPU tracing methods within a single ray query. In Soon Ki Jung and Neil A. Dodgson, editors, *SIGGRAPH Asia 2022 Technical Communications, Daegu, Republic of Korea, December 6-9, 2022*, pages 17:1–17:4. ACM, 2022.

[6] Guoyang Chen, Yue Zhao, Xipeng Shen, and Huiyang Zhou. Effisha: A software framework for enabling effficient preemptive scheduling of GPU. In Vivek Sarkar and Lawrence Rauchwerger, editors, *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Austin, TX, USA, February 4-8, 2017*, pages 3–16. ACM, 2017.

[7] Zhenqian Chen, Xinkui Zhao, Chen Zhi, and Jianwei Yin. Deepboot: Dynamic scheduling system for training and inference deep learning tasks in GPU cluster. *IEEE Trans. Parallel Distributed Syst.*, 34(9):2553–2567, 2023.

[8] NVIDIA Corporation. Getting started with cuda graphs. https://developer.nvidia.com/blog/cuda-graphs/, 2019.

[9] Jiangfei Duan, Runyu Lu, Haojie Duanmu, Xiuhong Li, Xingcheng Zhang, Dahua Lin, Ion Stoica, and Hao Zhang. Muxserve: Flexible spatial-temporal multiplexing for multiple LLM serving. In *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*. OpenReview.net, 2024.

[10] Glenn A. Elliott and James H. Anderson. Globally scheduled real-time multiprocessor systems with gpus. *Real Time Syst.*, 48(1):34–74, 2012.

[11] Haibing Guan, Jianguo Yao, Zhengwei Qi, and Runze Wang. Energy-efficient SLA guarantees for virtualized GPU in cloud gaming. *IEEE Trans. Parallel Distributed Syst.*, 26(9):2434–2443, 2015.

[12] Bing-Shiun Han, Tathagata Paul, Zhenhua Liu, and Anshul Gandhi. KACE: kernel-aware colocation for efficient GPU spatial sharing. In *Proceedings of the 2024 ACM Symposium on Cloud Computing, SoCC 2024, Redmond, WA, USA, November 20-22, 2024*, pages 460–469. ACM, 2024.

[13] Mingcong Han, Hanze Zhang, Rong Chen, and Haibo Chen. Microsecond-scale preemption for concurrent gpu-accelerated DNN inferences. In Marcos K. Aguilera and Hakim Weatherspoon, editors, *16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2022, Carlsbad, CA, USA, July 11-13, 2022*, pages 539–558. USENIX Association, 2022.

[14] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*, pages 770–778. IEEE Computer Society, 2016.

[15] Gao Huang, Zhuang Liu, and Kilian Q. Weinberger. Densely connected convolutional networks. *CoRR*, abs/1608.06993, 2016.

[16] Changho Hwang, KyoungSoo Park, Ran Shu, Xinyuan Qu, Peng Cheng, and Yongqiang Xiong. ARK: gpu-driven code execution for distributed deep learning. In Mahesh Balakrishnan and Manya Ghobadi, editors, *20th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2023, Boston, MA, April 17-19, 2023*, pages 87–101. USENIX Association, 2023.

[17] Zhuoran Ji and Cho-Li Wang. Compiler-directed incremental checkpointing for low latency GPU preemption. In *2022 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2022, Lyon, France, May 30 - June 3, 2022*, pages 751–761. IEEE, 2022.

[18] Jiamin Li, Hong Xu, Yibo Zhu, Zherui Liu, Chuanxiong Guo, and Cong Wang. Aryl: An elastic cluster scheduler for deep learning. *CoRR*, abs/2202.07896, 2022.

[19] Zhen Lin, Lars Nyland, and Huiyang Zhou. Enabling efficient preemption for SIMT architectures with lightweight context switching. In John West and Cherri M. Pancake, editors, *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2016, Salt Lake City, UT, USA, November 13-18, 2016*, pages 898–908. IEEE Computer Society, 2016.

[20] Liangkai Liu, Yanzhi Wang, and Weisong Shi. Understanding time variations of DNN inference in autonomous driving. *CoRR*, abs/2209.05487, 2022.

[21] Seungwon Min, Vikram Sharma Mailthody, Zaid Qureshi, Jinjun Xiong, Eiman Ebrahimi, and Wen-Mei Hwu. EMOGI: efficient memory-access for out-of-memory graph-traversal in gpus. *Proc. VLDB Endow.*, 14(2):114–127, 2020.

[22] Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, Xiaodong Wang, Udit Gupta, Carole-Jean Wu, Alisson G. Azzolini, Dmytro Dzhulgakov, Andrey Mallevich, Ilia Cherniavskii, Yinghai Lu, Raghuraman Krishnamoorthi, Ansha Yu, Volodymyr Kondratenko, Stephanie Pereira, Xianjie Chen, Wenlin Chen, Vijay Rao, Bill Jia, Liang Xiong, and Misha Smelyanskiy. Deep learning recommendation model for personalization and recommendation systems. *CoRR*, abs/1906.00091, 2019.

[23] Matthew R. Norman. miniweather: A mini-app to mimic atmospheric weather models on modern hardware architectures. https://github.com/mrnorman/miniWeather.

[24] NVIDIA. Open gpu kernel modules. https://github.com/NVIDIA/open-gpu-kernel-modules, 2021.

[25] OpenAI. Batch API. https://platform.openai.com/docs/guides/batch, 2024.

[26] Jason Jong Kyu Park, Yongjun Park, and Scott A. Mahlke. Chimera: Collaborative preemption for multitasking on a shared GPU. In Özcan Özturk, Kemal Ebcioglu, and Sandhya Dwarkadas, editors, *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2015, Istanbul, Turkey, March 14-18, 2015*, pages 593–606. ACM, 2015.

[27] Steven G. Parker, James Bigler, Andreas Dietrich, Heiko Friedrich, Jared Hoberock, David P. Luebke, David K. McAllister, Morgan McGuire, R. Keith Morley, Austin Robison, and Martin Stich. Optix: a general purpose ray tracing engine. *ACM Trans. Graph.*, 29(4):66:1–66:13, 2010.

[28] Alberto Parravicini, Arnaud Delamare, Marco Arnaboldi, and Marco D. Santambrogio. Dag-based scheduling with resource sharing for multi-task applications in a polyglot GPU runtime. In *35th IEEE International Parallel and Distributed Processing Symposium, IPDPS 2021, Portland, OR, USA, May 17-21, 2021*, pages 111–120. IEEE, 2021.

[29] Vignesh T. Ravi, Michela Becchi, Gagan Agrawal, and Srimat T. Chakradhar. Supporting GPU sharing in cloud environments with a transparent runtime consolidation framework. In Arthur B. Maccabe and Douglas Thain, editors, *Proceedings of the 20th ACM International Symposium on High Performance Distributed Computing, HPDC 2011, San Jose, CA, USA, June 8-11, 2011*, pages 217–228. ACM, 2011.

[30] Cyrille Rossant, Nicolas P. Rougier, João Comba, and Kelly P. Gaither. High-performance interactive scientific visualization with datoviz via the vulkan low-level GPU API. *Comput. Sci. Eng.*, 23(4):85–90, 2021.

[31] Amazon Web Services. Amazon Bedrock offers select FMs for batch inference at 50% of on-demand inference price. https://aws.amazon.com/about-aws/whats-new/2024/08/amazon-bedrock-fms-batch-inference-50-price/, 2024.

[32] Mohammad Shahrad, Rodrigo Fonseca, Iñigo Goiri, Gohar Irfan Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In Ada Gavrilovska and Erez Zadok, editors, *Proceedings of the 2020 USENIX Annual Technical Conference, USENIX ATC 2020, July 15-17, 2020*, pages 205–218. USENIX Association, 2020.

[33] Rong Shi, Sreeram Potluri, Khaled Hamidouche, Jonathan L. Perkins, Mingzhe Li, Davide Rossetti, and Dhabaleswar K. Panda. Designing efficient small message transfer mechanism for inter-node MPI communication on infiniband GPU clusters. In *21st International Conference on High Performance Computing, HiPC 2014, Goa, India, December 17-20, 2014*, pages 1–10. IEEE Computer Society, 2014.

[34] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

[35] Ivan Tanasic, Isaac Gelado, Javier Cabezas, Alex Ramírez, Nacho Navarro, and Mateo Valero. Enabling preemptive multiprogramming on gpus. In *ACM/IEEE 41st International Symposium on Computer Architecture, ISCA 2014, Minneapolis, MN, USA, June 14-18, 2014*, pages 193–204. IEEE Computer Society, 2014.

[36] Zhisheng Ye, Wei Gao, Qinghao Hu, Peng Sun, Xiaolin Wang, Yingwei Luo, Tianwei Zhang, and Yonggang Wen. Deep learning workload scheduling in GPU datacenters: A survey. *ACM Comput. Surv.*, 56(6):146:1–146:38, 2024.

[37] Lior Zeno, Avi Mendelson, and Mark Silberstein. Gpupio: the case for i/o-driven preemption on gpus. In David R. Kaeli and John Cavazos, editors, *Proceedings of the 9th Annual Workshop on General Purpose Processing using Graphics Processing Unit, GPGPU@PPoPP 2016, Barcelona, Spain, March 12 - 16, 2016*, pages 63–71. ACM, 2016.

[38] Husheng Zhou, Guangmo Tong, and Cong Liu. GPES: a preemptive execution system for GPGPU computing. In *21st IEEE Real-Time and Embedded Technology and Applications Symposium, Seattle, WA, USA, April 13-16, 2015*, pages 87–97. IEEE Computer Society, 2015.

A Artifact Appendix

Abstract

The artifact consists of the code of GPREEMPT and baseline used in evaluation, runner scripts for reproducing the experiments, plotter scripts for visualizing the results, and related workload trace and description files. It is intended for validating the claims made in the paper and facilitating further research on GPU preemptive scheduling. The code of GPREEMPT is host at https://github.com/thustorage/GPreempt.

Scope

The main claims of this paper, which can be verified through the provided artifact, are:

- GPREEMPT achieves low preemption-induced latency for latency-critical (LC) tasks when co-located with best-effort (BE) tasks.
- GPREEMPT maintains high system throughput for BE tasks even when co-located with preemptible LC tasks.
- GPREEMPT effectively supports preemption for a diverse range of GPU applications, including complex, non-idempotent workloads.

Hosting

GPREEMPT's artifact repository is hosted on GitHub at https://github.com/thustorage/GPreempt on the mas-

ter branch. The first usable commit version is the initial commit (59ab51e). However, please always use the latest commit due to possible future bug fixes and updates.

Contents

Our artifact contains the following components:

- `README.md`: A top-level guide for installing dependencies, building the system, and reproducing all experimental results described in the paper.
- `src/` and `include/`: Source code of GPREEMPT and the baseline system, including core implementation and interfaces.
- `patch/`: Patches for modifying the NVIDIA driver to support GPREEMPT's mechanisms.
- `scripts/`: Scripts to run the experiments, parse experiment logs, analyze performance metrics, and generate evaluation figures as presented in the paper.
- `config/`: Configuration files for different workloads and experimental settings.
- `third_party/`: External dependencies and modified third-party components required to build and run the system.

This structure is intended to facilitate reproduction of the paper's results and to help researchers extend GPREEMPT for their own experiments.

Requirements

- Ubuntu 22.04 LTS
- NVIDIA Open GPU Kernel Module 550.120
- glog
- jsoncpp
- disb [13]
- GDRCopy
- NVIDIA Ampere Architechture GPU

μEFI: A Microkernel-Style UEFI with Isolation and Transparency

Le Chen, Yiyang Wu, Jinyu Gu✉*, Yubin Xia, Haibo Chen*

Institute of Parallel and Distributed Systems, School of Computer Science, Shanghai Jiao Tong University

Abstract

The Unified Extensible Firmware Interface (UEFI) has established itself as the leading firmware standard in modern devices, offering enhanced extensibility, user-friendly graphical interface, and improved security capabilities. At the core of UEFI security is UEFI Secure Boot, designed to ensure that only trusted drivers and applications are loaded during system startup. However, the growing number of UEFI-related CVEs and the emergence of attacks that bypass UEFI Secure Boot have highlighted its limitations, exposing vulnerabilities that could be exploited by attackers.

We propose μEFI, the first isolation framework for UEFI firmware that can transparently run UEFI modules in sandboxes. Drawing inspiration from microkernel design, we deprivilege UEFI modules to user mode and isolate them in different address spaces (sandboxes). To enable the transparent execution of UEFI modules, we propose *trampoline injection* and *protocol analysis*. To further strengthen UEFI security, we incorporate a seccomp-like mechanism to restrict module capabilities and perform automated input validation to detect and prevent invalid inputs. Evaluation results demonstrate that our system can run complex UEFI modules without modifications, which incurs a small overhead of 1.91% for UEFI boot phase.

1 Introduction

The Unified Extensible Firmware Interface (UEFI) [92] has replaced the legacy BIOS and become the standard firmware interface in most devices, including servers, personal computers, and mobile devices [55, 58, 65]. It offers a modern interface that initializes the hardware, loads the operating system and provides runtime services for the OS. To strengthen the security, the UEFI specification introduced the Secure Boot feature, which verifies the integrity and authenticity of all modules before allowing them to load and execute [42], and becomes an essential security feature in the boot process [47, 73].

However, UEFI Secure Boot cannot protect against vulnerabilities within those legally signed modules, which may persist despite testing and auditing [34, 35, 71]. Besides, improper management of cryptographic keys may allow attackers to bypass UEFI Secure Boot [11]. A publicly disclosed UEFI bootkit capable of bypassing Secure Boot was reported in 2022 [84]. Furthermore, UEFI-related CVEs have been increasing in number, reflecting a concerning trend [24, 25, 29, 31, 32, 34–37, 39, 71, 72]. Exploiting these vulnerabilities enables attackers to gain unauthorized access to data or even take full control of the system.

A major factor driving the increase in vulnerabilities is the growing complexity of the UEFI ecosystem. The UEFI Forum community includes 365 vendors by 2025 [46]; The open-source project EDK2 [88], the most widely used framework for UEFI development and the base for most production firmwares, witnessed the total lines of code expand 9× between 2018 and 2025. The growth is largely driven by the rapid development of ARM architecture, as emerging ARM-based devices adopt UEFI firmware to improve system extensibility [2, 74, 83]. Consequently, the expansion of the codebase and proliferation of UEFI modules have significantly heightened the security risks within the ecosystem.

Moreover, the firmware supply chain is inherently intricate, involving multiple stakeholders such as original equipment manufacturers (OEMs), independent hardware vendors (IHVs), chip manufacturers, and BIOS/firmware providers. Downstream vendors build new modules atop firmware supplied by upstream sources [14]. This complexity increases the risk of introducing vulnerabilities into firmware and allows upstream flaws to propagate downstream, potentially compromising all devices that depend on the affected firmware.

To address the security risks of UEFI, we propose μEFI, a microkernel-style isolation framework for UEFI firmware which can run UEFI modules in sandboxes, based on the following observation: In UEFI, the majority of firmware logic resides in *modules* such as device drivers and bootloaders, while the UEFI *core* remains minimal, providing only essential functionalities. This pattern aligns with the microkernel design philosophy [12, 52, 63], which is widely recognized for its effectiveness in isolating errors in kernel modules.

Many UEFI modules, like the device drivers embedded in option ROMs[1], are available only as binaries. This makes compatibility a critical requirement for any new design. Applying microkernel-style isolation to UEFI while maintaining transparency for existing modules presents two challenges.

One key challenge is that the control flow transfer across modules and the UEFI core is hard to determine statically.

[1] An option ROM is a piece of firmware that stored on an external card.

UEFI module callouts are limited to two types: core-provided *services* and *protocol* interfaces offered by other modules, both accessed via function pointers. While services expose a fixed set of core functionalities, protocols represent capabilities of modules and are dynamically discovered through runtime lookup. The dynamic nature of interface discovery makes static analysis techniques used for analyzing Linux kernel module interactions [56, 69, 75] inadequate for resolving UEFI module control flow.

Another challenge stems from UEFI modules being designed with unconditional data sharing, without standardized format for inter-module data transfer. Consequently, data transfer between isolated modules must be explicitly coordinated. Since inter-module communications occur through protocol interfaces, and the type information of protocols is only available at compile time, the absence of such information at runtime makes data synchronization infeasible.

Fortunately, we make two key observations that inform our solution to the aforementioned challenges. First, both services and protocol interfaces are accessed through a centralized data structure known as the system table: services are retrieved directly via fixed pointers, whereas protocol interfaces are retrieved indirectly through service functions such as *LocateProtocol*. This unified access point allows us to intercept and redirect module callouts. Building on this insight, we introduce *trampoline injection* to dynamically generate and inject codes into modules, enabling seamless control flow redirection. Second, UEFI protocols are predefined and well-specified, allowing developers to rely solely on their interface contracts without requiring knowledge of the underlying module implementations. We leverage this by performing offline analysis of protocol definitions to extract type and structure information, which is then used to coordinate safe and structured cross-module data transfer at runtime.

To further enhance UEFI security, we incorporate a mechanism inspired by seccomp [68], which enforces capability restrictions at both the service and protocol levels. Additionally, our in-depth analysis of UEFI-related CVEs reveals that a significant portion of vulnerabilities stem from memory safety issues and inadequate input validation. To mitigate these risks, we implement *automatic parameter validation* for cross-module communications, which further strengthens the security of the firmware.

We have implemented a prototype of μEFI upon EDK2 [88] on both the x86_64 architecture using QEMU/KVM [6] and the AArch64 architecture using Raspberry Pi 4 Model B [80]. We perform security evaluations on potential vulnerabilities and, evaluate the performance of μEFI with six widely-used UEFI modules, ranging from simple modules like EnglishDxe to complex ones such as FAT and DiskIo. The performance results indicate that running all six modules in isolated sandboxes incurs only a small overhead of 1.91% on the UEFI boot time.

2 BACKGROUND AND MOTIVATION

2.1 Role of UEFI

UEFI has become the dominant firmware standard in modern devices. Most Windows machines [70], Mac with Intel processors [4], and even IoT devices [13] are using UEFI in their boot process. It plays a vital role in hardware initialization, system booting, and security. During system startup, UEFI initializes the hardware, ensuring all essential components are ready for OS handoff. For instance, UEFI detects, enumerates, and initializes USB devices, SSDs, and other peripherals. It also loads option ROMs from more complex devices like graphics cards, network adapters, and storage controllers. With the help of UEFI, users can configure the hardware and prioritize boot options conveniently. As for security, modern products default enforce the use of UEFI Secure Boot [94], which verifies the integrity of UEFI modules and ensures only trusted code is loaded. The use of UEFI Secure Boot has made UEFI a crucial part of the boot chain of trust, which is verified by hardware root of trust and responsible for validating subsequent OS bootloaders.

2.2 Security of UEFI

UEFI was designed to replace the legacy BIOS and featured for its support for larger storage devices, graphical user interface support, and enhanced extensibility. As malwares like rootkits and bootloader attacks became more sophisticated, UEFI Secure Boot was introduced as a defensive measure in 2011 [91]. It is considered to be the most important security feature concerning UEFI boot-time security and has become an industry-standard feature.

UEFI Secure Boot employs a signature and verification mechanism, ensuring that only signed UEFI modules can be loaded and executed during the system startup process. An essential prerequisite for the effectiveness of UEFI Secure Boot is that all trusted code must be free from vulnerabilities. This prerequisite is mainly supported by the following three reasons. First, in the early times, the codebase of UEFI ecosystem is rather small. As a result, the likelihood of vulnerabilities is relatively low. Second, most UEFI driver codes for specific devices are proprietary and closed-source, making it difficult for attackers to identify and exploit potential vulnerabilities. Third, before a UEFI module is certified to be used with UEFI Secure Boot, it typically undergoes code audit, which may include static analysis and testing to identify security vulnerabilities, logical errors, and non-compliance issues in the code [89, 90].

Increasing CVEs. In recent years, we have witnessed an increasing trend of UEFI-related vulnerabilities in the Common Vulnerabilities and Exposures (CVE) database. As shown in Figure 1, not only the number of UEFI-related CVEs is increasing, but also their criticality. The number of CVEs reached 72 in 2022, and 73 in 2023. Out of 45 vulnerabilities in 2024, 24 of them are rated with high severity. Besides,

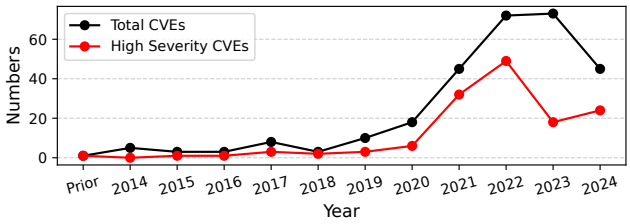


Figure 1: *Annual count of UEFI-related CVEs. The CVE count for each year is determined based on the year indicated in the CVE ID. The latest CVE ID for 2024 is CVE-2024-8105.*

many CVEs are disclosed only after fixes, suggesting that the number of 2024 CVEs is likely to increase in future updates. This trend can be attributed in part to the growing complexity of the UEFI ecosystem. On the one hand, the adoption of UEFI standards has expanded, with an increasing number of manufacturers using UEFI firmware for device management and initialization. The number of UEFI Forum members has grown from 236 in 2013 to 365 by 2025. On the other hand, the complexity of firmware modules has increased due to the introduction of more advanced and diverse devices, such as high-performance GPUs and smart NICs [78, 79], which demand enhanced capabilities and functionality.

2.2.1 Exploitation of UEFI Vulnerabilities

UEFI-related attacks typically have three main goals: obtaining control over the operating system, maintaining stealth to evade conventional security mechanisms, and achieving persistence that can survive operating system reinstallation and hard drive replacement. To gain a deeper understanding of UEFI-related threats, we further analyze how UEFI vulnerabilities are exploited by attackers to achieve the objectives.

Bypassing Secure Boot. While UEFI Secure Boot prevents the execution of unauthorized code, one prominent technique to circumvent Secure Boot involves exploiting vulnerabilities in legitimate, signed components. For example, the well-known BlackLotus bootkit [84]—previously sold on underground forums for approximately $5,000—exploited a vulnerability in older versions of the Windows bootloader [71] to overwrite memory regions containing Secure Boot policy data, effectively disabling Secure Boot enforcement. Another notable case is CVE-2024-7344 [40], where a signed third-party UEFI application embedded a custom image loader that bypassed standard secure UEFI loading functions, allowing the execution of unsigned binaries [82].

Compromising signed modules. Beyond exploiting the trust conferred by a signature to bypass Secure Boot directly, attackers also target inherent code-level vulnerabilities within signed UEFI modules, particularly DXE (Driver Execution Environment) drivers, to achieve arbitrary code execution or privilege escalation. Common examples include stack-based buffer overflows arising from the improper handling of UEFI services. For instance, vulnerabilities [17–20, 27, 28] have been identified in signed DXE drivers where sequential calls to the *GetVariable* service occur without correctly re-

Table 1: *Category of vulnerabilities and severity of corresponding CVEs.*

Severity	Memory Safety	Improper Input Validation	Uninitialized Data	Improper Access Control	Others
HIGH	44	34	5	17	40
MEDIUM	33	49	5	10	28
LOW	1	2	0	3	4
UNKNOWN	3	3	1	0	4

initializing size parameters, leading to stack overflows that can overwrite critical data, including return addresses [43].

Hooking mechanisms. Hooking mechanisms are extensively employed by UEFI firmware implants to intercept and redirect system execution flow. For instance, rootkits like LoJax [81] and MosaicRegressor [61] register malicious callback functions for specific UEFI events, such as *EFI_EVENT_GROUP_READY_TO_BOOT*, which is triggered just before the operating system bootloader is invoked. Another approach involves patching function pointers within UEFI service tables, redirecting calls to legitimate services towards malicious routines. The CosmicStrand rootkit, for example, was found to patch the entry point of a DXE driver and deploy a multi-stage hooking process that extended control into boot manager, OS loader, and even kernel functions [62]. Such deep-seated hooks allow firmware implants to subvert operating system-level security controls, maintain persistence across reboots, and operate with a high degree of stealth.

2.2.2 Vulnerability Analysis

By analyzing UEFI-related CVEs (as shown in Table 1), we identify four major categories of vulnerability causes: memory safety issues, improper input validation, uninitialized data, and improper access control. The statistics show that memory safety issues and improper input validation stand out in terms of both frequency and severity. Improper input validation [26, 36, 38] accounts for the largest proportion, representing 30.8% of all UEFI-related CVEs. And memory safety issues [15, 16, 21–23, 27, 30] are particularly critical, constituting 31.4% of high-severity CVEs. While the types of vulnerabilities are consistent with those commonly found in traditional software, their impact is significantly amplified in the context of UEFI firmware, primarily due to three factors:

- **Privileged execution environment**. All UEFI modules execute with high privileges, such as in System Management Mode on x86 or EL1 on ARM. Although this privileged execution is necessary for managing hardware and configuring the boot process, it also provides a convenient and powerful vector for malicious code.
- **Lack of inter-module isolation**. UEFI modules share a common address space and communicate via global variables or protocols. The absence of isolation allows a flaw in one module to compromise others, potentially affecting the entire firmware.
- **Coarse-grained access control**. The access control mechanisms in UEFI are less sophisticated compared to modern operating systems. Concerning the interactions between

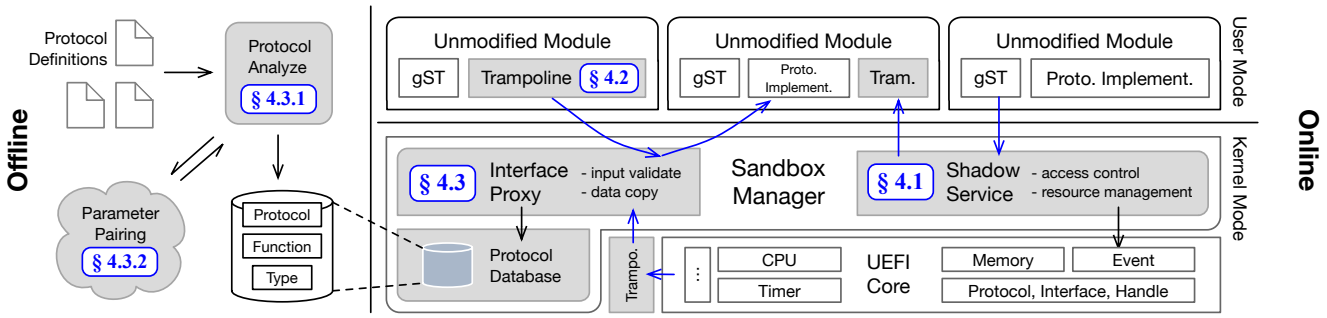


Figure 2: *µEFI's architecture.*

modules and the core, services and protocol interfaces are both globally accessible, potentially leading to unauthorized modifications.

Given these inherent limitations, we propose that a more robust isolation mechanism is needed to enhance the security of UEFI firmware beyond its current design.

2.3 Driver Isolation in UEFI vs. OS

Device driver isolation is a critical topic for enhancing the reliability and security of operating systems. The core principle involves executing drivers within protected environments to contain faults and prevent unauthorized access to kernel resources. Research in this area spans decades, leading to a diverse array of approaches. These include architectural strategies like user-mode drivers and microkernel design [8, 45, 49, 53, 54, 67], software-enforced methods such as software fault isolation [10, 44, 69, 95] and language-based safety [48, 76], virtualization-based containment [9, 50, 66, 77], and hardware-assisted mechanisms like memory protection keys [51, 52, 93]. Early systems like Nooks [85] demonstrated the feasibility of fault isolation within monolithic kernels. More recent efforts, exemplified by KSplit [56], focus on automating the complex process of isolating existing drivers and rely on static analysis to identify shared kernel driver states and generate synchronization logic.

Although driver isolation techniques has been extensively explored in traditional operating systems, the differences between the execution environment of UEFI modules and that of OS drivers bring new challenges and opportunities. There are three key differences. *First*, dynamic interface discovery. The UEFI core does not provide runtime symbol resolution for modules. Each module is a self-contained image, and its provided or required interfaces are identified only at runtime through protocol installation and lookup. As a result, static code analysis alone is insufficient for automating UEFI module isolation. *Second*, single-threaded execution. UEFI follows a single-threaded, event-driven execution model. There are neither synchronization primitives nor complex atomic operations, greatly simplifying concurrency concerns. *Third*, well-defined parameter scheme. All of the interfaces used by UEFI modules belong to pre-defined protocols, which means the parameter types and semantics are explicit. This characteristic enables effective data synchronization without reliance on static analysis of source code or binaries.

3 OVERVIEW

To address the security threats faced by UEFI while accounting for its unique characteristics, we present *µEFI*, providing transparent and fine-grained isolation for UEFI modules.

Design Goals. *µEFI* has four goals, focusing on the characteristics of UEFI vulnerabilities and the shortcomings of existing security measures.

- *G1: module memory safety.* *µEFI* must guarantee memory safety across modules. A UEFI module can only access its own memory and those legally shared by other modules.
- *G2: full module transparency.* *µEFI* should require no modification to the modules. Firmware images can be transparently loaded and executed with *µEFI*.
- *G3: module access control.* *µEFI* aims to provide seccomp-like mechanism in UEFI firmware, restricting modules' capabilities while ensuring that they can still perform their intended functions.
- *G4: automated input validation.* *µEFI* should be capable of automatically validating parameters for cross-module calls without the need for custom handling logic for each individual interface.

Approach: Isolation with microkernel architecture. UEFI follows a modular design in which modules interact through well-defined protocol functions. The majority of firmware code resides within UEFI modules, while the UEFI core remains minimal, performing only essential functionalities. The modular design principles of both microkernel and UEFI align closely, with the isolation of microkernel architecture fulfills the missing parts in UEFI. As a result, we apply microkernel architecture in UEFI firmware, deprivileging UEFI modules to user mode and isolating them with separate page tables.

Architecture. Figure 2 presents the architecture of *µEFI*. *µEFI* is composed of two parts: an offline analyzer and an online sandbox manager. The offline analyzer parses the definitions of protocols and constructs a database containing protocol-related information (§4.3.1). The sandbox manager controls the capabilities of sandboxed UEFI drivers and applications and serves as the supervisor of all sandboxed UEFI

modules. To facilitate the integration of the sandbox manager into existing UEFI firmware, the sandbox manager is implemented as a UEFI module operating alongside the core, thereby minimizing the need for modifications to the core. In the sandbox manager, we first introduce shadow service (§4.1) to manage the resources of sandbox and impose seccomp-like restrictions on sandboxed modules. Then we use trampoline injection (§4.2) to support cross-module calls without modifying existing UEFI modules. Before a cross-module call can be actually handled, interface proxy (§4.3) intercepts the call, validates the parameters to prevent error propagation, and performs necessary data transfer.

Threat model and security guarantees. μEFI assumes the UEFI core[2] is free of vulnerability and all preceding phases work as expected. UEFI drivers and applications loaded after the initialization of the core are not trusted, even if they are signed by legal third-party certificate authority, as these modules may have potential vulnerabilities.

As for security guarantees, μEFI ensures the memory safety of UEFI modules. During execution, a module cannot access memory not belong to it, and the corresponding memory attributes must be set (read/write/execute). μEFI also restricts the services and protocols a module can use. Modules should only be granted the capabilities they need to perform their work. In addition, μEFI provides the ability to validate the parameters for cross-module calls, making sure that the parameters are legally used by the caller. With the above guarantees, μEFI addresses the shortcomings of UEFI Secure Boot and further strengthens UEFI security.

3.1 Challenges and Insights

Transparent sandbox callout. The sandbox's callouts include invocations of services provided by the core and calls to protocol interfaces implemented by other modules. Originally, all the communications across UEFI modules and the core are conducted through function calls. With the deprivileging and isolation of UEFI modules, direct function calls are no longer feasible. However, the original calling convention must be preserved to ensure compatibility with existing modules.

Insight-❶: unique callout gate. System table is a central data structure in UEFI that provides access to services and configuration tables. Through the system table, we can manage all the callouts directly or indirectly, as services are accessed through the system table and protocol interfaces are obtained through services (depicted in Figure 3).

Transparent data transfer. UEFI modules communicate via protocol interfaces, with data shared directly across modules. Once isolation is enforced, explicit data synchronization becomes necessary. Since protocol type information is only available at compile time, the lack of such metadata at runtime makes synchronization across sandboxes infeasible.

[2]Specifically, the UEFI DXE Core, which is responsible for loading and managing DXE phase drivers.

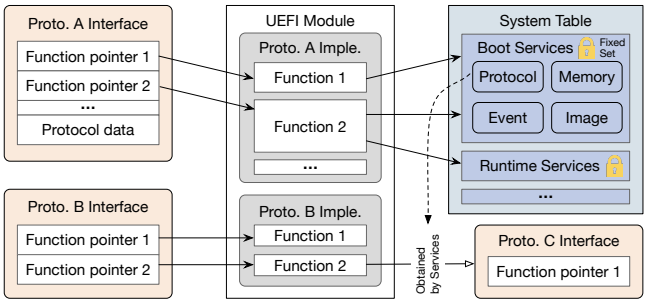


Figure 3: *Construction of protocols [87] and UEFI system table. Protocol defines a set of functions and data structures. Interface refers to the implementation of protocol. All the callouts of a module have to go through the system table directly or indirectly.*

Insight-❷: standardized protocols. To facilitate the independent development of UEFI modules, the UEFI specification defines standardized protocols for various types of drivers and applications. Thus, a caller module can interact with target modules without concerning about their specific implementations, and device drivers by different vendors can achieve compatibility with existing systems simply by implementing the standard protocols.

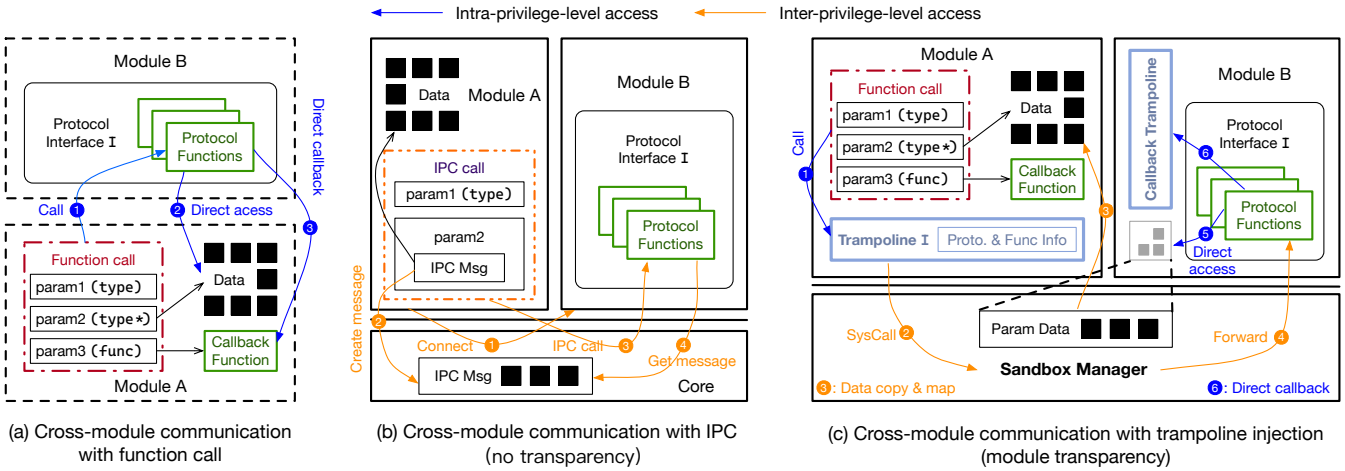
4 DESIGN

We employ a combination of techniques to achieve transparent isolation of binary-form modules. By controlling a unique callout gate, μEFI transparently redirects service calls to a corresponding set of system calls (§4.1). It also supports transparent protocol function calls between isolated modules by replacing original protocol function pointers with injected trampolines (§4.2). Leveraging the standardized nature of protocol definitions, μEFI extracts type information through offline analysis and stores it in a protocol database (§4.3.1). To enable correct data transfer and automatic input validation, μEFI employs a heuristic-based technique known as parameter pairing (§4.3.2). It further adopts customized tracking strategies for different types of data involved in the invocation process (§4.3.3).

4.1 Shadow Service

The UEFI core provides a standardized set of services to UEFI modules, including Boot Services and Runtime Services. These services offer essential functionalities throughout the lifecycle of a UEFI module, such as memory allocation, event registration, and protocol interface acquisition and installation. In the original architecture, the core focuses solely on providing capabilities without imposing restrictions on their usage, allowing UEFI modules to access most system resources without limitations. Therefore, we present shadow service to delegate the core services and manage the resources.

UEFI services are presented to modules as a set of function pointers contained within the system table, a global variable initialized by the core and passed to modules during initializa-



(a) Cross-module communication with function call

(b) Cross-module communication with IPC (no transparency)

(c) Cross-module communication with trampoline injection (module transparency)

Figure 4: *Cross-module communication workflow with (a) function call, (b) normal IPC, (c) trampoline injection. Trampolines are dynamically generated according to the associated protocol and function.*

tion. As μEFI isolates each module in its own address space, the old manner of function call is no longer feasible. Combining with the fact that the set of core services is rather stable, the primitive system call is perfectly suitable for providing services to sandboxed modules. To achieve full transparency, during the loading process of sandboxed modules, we replace the origin system table pointer with a per-sandbox shadow system table, and the services with functions that perform system calls. Every time the services are called in the sandbox, the control flow is redirected to the shadow service. Inspired by seccomp, the shadow service enforces two levels of access control before handling the request.

Service-level access control. The shadow service first regulates the sandboxed module's access to core services. Among the services provided by the core, fundamental capabilities such as memory allocation, protocol installation and acquisition, and event creation and signaling are universally required by all modules. However, services like *LoadImage* and *StartImage* provide more sensitive functionalities and are only necessary for a limited number of modules, such as the UEFI shell or GRUB. Therefore, the shadow service determines whether to grant access to high-privilege services based on the type of each module.

Protocol-level access control. Protocols serve as the fundamental mechanism for interaction between modules. An important fact is that the proper functioning of a UEFI module requires interaction only with modules directly related to its functionality. For instance, a UEFI driver responsible for hard drive I/O may invoke block device drivers, but has no need to access network-related features. Thus, every time a sandboxed module attempts to acquire or install a protocol interface, the shadow service evaluates the request and determines whether to grant access based on its type and the protocol involved.

Unfortunately, under the current specification, it is not feasible to directly determine a module's type and functionality during the process of loading its image. To address this limitation, we employ two distinct approaches:

- **Explicit declaration.** Each module is uniquely identified by a GUID, and developers are required to explicitly declare the services and protocols a module is allowed to invoke. This enables strict, seccomp-like sandboxing and enforces a security-by-contract model.
- **Heuristic detection.** Modules are restricted to access protocols of a specific type. The module type is inferred based on the first non-generic protocol interface it attempts to acquire or install. This approach adopts a fail-stop execution model upon missing or unexpected protocols, with type classification resolved during pre-deployment testing.

By combining these two approaches, the system establishes a tiered security model that balances strict least-privilege enforcement with backward compatibility for legacy modules.

4.2 Safe Trampoline Injection

When a module intends to use the functionality of other modules, it obtains the corresponding protocol interface through the core services, and directly invokes functions in the protocol interface to fulfill its job. As illustrated in Figure 4 (a), references provided in call parameters are accessed directly, and callback functions are invoked without intermediaries.

Since μEFI isolates different modules, we can no longer access the functionalities within protocol interfaces through direct function calls. In the microkernel architecture, interprocess communication (IPC) is introduced to enable cross-module interactions. However, IPC cannot be seamlessly applied to UEFI modules because it relies on dedicated interfaces and requires explicit cooperation between the client and server. The IPC client must know the IPC server's identity, intentionally establish a connection, request the kernel to prepare IPC payloads, and invoke the server using a specialized IPC call, as illustrated in Figure 4 (b). Moreover, the approach of shadow service is not suitable for managing protocol inter-

faces. Unlike core services, the functions provided by protocol interfaces cannot be encapsulated within a fixed set, as they represent the diverse functionalities of UEFI modules.

μEFI introduces trampoline injection to enable transparent cross-module protocol function calls. Trampoline injection is implemented with the assistance of shadow service. When a sandboxed module attempts to acquire a protocol interface provided by another module, the sandbox manager does not provide the actual protocol interface. Instead, it dynamically generates trampolines that includes metadata about the protocol functions and injects it into the sandbox. Figure 4 (c) illustrates the workflow of cross-module communication facilitated by trampoline injection. When a sandboxed module invokes a protocol function, the call is transparently redirected to the injected trampoline (❶). The trampoline then issues a system call, carrying the parameters and transferring control to the sandbox manager (❷). The sandbox manager processes the parameters (❸) before forwarding the call to the target module (❹). For the target module, data passed through the cross-module call is also directly accessible (❺), and even callback functions can be invoked directly with the aid of another trampoline (❻).

Trampolines are injected into the sandbox without introducing security risks. Generation of a trampoline signifies that access to the protocol is approved by the shadow service. Furthermore, trampolines are configured as read-only upon injection, ensuring that their logic cannot be modified.

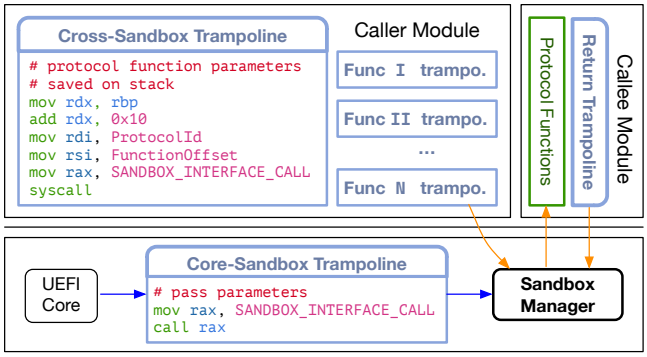


Figure 5: *Three types of trampolines with different use cases and the work mechanism of trampolines.*

Three types of trampolines. In addition to facilitating cross-module communication, trampolines are also employed within the core to enable transparent invocation of sandboxed modules by the core, as well as within the callee sandbox to ensure the automatic return of control after function execution. During module management, the core leverages several basic protocols to retrieve module information and control module lifecycle operations, such as obtaining the module's name or checking whether a driver supports a specific device. These protocol interfaces are also obtained through core services and accessed via function pointers. To minimize modifications to the core, the shadow service generates trampolines for the core every time a sandboxed module installs its protocol interface. When the core invokes these interfaces, the calls are routed to the sandbox manager.

Figure 5 illustrates the three use cases of trampolines and highlights their differences in detail. In summary, trampolines provide a transparent mechanism for cross-module calls and support different kinds of interfaces. For instance, asynchronous processing logic is implemented using events, and callback functions for sandbox events can be executed via trampolines. Although their application scenarios differ, all trampolines redirect the control flow to the sandbox manager. Within the sandbox manager, the interface proxy handles the preprocessing of parameters before invoking protocol functions and synchronizes data upon completion of the calls.

4.3 Interface Proxy

The interface proxy is responsible for validating the parameters of the call and, when necessary, copying the parameters. Although the UEFI core manages the installation and acquisition of protocols, it only records basic information about the installed protocols, such as the module that installed the protocol, the corresponding interface that implements the protocol, and a unique GUID used to identify the protocol. Lacking information related to the contents of the protocol, it becomes challenging to validate the parameters of protocol function calls without modifying the module's code. To address this issue, the interface proxy needs to cooperate with the offline analyzer. During online execution, the interface proxy identifies the protocol GUID corresponding to a protocol function call and queries the protocol's metadata with the GUID from the protocol database, which is constructed in the offline process. With the metadata, the interface proxy can validate the call parameters and, guided by this metadata, formulate data transfer strategies.

4.3.1 Protocol Database

Protocols are basic building blocks in UEFI firmware designed to facilitate communication and functionality between various components, drivers, and applications. Their definitions include both data structures representing device-specific configurations and functions for invoking device functionalities. The offline analyzer reads the protocol definition, parses the data structures, functions, and type definitions, and generates the protocol database. To reduce the size of the protocol database and avoid excessive query latency, the analyzer records only the essential information. We define three tables in the protocol database to store different types of information: *protocol*, *function*, and *type*. The protocol table records the name and GUID of each protocol, and variables contained within each protocol. The function table stores information about the function parameters, including their types, names, and input/output attributes. The type table captures other custom data structures and macro definitions. Figure 6 illustrates an example protocol definition of SerialIOProtocol, which is defined with the above three types. Once generated, the pro-

```
# @param Control      A pointer to return the control signals
Func GetControlBits:
  OUT Type *Control

# @param BufferSize   Input: size of the Buffer.
#                     Output: amount of data returned.
# @param Buffer       The buffer to return the data into.
Func Read:
  IN OUT Type *BufferSize
  OUT Type *Buffer

Func Write:
  IN OUT Type *BufferSize
  IN Type *Buffer

# Used to communicate with all UART-style serial devices.
Protocol SerialIOProtocol:
  Func GetControlBits
  Func Read
  Func Write
  Type *Mode  # pointer to mode data
```

Figure 6: *Example protocol definition for serial I/O device.*

tocol database is included in the sandbox manager's firmware image at compile time and contains all the information required by the interface proxy during runtime.

At runtime, the interface proxy does not load the entire protocol database at once; instead, it employs a lazy loading approach. During the initialization process, only the protocol table is parsed, while the types and functions associated with each protocol are not recursively analyzed. When the interface proxy processes a cross-module call request for the first time, it dynamically queries the necessary function and type definitions required to handle the request.

4.3.2 Call Parameter Validation

The interface proxy begins with validating the parameters of cross-module calls. A key observation is that severe vulnerabilities caused by improper input validation often stem not from logical errors (i.e., valid inputs that are logically incorrect), but from a failure to verify the validity of input parameters. For instance, a caller might pass a memory address to the callee that it is not authorized to access, or set the buffer size in parameters (example in Figure 6) to 128 bytes when the actual buffer size is only 64 bytes. Such invalid behaviors can result in unauthorized memory access or overwrite. Thus, the interface proxy focuses on validating the legitimacy of pointers within parameters.

Before initiating validation, the interface proxy must determine whether a pointer references a memory buffer, represents an array, or serves as an identifier handle/token. However, at runtime, it is impossible to determine this solely based on the values passed by the caller. To address this, μEFI employs a heuristic method in the offline analyzer to decide how to validate a given pointer. The information is recorded in the function table and is later utilized by the interface proxy to verify whether the pointer references a valid address and whether the caller has necessary access permissions for the passing memory region.

Offline parameter analysis. When analyzing function parameters, the offline analyzer begins by checking the data type of pointers. If the data type of a pointer is a protocol, the analyzer interprets it as a handle. This approach is based on the observation that protocols are seldom used in function parameters to transfer data. Instead, the protocol itself serves as an identifier, similar to the 'this' pointer in C++ class.

Then, the analyzer tells if a pointer references a buffer or an array, using a heuristic method named *parameter pairing*. Due to the inherent limitations of the C language, passing an untyped pointer to represent a memory buffer in a function call does not allow the callee to determine the buffer size unless both parties have established a standard logic for handling parameters. After analyzing numerous protocol definitions, we observe that when an untyped reference to a memory buffer is passed, it is often accompanied by an additional parameter specifying the buffer size. Similarly, for pointers that have a deterministic type size, the analyzer checks whether there is another parameter indicating the number of elements in the array. For pointers that need to be parsed with specific methods (e.g. string and device path), the offline analyzer leaves the job to the interface proxy, and the buffer size is determined at runtime using standard library functions (e.g. *StrLen* and *GetDevicePathSize*).

Approach of parameter pairing. The offline analyzer utilizes parameter names and function comments from protocol definitions to perform parameter pairing. It first examines whether the parameter names conform to certain patterns (e.g., *Buffer* and *BufferLength*, or *Data* and *DataCount*). If they do not, the offline analyzer proceeds to analyze the function comments. Since developers typically focus on protocol definitions rather than implementation details when they use protocols, function comments within protocol definitions often provide detailed explanations of the function's purpose and the roles of its parameters. This enables the offline analyzer to leverage large language models (LLMs) to identify potential relationships between parameters.

Limitation. The offline analyzer fails to resolve parameters of type void* when they cannot be unambiguously classified as handles, data buffers, or arrays. Such cases, which account for less than 2.5% of instances, typically involve implicit payload semantics between caller and callee—for example, parsing packets passed via void* in network modules. In these scenarios, manual effort is required to correctly process the parameters. At runtime, the sandbox manager flags unhandled interface arguments when such ambiguous cases are encountered without proper handler.

4.3.3 Interface-aware Data Transfer

Given that modules in different sandboxes cannot access each other's memory, the interface proxy not only validates the parameters of protocol function calls but also carries out data transfer and synchronization during the process. Without the use of specialized hardware features, memory isolation is typically maintained at the page level. As a result, when handling memory references in cross-module calls, the interface proxy needs to allocate a memory region for the callee and copy the data from the caller's address space to the callee's.

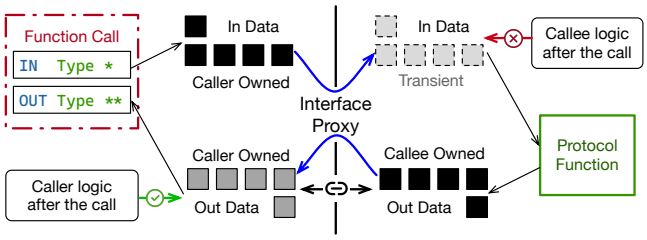


Figure 7: *Example of data transfer in cross-module protocol function call and the lifecycles of the memory buffers.*

By utilizing information from the protocol database and results of parameter pairing, the interface proxy is capable of recursively dissecting complex data structures and handling parameter transfer seamlessly, thereby obviating the need for UEFI modules to explicitly manage data transfer.

Cross-module memory management. During the handling of protocol function calls, the callee might allocate a memory region for the caller, write the data to be returned into this memory, and then return a reference to this memory to the caller—utilizing multi-level pointers. In such scenarios, the interface proxy needs to resolve the contents of multi-level pointers layer by layer until it accesses the actual data, and then allocates a buffer for the caller to store the data.

The key difference between memory buffers returned by the callee and those used for input data lies in their lifecycles. As illustrated in Figure 7, the memory allocated for transferring input data can be immediately released after the call finishes. In contrast, memory buffers returned by the callee are intended for use in the caller's subsequent execution and thus have a lifecycle extending beyond the scope of the current protocol function call. Moreover, the caller only accesses the data without managing it. Memory allocated by the callee for the caller is typically released by the callee itself, often through another protocol function. For instance, *OpenFile* returns a pointer of an *EFI_FILE* structure which should later be freed by calling *CloseFile*.

To prevent memory leaks, it is necessary to continue tracking the memory buffer even after the current protocol function call has completed. Since all memory in sandboxed modules is managed by the shadow service, additional metadata can be recorded for allocated buffers. When the interface proxy allocates memory for the caller to transfer returned data, the shadow service binds this buffer to the memory that is previously allocated by the callee, establishing a hierarchical relationship (depicted in Figure 7). When the callee module releases its allocated memory, the shadow service ensures that dependent memory is also freed in a cascading manner.

5 IMPLEMENTATION

We implemented a prototype of μEFI on both the x86_64 and the AArch64 architectures. To avoid invasive modifications to the core, the sandbox manager is implemented as a UEFI module running at the same privilege level as the core. The core is modified to ensure that the sandbox manager is prioritized during the module loading process. It interacts with the sandbox manager to schedule the sandboxed modules exclusively through interfaces defined by *SANDBOX_PROTOCOL*, facilitating operations such as *CREATE_SANDBOX* and *START_SANDBOX*. The sandbox manager module consists of 4,773 SLOC in C and 253 SLOC in assembly, while the modification to the core is less than 100 SLOC.

In terms of the offline analyzer, we leveraged LLVM Frontend Infrastructure, more specifically, LibClang, to directly traverse and parse all the protocol headers of UEFI drivers primarily in the MdePkg of EDK2 Repository. We programmed the offline analyzer in C++ to extract all the types info and function signatures used by all protocols and selectively filter the protocols based on the INF sections defined in firmware's FV definition. After the process of type extraction, the offline analyzer is set to dump the protocol definitions inside a JSON file which is later statically embedded inside the Sandbox Manager Driver. The offline analyzer consists of 907 LOCs in C++ and the generated JSON file is less than 20000 LOCS.

6 EVALUATION

We evaluated μEFI to answer three major questions:

- Can μEFI protect UEFI firmware from vulnerability exploitation and real-world attacks? (§6.1 & §6.2)
- To what extent can μEFI achieve generality in ensuring module transparency? (§6.3)
- How much overhead does μEFI incur? (§6.4 & §6.5)

Experimental Setup. Our system is based on the edk2-stable202402 release version of UEFI EDK2. For evaluation, we utilize two distinct platforms:

- Intel i7-13700 Desktop: This platform supports the VT-x virtualization feature. We use the EDK2 OVMF package to run μEFI within KVM-enabled QEMU virtual machine. In this setup, the core operates at Ring 3, while sandbox modules run at Ring 0. Execution cycles are measured by reading the TSC (Timestamp Counter).
- Raspberry Pi 4 Model B: This platform features a quad-core Cortex-A72 (ARM v8) 64-bit SoC. On this system, the core executes in EL1, and sandbox modules are confined to EL0. Execution cycles are measured by reading the PMU (Performance Monitoring Unit) registers.

6.1 Empirical Security Evaluation

To evaluate the effectiveness of μEFI in defending against various vulnerabilities, we constructed multiple test cases based on known CVEs. These cases target potential issues within modules, including memory safety vulnerabilities and improper input validation.

Security protection for memory safety. Memory safety issues, such as stack overflow, heap overflow, and use-after-free, can lead to the corruption of critical data structures and potentially enable arbitrary code execution. To evaluate the

effectiveness of our defense mechanisms, we designed two test cases targeting stack and heap overflow scenarios.

Stack-based buffer overflow. CVE-2021-39297 is a stack overflow vulnerability found in the UEFI firmware of certain HP laptops [43]. Exploiting this vulnerability allows an attacker to overwrite the return address of the parent function. In μEFI, a temporary stack is allocated each time control is transferred to a sandboxed environment. Upon function return, a return trampoline is employed to securely redirect control back to the sandbox manager. Combined with the fact that a sandboxed module can only access its own code and data, this design ensures that malicious code outside the vulnerable module cannot be injected or executed via stack overflows.

Heap-based buffer overflow. Heap overflow vulnerabilities occur when a program allocates dynamic memory on the heap but fails to properly validate buffer lengths, allowing an attacker to overwrite adjacent heap data or metadata [21–23]. In μEFI, the heap memory of each sandbox is managed by the sandbox manager independently. Even if an attacker exploits a vulnerability to overwrite heap data, the access is confined to the heap data of the compromised module itself.

Security protection for improper input validation. We design two types of protocol function call inputs, each representing common scenarios that require different levels of input validation by the callee. The first type involves cases in which the caller passes a memory pointer that it does not own, potentially resulting in unauthorized memory writes. The second category involves cases where the caller supplies a valid memory buffer but specifies a *BufferSize* parameter that exceeds the actual buffer size. This can cause out-of-bounds memory access and unintended data corruption.

In μEFI, the interface proxy addresses these issues by leveraging parameter pairing to associate *Buffer* with its corresponding *BufferSize*. It then consults metadata maintained by the shadow service's memory management subsystem to verify the legitimacy of the input buffer, thereby preventing illegal memory operations.

There are a few cases in which the interface proxy may fail. For instance, in CVE-2023-39538 [33], two input parameters are used to initialize a buffer without proper validation. This can lead to an integer overflow, resulting in the allocation of a buffer that is too small to hold the intended data and ultimately causing heap overflow [7]. Because the vulnerable buffer is not directly passed as a function parameter, the interface proxy cannot detect this issue. Nevertheless, the exploit is still mitigated by the memory isolation enforced by μEFI.

6.2 Security Analysis

We further demonstrate how μEFI mitigates vulnerability exploitation by analyzing UEFI-specific attacks in the wild and PoC attacks proposed by researchers.

ThunderStrike represents an early firmware attack targeting UEFI. It is launched via a compromised Thunderbolt peripheral, which leverages its Option ROM to inject malicious code into the firmware. A critical step in its execution involves the malicious module altering the normal boot sequence. Specifically, it replaces the *ProcessFirmwareVolume* function pointer in the DXE services table with its own function pointer. This manipulation allows the bootkit to seize control over subsequent stages of the boot process [57]. As μEFI utilizes the shadow service to ensure that the service table accessed by each sandboxed module is an independent copy, even if a malicious module modifies a function pointer in its service table, the hook will not affect other modules.

BootHole exploits a buffer overflow vulnerability within the GRUB2 bootloader, triggered when parsing a maliciously modified grub.cfg configuration file by an attacker with administrative or physical access. This buffer overflow overwrites critical structures in the heap and allows for arbitrary code execution. As GRUB2 itself is signed and verified, this effectively bypasses Secure Boot and enable the loading of untrusted code or the installation of persistent bootkits [41]. If μEFI is deployed, the memory isolation enforced between sandboxes prevents an attacker from overwriting memory belonging to other modules or the core. This significantly raises the difficulty of exploiting such vulnerabilities.

BlackLotus is a sophisticated UEFI bootkit recognized for its capacity to circumvent Secure Boot by leveraging vulnerabilities like 'BatonDrop'. A key aspect of its methodology is that it loads a malicious module ('grubx64.efi') and intercepts the execution of components included in the typical Windows boot flow, such as Windows Boot Manager, Windows OS loader, and Windows OS kernel, and hooks some of their functions in memory. This deep level of interference allows BlackLotus to effectively neutralize various security mechanisms, including BitLocker and Hypervisor-protected Code Integrity (HVCI) [84]. With μEFI in place, even if Secure Boot is bypassed, running 'grubx64.efi' within a sandbox prevents it from inserting malicious hooks into other modules.

6.3 Generality of Offline Analysis

A critical question is how general can μEFI be used for providing isolated execution environments for UEFI modules. To answer this question, we employ the offline analyzer to examine all protocols within the UEFI EDK2 project. Because the key to achieving module transparency is to support different interfaces and correctly handle the pointers, we focus on analyzing pointer parameters in functions and pointer fields in the definition of protocols and compound types. We also include non-pointer types in the analysis to reflect the

Table 2: *Number of different types of pointers in the definition of Functions, Protocols, and Compound Types in UEFI EDK2. Total Fields contain the parameters that are not pointers.*

Definition	Buffer w/ Size	Array w/ Count	Handle /Token	Parse w/ Library	Manual Effort	Total Fields
Protocol	0	0	22	10	2	1147
Compound Type	28	1	85	189	87	33278
Function	159	6	369	101	90	3639

broader applicability of μEFI, since most interface parameters are plain data structures in production firmware.

The offline analyzer recognizes four types of pointers that need to be specially handled, as detailed in Table 2. Among all the protocol fields, only two fields require manual intervention for adaptation. This is because the definitions of most protocols are composed of function pointers and a few simple data structures that represent the hardware configurations. Similarly, the failure rate for analyzing compound types is as low as 0.26%. Regarding functions, 17.4% of all parameters are classified as belonging to the four special pointer types, with a failure rate of 2.5%. Moreover, many failure cases are clustered within a few modules that contain multiple data structures requiring manual effort.

6.4 Performance Evaluation

6.4.1 UEFI Boot Performance

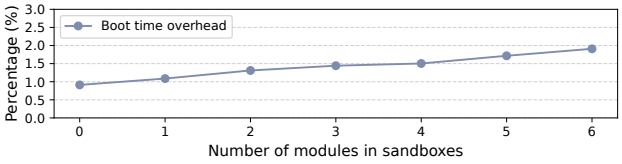


Figure 8: *The impact of the number of sandboxed modules on UEFI boot time (measured from the entry of DxeMain to the completion of ExitBootServices).*

To evaluate the performance impact of running modules in sandboxes, we first measure and compare the boot time of UEFI [3] with and without sandboxed modules. The comparison of boot latency focuses on the UEFI boot process, from the entry into the DXE phase to the handover of control to the operating system. We select six widely used UEFI modules, ranging from simple modules such as EnglishDxe to complex ones like FAT, which is among the most complex in EDK2, interacting with other modules via a rich set of protocols and intricate data structures.

As illustrated in Figure 8, running six modules in different sandboxes only incurs an overhead of 1.91%. When no modules are executed within the sandbox, the loading of the sandbox manager introduces a latency of 0.91%. This overhead arises from the sandbox manager's initialization process, during which it preloads portions of the protocol database. The loading of each individual module incurs an average overhead of 0.17%, with the overheads of running different modules ranges from 0.06% to 0.22% (more details in §6.4.2). These differences arise because modules perform distinct tasks. Modules are occasionally invoked by the core via the *EFI_DRIVER_BINDING_SUPPORTED* interface to determine whether they support specific protocols. In certain cases, modules may be loaded but their interfaces are never invoked due to a failed driver binding process. In contrast, modules such as FAT are both loaded and executed. The more complex the protocol interfaces they expose, the higher the runtime overhead they incur. Moreover, during system boot process, the operating system's boot time significantly exceeds the boot time of the UEFI firmware, rendering the overhead of μEFI more negligible.

[3] We failed to configure to boot Linux with GRUB on Raspi4, the boot time is only tested on the x86_64 platform.

6.4.2 Module Execution Performance

To evaluate the performance impact of μEFI on UEFI modules, we conduct experiments on two frequently invoked and highly interactive modules during the boot process: FAT and DiskIo. The protocols provided and consumed by these modules are sufficiently representative to cover all the parameter cases discussed in §4.3. The end-to-end test involve creating a file using FAT-provided interfaces, writing to and reading from the file, and then deleting it.

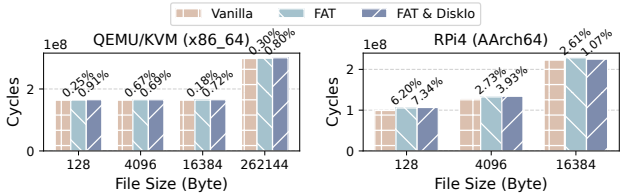


Figure 9: *Execution time for end-to-end test with and without FAT and DiskIo in sandboxes.*

As illustrated in Figure 9, in the x86 virtual machine, running FAT with μEFI incurs an average overhead of only 0.35%, while isolating FAT and DiskIo into separate sandboxes results in an average overhead of 0.78%. This increased overhead is attributed to the additional context-switching and data transfer involved in interactions between sandboxed modules compared to those between sandboxed and core modules On the Raspberry Pi 4 platform, the corresponding overheads are 3.85% and 4.11%, respectively.

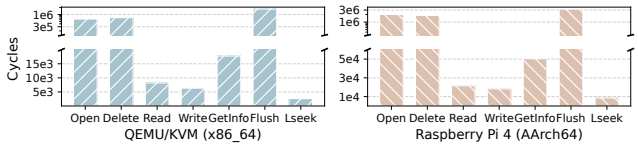


Figure 10: *Execution overhead of various FAT interfaces when executing FAT and DiskIo within μEFI.*

Figure 10 shows the execution overhead of various FAT interfaces by comparing baseline performance with that observed when both FAT and DiskIo modules are executed within sandboxes. On the x86_64 platform, Read, Write, and Lseek operations each take less than 9,000 additional cycles. As for others, Open, Delete, and Flush involve I/O operations, where the time consumed by the driver's internal logic significantly exceeds the overhead introduced by μEFI. The overheads shown primarily reflect the natural fluctuations in the interface execution times, which are less than 5% compared to the baseline.

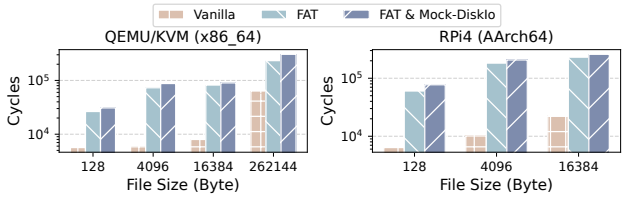


Figure 11: *Execution time for end-to-end test with and without FAT and Mock-DiskIo in sandboxes.*

Mock test excluding I/O operations. UEFI primarily initializes hardware and sets up the environment for OS booting. As such, most UEFI modules inherently involve I/O operations through direct or indirect hardware interaction. To better test the overhead introduced by μEFI, we conduct a mock test that excludes all I/O operations.

In this test, we substitute the I/O operations in the DiskIo module with equivalent memory operations, and the results are shown in Figure 11. Since the test involves only memory copies and function calls, the baseline execution on x86_64 with a file size of 128 bytes takes 5,646 cycles. When FAT is sandboxed, the execution time increases to 26,437 cycles. When both FAT and Mock-DiskIo module are sandboxed, the execution time further rises to 30,870 cycles. Similar results are also observed on the Raspberry Pi 4 platform.

This increase stems from two primary sources: (1) Additional context switches. All service and protocol interface invocations from sandboxed modules must enter the kernel. For instance, FAT uses locks during file operations, implemented via *RaiseTPL* and *RestoreTPL*, which each incur a system call. (2) Interface parameter synchronization. The sandbox manager is responsible for pre-invocation input transfer and post-execution output synchronization. These operations introduce overheads that scale with interface complexity and the buffer size. This partially explains why sandboxing Mock-DiskIo on top of FAT results in less than 20% overhead.

6.4.3 Overhead Breakdown

To assess how different sources of overhead contribute to the overall performance impact, we examine the latency of interface calls involving different parameter types. Four distinct types are included:

- Simple call performs a call without parameters.
- Input object uses an input pointer to pass an integer object.
- Input buffer uses an untyped input pointer to pass a buffer.
- Output buffer passes in the reference of an untyped pointer and retrieves the pointer with buffer allocated by the callee.

The combination of the four types collectively represents all possible interface scenarios. The latency of switching between the caller and the core is not included, as it is roughly the same as the switch latency between the core and the callee.

As illustrated in Figure 12, when running on x86_64 architecture using QEMU/KVM, context switching introduces an average latency of 1,126 cycles. The average latency for DB query is only 55 cycles, as protocol database entries are

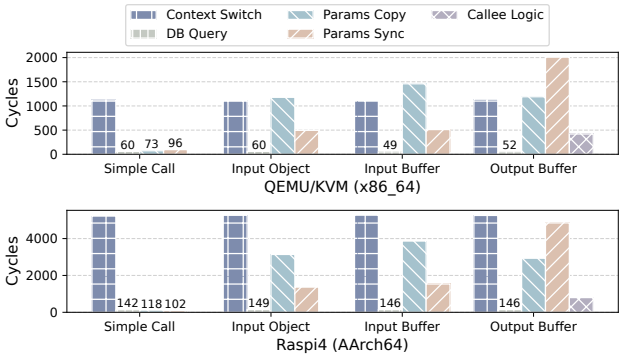


Figure 12: *Overhead breakdown for different kinds of cross-sandbox calls.*

cached after the first analysis, making subsequent queries nearly instantaneous. Even for initial queries, DB query requires only 200–300 cycles. Handling an integer pointer in the input consumes 1,178 cycles, which includes the allocation of transient memory and data transfer. For an input buffer of 800 bytes, the latency increases by an additional 280 cycles. After returning from the callee, 500 more cycles are consumed to free the transient memory. If there exists an output buffer, the overhead of synchronization accounts for the largest proportion, which includes caller memory allocation and data transfer. In the test case of the output buffer, the callee performs a system call to allocate memory for the caller, introducing an additional overhead of 420 cycles.

On the Raspberry Pi 4, the proportional distribution of overhead across different components is similar to that observed with QEMU/KVM. However, context switching incurs an average overhead of 5,262 cycles. This increased cost arises because, for different sandbox calls, the transient return trampoline and jump buffer used for redirection may be allocated to the same physical address. As a result, cache invalidation is required before invoking sandbox functions.

6.5 Memory Consumption

Table 3 summarizes the additional memory consumption of sandboxed modules. When loading a UEFI module, the core allocates at least 32 KB memory. For more complex drivers, such as FAT, the allocation can reach up to 112 KB. On top of this, the additional memory overhead introduced by each sandbox typically remains below 40 KB. Given that modern firmware environments generally support up to 4 GB of physical memory, this overhead is negligible. The additional memory primarily consists of two parts: The memory required for sandbox management and the memory necessary for interface calls.

Part 1: Each sandbox maintains its own page table and independent heap manager, resulting in a memory management overhead of 17 KB, which increases proportionally with future allocation. To safeguard the core against malicious hooks, system tables are not shared among sandboxes which takes up 8 KB. Additionally, the metadata of protocol interfaces

Table 3: *Memory consumption of a sandboxed module.*

Source	Size	Detail usage of the memory
Memory Management	17 KB	Pagetable and heap metadata
Sandbox SystemTable	8 KB	Shadow system table and shadow services
Interface Management	520 B	Every acquired/installed interface
Core-Sandbox Trampoline	128 B	Every installed protocol function
Cross-Sandbox Trampoline	128 B	Every acquired protocol function
Sandbox Stack	32 KB	Every call to the sandbox (transient)
Return Trampoline	64 B	Every call to the sandbox (transient)
Jump Buffer	352 B	Every call to the sandbox (transient)
Call Parameter	Dynamic	Every call from/to the sandbox

and the generated trampolines require 648 bytes of memory.

Part 2: The memory overhead of a sandbox interface call includes a dedicated sandbox stack, a return trampoline for transferring control back to the sandbox manager, a jump buffer for storing core context, and memory allocated for transferring call parameters. Except for the output values, all of these memory occupation are transient, collectively requiring 33 KB.

7 Discussion

μEFI TCB. Our implementation considers the core as the trusted computing base (TCB) and assumes that all preceding phases, including the SEC and PEI phases, are secure. While vulnerabilities exist in the core and these earlier phases, their prevalence is significantly lower compared to other components. Moreover, as modules in the PEI phase primarily handle basic hardware initialization, we implement μEFI to monitor the modules loaded after the core.

Privileged instruction handling. Since UEFI firmware interacts directly with hardware, UEFI modules may rely on privileged instructions to perform their tasks. Executing such instructions in user mode triggers system exceptions. Previous research [1,60,64] on transitioning high privilege components to lower privileges has explored solutions to address this issue. In μEFI, we adopt a similar approach to trap and emulate the execution of privileged instructions.

Usage of μEFI during OS runtime. μEFI primarily works in UEFI boot phases to ensure the modules are loaded and executed safely. Beyond this, the techniques employed by μEFI can also be utilized in the firmware runtime to monitor calls to UEFI runtime services and SMM handlers. For instance, μEFI can serve as a proxy to validate SMM call parameters.

8 Related Work

In addition to UEFI Secure Boot, UEFI incorporates a range of methods and features aimed at enhancing security.

Data flow and control flow protection mechanisms used in operating systems, such as stack guards, address space layout randomization (ASLR), and control flow guard (CFG), can be employed in UEFI [86, 96]. Microsoft, for instance, introduced Enhanced Memory Protection [5], which enforces section-level security by applying execution-protection flags to data sections and read-only flags to code sections. While these methods increase the difficulty of attacks, they fail to address fundamental issues. All modules still operate with high privileges, and their capabilities remain unrestricted. Consequently, attackers can exploit module vulnerabilities to carry out actions such as loading malicious images or hooking malicious functions.

Static analysis and fuzzing of driver source code or binaries are common approaches for uncovering firmware vulnerabilities. Intel's Host Based Firmware Analyzer (HBFA) is an open-source fuzzer targeting EDK II, leveraging manually crafted harnesses to test protocol interfaces [59]. While this design facilitates targeted protocol analysis, its dependence on manual setup and limited input diversity hampers both efficiency and scalability. SPENDER [99] applies static analysis to identify SMM privilege-escalation vulnerabilities in UEFI firmware but faces challenges with precision when analyzing stripped binaries. RSFUZZER [98] adopts a hybrid fuzzing strategy to discover vulnerabilities in SMI handlers, but suffers from the overhead and slow performance of emulation and is incapable of detecting silent vulnerabilities. While these tools seek to discover vulnerabilities, our work takes an orthogonal approach by focusing on attack surface reduction via module isolation, thereby complementing existing analysis-based defenses.

Apple has proposed using OROM Sandbox to constrain the execution of unsigned option ROMs [3]. However, its approach focuses solely on limiting the capabilities of option ROM invocation interfaces and does not address the challenge of achieving module transparency. Rust for UEFI suggests leveraging Rust's memory safety features to enhance UEFI security [97]. However, this requires rewriting the existing codebase. As current firmware codes are predominantly written in C and assembly code, integrating Rust would inevitably introduce significant portions of unsafe code, undermining its benefits in memory safety.

9 Conclusion

μEFI adopts a microkernel architecture to deprivilege and isolate the execution environments of UEFI drivers and applications. To address the challenge of maintaining compatibility with existing firmware modules, μEFI incorporates key techniques such as trampoline injection and protocol analysis. Additionally, it strengthens UEFI security by implementing a seccomp-like mechanism and enabling automated input validation.

Acknowledgments

We sincerely thank our shepherd, Ruslan Nikolaev, and the anonymous reviewers of ATC 2025, whose reviews, feedbacks, and suggestions have significantly strengthened our work. This research was supported in part by National Key Research & Development Program of China (No. 2023YFB3308501), National Natural Science Foundation of China (No. 62202292,62432010,62472279), and research grants from Huawei Technologies. Corresponding author: Jinyu Gu (gujinyu@sjtu.edu.cn).

References

[1] Adil Ahmad, Alex Schultz, Byoungyoung Lee, and Pedro Fonseca. An extensible orchestration and protection framework for confidential cloud computing. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 173–191, 2023.

[2] Ampere. Ampere. https://www.amperecomputing.com, 2025.

[3] Apple. Behind the scenes of ios and mac security. Technical report, Apple, 2019.

[4] Apple. Boot process for an intel-based mac. https://support.apple.com/en-sg/guide/security/sec5d0fab7c6/web, 2021.

[5] Taylor Beebe. Hardening the core: Enhanced memory protections. Technical report, Microsoft, 2023.

[6] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *USENIX annual technical conference, FREENIX Track*, volume 41, pages 10–5555. California, USA, 2005.

[7] Binarly REsearch Team. BRLY-LOGOFAIL-2023-019: Memory corruption vulnerability in dxe driver. Technical report, Binarly, June 2023. Integer overflow on memory allocation size leading to out-of-bounds write operations during PNG file processing in AMI firmware.

[8] A.C. Bomberger, A.P. Frantz, W.S. Frantz, A.C. Hardy, N. Hardy, C.R. Landau, and J.S. Shapiro. The keykos nanokernel architecture. In *Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, pages 95–112, 1992.

[9] Silas Boyd-Wickizer and Nickolai Zeldovich. Tolerating malicious device drivers in linux. In *2010 USENIX Annual Technical Conference (USENIX ATC)*, 2010.

[10] Miguel Castro, Manuel Costa, Jean-Philippe Martin, Marcus Peinado, Periklis Akritidis, Austin Donnelly, Paul Barham, and Richard Black. Fast byte-granularity software fault isolation. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, pages 45–58. ACM, 2009.

[11] CERT Coordination Center. Insecure platform key (pk) used in uefi system firmware signature. https://www.kb.cert.org/vuls/id/455367, August 2024.

[12] Haibo Chen, Xie Miao, Ning Jia, Nan Wang, Yu Li, Nian Liu, Yutao Liu, Fei Wang, Qiang Huang, Kun Li, et al. Microkernel goes general: Performance and compatibility in the {HongMeng} production microkernel. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 465–485, 2024.

[13] Hawk Chen. Uefi and iot: Best practices in developing iot firmware solutions. Technical report, Byosoft, 2017.

[14] Jake Christensen, Ionut Mugurel Anghel, Rob Taglang, Mihai Chiroiu, and Radu Sion. Decaf: automatic, adaptive de-bloating and hardening of cots firmware. In *Proceedings of the 29th USENIX Conference on Security Symposium*, SEC'20, USA, 2020. USENIX Association.

[15] MITRE Corporation. Cve-2020-5376. https://www.cve.org/CVERecord?id=CVE-2020-5376, 2020.

[16] MITRE Corporation. Cve-2020-5378. https://www.cve.org/CVERecord?id=CVE-2020-5378, 2020.

[17] MITRE Corporation. Cve-2021-39297. https://www.cve.org/CVERecord?id=CVE-2021-39297, 2021.

[18] MITRE Corporation. Cve-2021-39299. https://www.cve.org/CVERecord?id=CVE-2021-39299, 2021.

[19] MITRE Corporation. Cve-2021-39300. https://www.cve.org/CVERecord?id=CVE-2021-39300, 2021.

[20] MITRE Corporation. Cve-2021-39301. https://www.cve.org/CVERecord?id=CVE-2021-39301, 2021.

[21] MITRE Corporation. Cve-2022-1890. https://www.cve.org/CVERecord?id=CVE-2022-1890, 2022.

[22] MITRE Corporation. Cve-2022-1891. https://www.cve.org/CVERecord?id=CVE-2022-1891, 2022.

[23] MITRE Corporation. Cve-2022-1892. https://www.cve.org/CVERecord?id=CVE-2022-1892, 2022.

[24] MITRE Corporation. Cve-2022-28858. https://www.cve.org/CVERecord?id=CVE-2022-28858, 2022.

[25] MITRE Corporation. Cve-2022-32482. https://www.cve.org/CVERecord?id=CVE-2022-32482, 2022.

[26] MITRE Corporation. Cve-2022-33209. https://www.cve.org/CVERecord?id=CVE-2022-33209, 2022.

[27] MITRE Corporation. Cve-2022-40516. https://www.cve.org/CVERecord?id=CVE-2022-40516, 2022.

[28] MITRE Corporation. Cve-2022-40517. https://www.cve.org/CVERecord?id=CVE-2022-40517, 2022.

[29] MITRE Corporation. Cve-2023-23904. https://www.cve.org/CVERecord?id=CVE-2023-23904, 2023.

[30] MITRE Corporation. Cve-2023-26589. https://www.cve.org/CVERecord?id=CVE-2023-26589, 2023.

[31] MITRE Corporation. Cve-2023-28031. https://www.cve.org/CVERecord?id=CVE-2023-28031, 2023.

[32] MITRE Corporation. Cve-2023-32466. https://www.cve.org/CVERecord?id=CVE-2023-32466, 2023.

[33] MITRE Corporation. Cve-2023-39538. https://www.cve.org/CVERecord?id=CVE-2023-39538, 2023.

[34] MITRE Corporation. Cve-2023-43626. https://www.cve.org/CVERecord?id=CVE-2023-43626, 2023.

[35] MITRE Corporation. Cve-2024-21829. https://www.cve.org/CVERecord?id=CVE-2024-21829, 2024.

[36] MITRE Corporation. Cve-2024-21871. https://www.cve.org/CVERecord?id=CVE-2024-21871, 2024.

[37] MITRE Corporation. Cve-2024-23980. https://www.cve.org/CVERecord?id=CVE-2024-23980, 2024.

[38] MITRE Corporation. Cve-2024-31154. https://www.cve.org/CVERecord?id=CVE-2024-31154, 2024.

[39] MITRE Corporation. Cve-2024-33656. https://www.cve.org/CVERecord?id=CVE-2024-33656, 2024.

[40] MITRE Corporation. Cve-2024-7344. https://www.cve.org/CVERecord?id=CVE-2024-7344, 2024.

[41] Eclypsium. There's a hole in the boot. https://eclypsium.com/blog/theres-a-hole-in-the-boot/, 2020.

[42] Eclypsium. Firmware security realizations - part 1 - secure boot and dbx. https://eclypsium.com/research/firmware-security-realizations-part-1-secure-boot-and-dbx/, 2022.

[43] BINARLY efiXplorer team. The stack buffer overflow vulnerability leads to arbitrary code execution in uefi application on multiple hp devices. https://github.com/binarly-io/Vulnerability-REsearch/blob/main/HP/BRLY-2021-003-PoC/rsrc/poc_604c4ac862ab8e21d73f3df3977bd09c.gif, 2022.

[44] Úlfar Erlingsson, Martín Abadi, Michael Vrable, Mihai Budiu, and George C Necula. Xfi: Software guards for system address spaces. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, pages 75–88. USENIX Association, 2006.

[45] Alessandro Forin, David Golub, and Brian N Bershad. An i/o system for mach 3.0. Technical Report CMU-CS-91-178, Carnegie Mellon University, Department of Computer Science, 1991.

[46] UEFI Forum. Uefi forum membership list. https://uefi.org/members, 2025.

[47] Dann Frazier. Ubuntu secure boot. https://wiki.ubuntu.com/UEFI/SecureBoot, 2020.

[48] Manuel Fähndrich et al. Language support for fast and reliable message-based communication in singularity os. In *EuroSys*, 2006.

[49] Vinod Ganapathy, Trent Jaeger, and Somesh Jha. Microdrivers: A new architecture for device drivers. In *Proceedings of the 11th USENIX Workshop on Hot Topics in Operating Systems (HotOS)*, 2007.

[50] Tal Garfinkel, Ben Pfaff, Jim Chow, Mendel Rosenblum, and Dan Boneh. Terra: A virtual machine-based platform for trusted computing. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, pages 193–206. ACM, 2003.

[51] Jinyu Gu et al. Epk: Scalable and efficient memory protection keys. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 1–14, 2022.

[52] Jinyu Gu, Xinyue Wu, Wentai Li, Nian Liu, Zeyu Mi, Yubin Xia, and Haibo Chen. Harmonizing performance and isolation in microkernels with efficient intra-kernel isolation and communication. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 401–417, 2020.

[53] Gernot Heiser, Kevin Elphinstone, Ihor Kuz, Gerwin Klein, and Stefan M Petters. Towards trustworthy computing systems: Taking microkernels to the next level. *ACM SIGOPS Operating Systems Review*, 41(4):3–11, 2007.

[54] Jorrit N Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S Tanenbaum. Minix 3: A highly reliable, self-repairing operating system. *ACM SIGOPS Operating Systems Review*, 40(3):80–89, 2006.

[55] HP. Legacy bios boot support removal from hp computers. https://support.hp.com/ca-en/document/c07054518, 2021.

[56] Yongzhe Huang, Vikram Narayanan, David Detweiler, Kaiming Huang, Gang Tan, Trent Jaeger, and Anton Burtsev. Ksplit: Automating device driver isolation. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 613–631, 2022.

[57] Trammell Hudson and Larry Rudolph. Thunderstrike: Efi firmware bootkits for apple macbooks. In *Proceedings of the 8th ACM International Systems and Storage Conference*, SYSTOR '15, New York, NY, USA, 2015. Association for Computing Machinery.

[58] Intel. Removal of legacy boot support for intel platforms. Technical report, Intel, 2023.

[59] Intel. Using hbf to improve platform resiliency, 2025. Accessed: 2025-05-19.

[60] Yuekai Jia, Shuang Liu, Wenhao Wang, Yu Chen, Zhengde Zhai, Shoumeng Yan, and Zhengyu He. {HyperEnclave}: An open and cross-platform trusted execution environment. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 437–454, 2022.

[61] Kaspersky. Mosaicregressor: Lurking in the shadows of uefi. https://media.kasperskycontenthub.com/wp-content/uploads/sites/43/2020/10/07080558/MosaicRegressor_Technical-details.pdf, 2020.

[62] Kaspersky. Cosmicstrand: the discovery of a sophisticated uefi firmware rootkit. https://securelist.com/cosmicstrand-uefi-firmware-rootkit/106973/, 2022.

[63] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al. sel4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 207–220, 2009.

[64] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Krste Asanović, and Dawn Song. Keystone: An open framework for architecting trusted execution environments. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–16, 2020.

[65] Lenovo. Legacy bios boot support removed in lenovo 2020 products. https://support.lenovo.com/sg/en/solutions/ht510878-legacy-bios-boot-support-removed-in-lenovo-2020-products, 2020.

[66] Joshua LeVasseur, Volkmar Uhlig, Jan Stoess, and Stefan Götz. Unmodified device driver reuse and improved system dependability via virtual machines. In *6th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 17–30, 2004.

[67] Jochen Liedtke, Ulrich Bartling, Uwe Beyer, Dietmar Heinrichs, Rudolf Ruland, and Gyula Szalay. Two years of experience with a µ-kernel based os. *ACM SIGOPS Operating Systems Review*, 25(2):51–62, 1991.

[68] Linux. seccomp. https://man7.org/linux/man-pages/man2/seccomp.2.html.

[69] Yandong Mao, Haogang Chen, Dong Zhou, Xi Wang, Nickolai Zeldovich, and M Frans Kaashoek. Software fault isolation with api integrity and multi-principal modules. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 115–128, 2011.

[70] Microsoft. Boot to uefi mode or legacy bios mode. https://learn.microsoft.com/en-us/windows-hardware/manufacture/desktop/boot-to-uefi-mode-or-legacy-bios-mode?view=windows-11, 2021.

[71] Microsoft. Cve-2022-21894. https://www.cve.org/CVERecord?id=CVE-2022-21894, 2022.

[72] Microsoft. Cve-2022-21894 security update. https://msrc.microsoft.com/update-guide/vulnerability/CVE-2022-21894, 2022.

[73] Microsoft. Secure boot. https://learn.microsoft.com/en-us/windows-hardware/design/device-experiences/oem-secure-boot, August 2023.

[74] Microsoft. Arm-based surface devices faq. https://learn.microsoft.com/en-us/surface/surface-arm-faq, 2024.

[75] Vikram Narayanan, Abhiram Balasubramanian, Charlie Jacobsen, Sarah Spall, Scott Bauer, Michael Quigley, Aftab Hussain, Abdullah Younis, Junjie Shen, Moinak Bhattacharyya, et al. {LXDs}: Towards isolation of kernel subsystems. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 269–284, 2019.

[76] Vikram Narayanan et al. Redleaf: Isolation and communication in a safe operating system. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2020.

[77] Ruslan Nikolaev and Godmar Back. Virtuos: An operating system with kernel virtualization. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, pages 116–132. ACM, 2013.

[78] NVIDIA. Dgx h100/h200 firmware update guide. https://docs.nvidia.com/dgx/dgxh100-fw-update-guide/.

[79] NVIDIA. Updating firmware for nvidia bluefield-3 dpu. https://network.nvidia.com/support/firmware/bluefield3/.

[80] Raspberry Pi. Raspberry pi 4 specification. https://www.raspberrypi.com/products/raspberry-pi-4-model-b/specifications/.

[81] ESET Research. Lojax: First uefi rootkit found in the wild, courtesy of the sednit group. https://www.welivesecurity.com/2018/09/27/lojax-first-uefi-rootkit-found-wild-courtesy-sednit-group/, 2018.

[82] ESET Research. Under the cloak of uefi secure boot: Introducing cve-2024-7344. https://www.welivese

curity.com/en/eset-research/under-cloak-uefi-secure-boot-introducing-cve-2024-7344/, 2024.

[83] Samsung. Galaxy book go 5g. https://www.samsung.com/us/computing/galaxy-books/galaxy-book-go/galaxy-book-go-5g-14-256gb-silver-att-np545xla-ka1tt/, 2025.

[84] Martin Smolár. Blacklotus uefi bootkit: Myth confirmed. https://www.welivesecurity.com/2023/03/01/blacklotus-uefi-bootkit-myth-confirmed/, 2023.

[85] Michael M Swift, Steven Martin, Henry M Levy, and Susan J Eggers. Nooks: An architecture for reliable device drivers. In *Proceedings of the eighteenth ACM symposium on Operating systems principles*, pages 33–48. ACM, 2002.

[86] TianoCore. Enable aslr for uefi in edk ii. https://edk2-docs.gitbook.io/a-tour-beyond-bios-mitigate-buffer-overflow-in-ue/address_space_layout_randomization/enable_aslr_for_uefi_in_edkii.

[87] TianoCore. *EDK II Driver Writer's Guide for UEFI 2.3.1*, 2018.

[88] TianoCore. Edk ii project. https://github.com/tianocore/edk2, 2019.

[89] Kevin Tremblay. Updated: Uefi signing requirements. https://techcommunity.microsoft.com/t5/hardware-dev-center/updated-uefi-signing-requirements/ba-p/1062916, 2021.

[90] Kevin Tremblay. New uefi ca memory mitigation requirements for signing. https://techcommunity.microsoft.com/blog/hardwaredevcenter/new-uefi-ca-memory-mitigation-requirements-for-signing/3608714, August 2022.

[91] UEFI Forum. *Unified Extensible Firmware Interface Specification Version 2.3.1*. UEFI Forum, 2011.

[92] UEFI Forum. *Unified Extensible Firmware Interface Specification Version 2.11*. UEFI Forum, 2024.

[93] Anjo Vahldiek-Oberwagner et al. Intra-unikernel isolation with intel memory protection keys. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pages 1–15, 2020.

[94] Chad Duffey Vinay Pamnani, Paolo Matarazzo. Secure the windows boot process. https://learn.microsoft.com/en-us/windows/security/operating-system-security/system-security/secure-the-windows-10-boot-process, July 2024.

[95] Robert Wahbe, Steven Lucco, Thomas E Anderson, and Susan L Graham. Efficient software-based fault isolation. In *Proceedings of the fourteenth ACM symposium on Operating systems principles*, pages 203–216. ACM, 1993.

[96] Jiewen Yao. Cet in smm. https://github.com/tianocore/tianocore.github.io/wiki/CET-in-SMM.

[97] Jiewen Yao. Enabling rust for uefi firmware. Technical report, Intel, 2020.

[98] Jiawei Yin, Menghao Li, Yuekang Li, Yong Yu, Boru Lin, Yanyan Zou, Yang Liu, Wei Huo, and Jingling Xue. Rsfuzzer: Discovering deep smi handler vulnerabilities in uefi firmware with hybrid fuzzing. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 2155–2169. IEEE, 2023.

[99] Jiawei Yin, Menghao Li, Wei Wu, Dandan Sun, Jianhua Zhou, Wei Huo, and Jingling Xue. Finding smm privilege-escalation vulnerabilities in uefi firmware with protocol-centric static analysis. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 1623–1637. IEEE, 2022.

PageFlex: Flexible and Efficient User-space Delegation of Linux Paging Policies with eBPF

Anil Yelam, *Google*
Kan Wu, *Google*
Zhiyuan Guo, *UC San Diego*
Suli Yang, *Google*
Rajath Shashidhara, *University of Washington*
Wei Xu, *Google*
Stanko Novakovic, *Google*
Alex C. Snoeren, *Google and UC San Diego*
Kimberly Keeton, *Google*

Abstract

To increase platform memory efficiency, hyperscalers like Google and Meta transparently demote "cold" application data to cheaper cost-per-byte memory tiers like compressed memory and NVMe SSDs. These systems rely on standard kernel paging policies and mechanisms to maximize the achievable memory savings without hurting application performance. Although the literature promises better policies, implementing and deploying them within the Linux kernel is challenging. Delegating policies and mechanisms to user space, through `userfaultfd` or library-based approaches, incurs overheads and may require modifying application code.

We present PageFlex, a framework for delegating Linux paging policies to user space with minimal overhead and full compatibility with existing real-world deployments. PageFlex uses eBPF to delegate policy decisions while providing low-overhead access to in-kernel memory state and access information, thus balancing flexibility and performance. Additionally, PageFlex supports different paging strategies for distinct memory regions and application phases. We show that PageFlex can delegate existing kernel-based policies with little (< 1%) application slowdown, effectively realizing the benefits of state-of-the-art policies like Hyperbolic caching [18] and Leap [12] prefetching, and unlocking application-specific benefits through region- and phase-aware policy specialization.

1 Introduction

Memory cost as a fraction of hyperscalers' total cost of ownership (TCO) has been growing at an alarming rate [25, 32, 48]. Offloading cold memory to cheaper cost-per-byte memory tiers like compressed memory and NVMe SSDs offers a promising way to increase platform memory efficiency. Recent hyperscale deployments like Google's `g-swap`[1] [32] and Meta's Transparent Memory Offloading (TMO) [48] demonstrate that significant (20–30%) memory savings can be achieved with minimal impact to application performance. These systems rely on the OS's page reclamation mechanisms to offload "cold" application pages (pages that haven't been accessed recently) and minimize the chances of costly page refaults from the swap backend, which directly impact application performance. Consequently, correctly inferring what pages are cold and whether to offload them—i.e., the page offload or reclamation[2] policy—is crucial to improving memory savings without hurting application performance. Similarly, prefetching the offloaded pages before they are reaccessed can hide the cost of refaults, making prefetching equally important.

Existing deployments leave the critical page offloading decisions in the kernel: they maintain page ordering based on page access information using kernel mechanisms, and use control loops in user space to determine the threshold for offloading pages based on application impact. For page ordering, they rely on traditional Linux kernel LRU mechanisms like active/inactive lists [48]; multi-generational LRU (MGLRU) [6]; or implicit ordering through per-page "age" fields [32], where orderings are updated based on page access information that is checked periodically [32]. However, many more sophisticated eviction algorithms have been proposed in recent work [15, 18, 35, 42, 50] that could potentially further improve memory savings. Our measurements show that there is a significant gap between LRU and the optimal decisions for real-world accesses from page access traces (§2.3). Similarly, for prefetching, existing systems rely on Linux's default read-ahead, while more advanced prefetching policies like Leap can improve performance by 7—75% [12].

Linux currently offers limited paging policy choices, and improving these policies or implementing new ones in kernel code brings significant challenges. Deploying kernel changes across a large fleet is a slow and risky process, and experimental or application-specific policies are harder to upstream [28, 34]. Existing user-space designs, such as those that rely on Linux `userfaultfd` [3, 38, 40, 54] or custom

[1] We adopt [48]'s terminology for Google's swap infrastructure.

[2] We use the terms *offload* and *reclaim* (Linux term) synonymously.

libraries [26, 41, 45, 53], can *externalize* (i.e., allow out-of-kernel implementations for) paging, but introduce new problems. By migrating the entire paging infrastructure to user space, `userfaultfd` dramatically increases the cost of refaults (e.g., by more than 50% for zswap (§ 2.4)). Additionally, it imposes a high deployment barrier by requiring user-level swap implementations—along with the effort to maintain them—since it cannot leverage the kernel's mechanisms.

Custom memory management libraries such as DiLOS or Application-Integrated Far Memory (AIFM) [23,26,41,45,53] completely avoid kernel paging for flexibility and lower paging overhead, but pose similar challenges. They often require modifying applications, and while their slow path (i.e., refault from swap) is generally much faster, they introduce considerable performance overhead even when no memory is offloaded, as they force a custom interface [52]. Both `userfaultfd` and custom libraries introduce significant operational overhead at scale, where maintaining stability, efficiency, and seamless integration with existing infrastructure is critical. Thus, in this paper, we ask the following question: *Can we delegate paging policies to user space without slowing down the application or compromising compatibility with existing deployments (i.e., avoiding application modifications and the need to reimplement swap infrastructure)?*

Our solution is **PageFlex**, a framework designed to delegate the paging policy outside of the kernel, while keeping it compatible with existing kernel-based swap deployments like `g-swap` [32] and TMO [48]. To achieve these goals, PageFlex only delegates non-performance-critical policy decisions made for prefetching and proactive reclamation outside the kernel code. Enabling policies requires tracking the state of memory and how it is being accessed (e.g., paging activity and page access information), which could introduce considerable overhead if significant kernel state needs to be exported to and processed in user space. To address this issue, PageFlex uses eBPF programs [10] to provide a flexible, low-overhead view of in-kernel memory state to externalized policy code.

At a high level, PageFlex implements policies by observing the changes to memory state and access information, and making decisions based on them. To track memory state, PageFlex traces various paging events (page status changes like page allocations, frees, swap-ins and -outs) and access signals (through page faults and periodic checks on access bits) and exposes them to the policies implemented as eBPF-based event handlers. To enable policies to efficiently track and maintain metadata based on observed state, PageFlex provides event handlers with safe, overhead-free access to a small amount of in-kernel metadata for each page. Policies subscribe to only the events they need, providing handlers to process them and optionally communicate with user-level code for more complex decision making. Any policy decisions made in user space are enforced by PageFlex through paging hints (page in or out) using the `madvise` syscall interface.

To make it easier to write policies, PageFlex provides generic reclamation and prefetching policy models that support a variety of policies with a simple interface. Our reclamation model generalizes `g-swap`'s per-page, age-based ordering to instead employ generic, per-page weights. In this model, most existing eviction heuristics can be implemented as custom weight-update functions that consume the page-access information PageFlex periodically exports. Our prefetching model enables customized trend detection, allowing the implementation of state-of-the-art policies like Hyperbolic caching [18] and the Leap [12] prefetching. Our re-implementations require no changes to the kernel and only a handful of lines of policy code (17 for Hyperbolic caching and 21 extra lines for Leap in addition to 160 borrowed from its original implementation [1]). Going beyond a single static policy for an application, PageFlex supports specialization for memory regions within a process and runtime policy adjustments.

We evaluate PageFlex with a variety of workloads and policies. PageFlex introduces very little additional application slowdown compared to kernel implementations of the same policies. For page reclamation, we show that Redis and synthetic benchmarks run at most 1% slower when using PageFlex's LRU implementation as compared to `g-swap`'s functionally-equivalent but kernel-based variant. For prefetching, a PageFlex-based implementation of Linux's read-ahead policy provides similar benefit to the native in-kernel implementation: when measuring the page refault rate from an SSD-like swap backend, Linux's kernel-based read-ahead is 82% faster than no prefetching, while PageFlex delivers a 76% improvement with minimal (0.8%) overhead when there are no prefetching benefits.

Going beyond these default policies, we implement various policies from the literature and demonstrate their memory savings and performance benefits. PageFlex's LFU policy offloads up to 2× more memory than LRU at any given performance slowdown for a synthetic LFU-friendly workload. Similarly, PageFlex-based Hyperbolic caching allows up to 5% more memory savings for real-world workload traces. PageFlex-based Leap prefetching improves the refault rate with strided accesses by 75.4% over the Linux default, which cannot capture strided patterns. Our workload-aware, specialized policies show up to 36% more memory savings for a key/value store, and 6% for the GAPBS PageRank workload, compared to LRU for the same performance target. Moreover, PageFlex enables these flexibility and performance benefits despite modest overheads from its supporting infrastructure. For example, compared to the in-kernel implementation of `g-swap`, PageFlex's periodic page table scans are 17% slower due to eBPF overheads, and enforcing paging actions from user space through syscalls takes 14% longer.

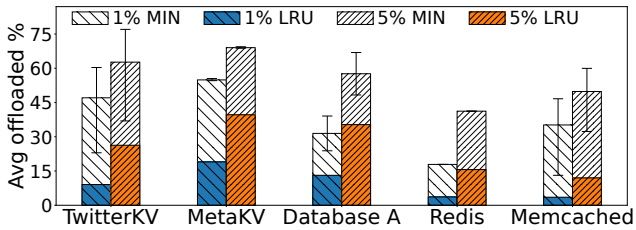


Figure 1: Average memory offloaded (% of total memory usage) with periodic LRU eviction (colored bars) simulated on page access traces of real workloads (detailed in Table 2) at two different allowed page refault rates (1% and 5%), overlaid against Belady's MIN. The white space in each bar shows the opportunity for more memory offloaded at the same performance target.

2 Background and Motivation

We begin by motivating the need for a platform to easily explore novel paging policies. We then discuss the requirements for investigating paging policies in the data center. We describe paging alternatives, including kernel-based, user-level, and library-based implementations, showing that current solutions fall short of meeting these requirements. Additionally, we survey paging policies from the literature and provide a background on eBPF.

2.1 Motivation for policy investigation

We begin by exploring the potential for improving on standard reclamation policies like LRU by comparing its effectiveness against an optimal policy based on Belady's MIN [16]. We evaluate these policies by simulating the proactive reclamation approach used in hyperscaler data centers using the traces described in Table 2. These deployments [32,48] proactively offload application pages to swap devices like NVMe SSDs [48] or in-memory compressed caches like zswap [7]. Rather than reclaiming pages in response to memory pressure, which can be disruptive to application performance (since reclamation happens on the critical path of memory allocation), these systems proactively reclaim data in a manner that limits the impact on application performance.

Figure 1 shows the difference in the fraction of memory offloaded for two different performance constraints. To simulate the policies, a target amount of memory is offloaded at the end of each interval (similar to TMO [48] and `g-swap` [7]) based on each algorithm (i.e., ordered by the last access time for LRU and the next access time for MIN). As a proxy for application impact, we use `g-swap`'s *promotion rate*[3], the rate of page refaults normalized to the working set, averaged for all intervals. We vary the memory offload targets and measure the largest target that results in a refault rate below 1% and 5%. Even at these low refault rates, the gap between LRU and MIN is significant for all traces, ranging from 14–37% at 1% refault rate and 22–38% at 5%, strongly suggesting that there is opportunity to design better eviction algorithms. Our goal is to make it easier to experiment with and adopt such algorithms—both existing research algorithms and novel ones—to close this gap.

[3]We use the terms *promotion rate* and *refault rate* synonymously.

Policy Infrastructure	Kernel	User-faultfd	Library	PageFlex
App portability	✓	✓	×	✓
Debuggability	×	✓	✓	✓
Reuse swap stack	✓	×	×	✓
Faster rollouts	×	✓	✓	✓
Min. overheads	✓	×	×	✓
Fault isolation	×	✓	✓	✓

Table 1: Comparing various paging designs.

2.2 Hyperscaler paging goals

When experimenting and adopting a new paging policy, hyperscalers have several goals. Application portability without source code modifications increases ease of adoption for new policies. Hyperscalers need the ability to easily develop and debug a new policy implementation. To aid in this goal, they want to reuse existing swap stack implementations for components that are not critical to the policy (e.g., swap device implementations, swap device sharing, swap accounting). Once policies are developed, hyperscalers need the ability to roll them out to their fleet of servers; given fleet sizes, deployment speed of new policies (and any fixes to critical bugs) is critical. Once deployed, policy enforcement should operate with minimal overheads to limit potential regressions in application performance. Finally, if the policy has bugs, their impact should be limited in scope. Table 1 summarizes various paging designs with respect to these criteria. In the subsequent sections, we dig into these alternatives in more detail.

2.3 Kernel-level paging

Page reclamation. Page faults can be expensive (e.g., 4–10 μs for zswap [32] to tens/hundreds of μs for SSD), so minimizing refaults is crucial to avoid hurting application performance. The page reclamation policy determines what pages to reclaim and when, relying on past page access information to predict future demands. Reclamation policies rely on two key mechanisms: (1) per-page measures to order pages based on their re-access probability and (2) a control loop that determines which pages to offload by enforcing a threshold based on the perceived application performance impact.

Current hyperscaler reclamation reuses existing Linux kernel mechanisms based on various LRU approximations to

maintain page ordering. TMO [48] reuses Linux's active/inactive lists to track pages for on-demand reclamation. While efficient, these lists track page coldness at a very coarse granularity (active vs. inactive) and do not track page access counts. Recent improvements like MGLRU [6] enable a configurable number of levels, allowing for more fine-grained page coldness tracking. MGLRU also introduces page "aging" scans to scan page tables and constantly move pages from younger to older generations, rather than relying on memory pressure events. Another approach, employed by `g-swap` [32], maintains an implicit ordering based on a per-page coldness "age", which tracks time since the last access for every page. Similar to MGLRU, `g-swap` employs aging scans using its *kstaled* kernel thread; the resulting aging allows bucketing pages by coldness at a fine granularity. `g-swap` relies on a pre-configured age threshold to trigger reclamation.

To determine how many pages to offload, hyperscaler systems employ control loops to constantly monitor application performance impact metrics (e.g., `g-swap`'s refault rate or TMO's pressure stall information (PSI)) and adjust the offloading threshold accordingly. These control mechanisms are implemented in user space, and they use cgroup knobs like `memory.reclaim` or `g-swap`'s reclaim age threshold to inform kernel mechanisms.

Page prefetching. Prefetching offloaded pages that are likely to be accessed into local memory hides the high cost of refaults, which is especially useful for high-latency swap backends like SSDs. Current prefetch policies detect historical patterns in refaulted pages to guess simple trends like sequential or strided accesses. For example, Linux uses a simple read-ahead policy that looks for successive refaults to adjacent pages for sequential patterns; even minor levels of noise can throw off its trend detection. Prefetch aggressiveness, the number of pages to prefetch, depends on the number of hits seen for the previously prefetched pages.

As with reclamation policies, we want to support easy adoption of experimental policies. For example, the Leap [12] policy uses a more complex trend detection that can handle both sequential and strided patterns even in the presence of noise. Leap can achieve up to 0.7—37.5% higher prefetching coverage over Linux read-ahead, while maintaining similar accuracy for some workloads [12].

Challenges with kernel-based policies. Experimenting and adopting new policies written in the kernel is challenging. Deploying kernel changes across a large fleet is difficult, analogous to what warehouse-scale operators like Google already observed for CPU scheduling [28] and networking stacks [34]. Kernel rollouts even at a monthly cadence are "not well-tolerated" at Google [28]. This prevents both fast roll-outs for any critical bugs as well as rapid iteration with new policies. Furthermore, experimental policies or policies tailored to specific workloads are harder to upstream, requiring maintaining separate forks for new policies and a constant developer effort to keep them in sync with the upstream kernel. These challenges motivate the need for moving paging policies out of the kernel.

2.4 User-level paging

User-level paging is a well-known technique used to transparently manage OS pages from user space [2, 20, 27, 38, 40, 43, 54], typically supported by an interface like Linux's `userfaultfd` [3]. With `userfaultfd`, a user-space handler process can register to manage sections of an application's address space, causing the kernel to synchronously route all page operations (page faults, map, unmap, `madvise`, etc.) for these regions to user space. Combined with the ability to write-protect and unmap the pages at any time (using `madvise`), the handler can transparently back the registered memory regions with a custom swap mechanism. Policies can then be implemented in the handler process without any kernel involvement. However, this approach has several limitations, both in terms of overheads and compatibility.

First, by routing page faults through the user-space handler, the approach increases refault costs, directly impacting application performance. This overhead comes from the additional user-kernel transitions and the extra copy required to move the page data from the buffer provided by the user-space handler into the destination page. To quantify this overhead, we measure the cost of a zero-page fault (a fault that does not require servicing by the swap backend) for both `userfaultfd` and the kernel, focusing on the per-page fault overheads due to the interface. Figure 2 shows the median cost; while a kernel zero-page fault takes 1.2 μs, the same fault handled by `userfaultfd` takes 5.6 μs, with up to 4 μs spent waiting for the handler. For reference, fetching a page from zswap backend takes 6 μs at the median; `userfaultfd` would add more than 50% overhead to this cost. In §6.2, we show that overheads of even a few microseconds can entirely erase the performance benefit gained by avoiding refaults.

Second, memory regions managed through `userfaultfd` require custom swap backend implementations in the handler. Normally this is an advantage, as it allows for non-standard swap backends like cloud-based storage [2]. However, existing large-scale deployments like `g-swap` and TMO heavily rely on the kernel's swap subsystem for paging mechanisms, swap device implementations (like zswap [7]), swap device sharing, cgroup-based memory and swap accounting, etc. Moving this functionality outside the kernel requires significant development effort.

2.5 Custom library-based approaches

Flexibility in memory management has been well-explored in the RDMA-based far memory space [13, 19, 26, 39, 41, 45, 47, 53, 56]. Systems like DiLOS [53] and AIFM [41] and its derivatives [26, 45] completely bypass the OS and provide libraries/user-level runtimes for applications to use directly.

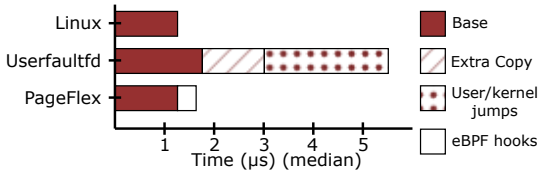


Figure 2: Cost of zero-page fault served from the kernel and from the `userfaultfd` handler.

By parceling everything within the application, they also limit the blast radius of any policy implementation bugs to the application.

However, library-based approaches introduce a custom memory interface that requires application modifications. They also introduce performance overhead even without any memory offloading, as they bypass the highly-efficient address translation hardware (TLB & MMU) and introduce additional dereferences for memory indirection in software, resulting in up to a 40% slowdown [45]. Note that, unlike the overhead caused by refaults (which can be traded off with memory savings), this overhead cannot be avoided. Like user-level paging, library approaches face compatibility challenges, as they, too, lift the entire swap infrastructure into user space and do not offer a clear solution for swap device sharing.

2.6 eBPF

eBPF (Extended Berkeley Packet Filter) allows user-defined programs to run within the Linux kernel, offering the flexibility to observe kernel activities and even customize their behavior without any overhead [10]. These programs are written in a C-like language, compiled into bytecode, and then verified by the kernel to ensure they are safe to execute. The programs are then "hooked" to specific points in the kernel (e.g., tracepoints, kprobes, etc.), triggering synchronous execution at those points. Data can be shared efficiently between eBPF programs and user space using eBPF maps. For safety, access to kernel data structures is not permitted unless explicitly whitelisted through specific access points, such as writable tracepoints [5]. eBPF has been widely used to unlock customization for various kernel subsystems like packet processing [24,46], file systems [17], process scheduling [28,31], storage [51,55], kernel locks [37] and even prefetching [21].

3 PageFlex Design

In this section, we first discuss our design goals and how they motivate PageFlex's design, followed by a detailed overview of PageFlex's mechanisms.

3.1 Design Goals

Our key goal is to enable writing new paging policies while meeting the hyperscaler paging goals discussed in § 2.2. More concretely, our solution requires the following considerations:

1. Compatibility: Our solution must coexist with current kernel-based policies and swap backends; i.e., some workloads running with the default policies and others using PageFlex but sharing the same swap backend.

2. Fail-safety: Failure of any PageFlex components (e.g., due to policy bugs) should not affect other workloads or even crash the apps whose policy is handled. This allows for safe, non-disruptive policy upgrades that are crucial for warehouse-scale deployments [28].

3. Minimal overheads: Any overheads introduced by the PageFlex infrastructure must be minimal. More importantly, overheads that directly impact the application performance must be negligible (i.e., background overheads are more tolerable).

To achieve these goals, PageFlex only externalizes non-critical policy decisions for prefetching and proactive reclamation to user space, while the kernel continues to provide core mechanisms for page access tracking, on-demand page fault handling and page reclamation. This division of labor achieves compatibility (1), unlike `userfaultfd` and library-based approaches. Also, because policy decisions are not critical to application functionality, PageFlex failures will not crash the application (2). Finally, PageFlex uses eBPF programs to provide a flexible, low-overhead view of in-kernel memory state to the externalized policy code. More specifically, policies run their performance-critical code inline with the kernel paths using eBPF to flexibly process the page events as they occur, without incurring the overheads of exporting large volumes of page event metadata to user space (3).

3.2 System Overview

Figure 3 provides an overview of PageFlex's design. At a high level, PageFlex consists of a kernel component (left half of the figure) with eBPF programs where policies run in-band with the kernel events to get a low-overhead memory view and arrive at paging decisions. This information is then shared out-of-band with a user-level component (right half) which is used to enforce these decisions and implement any non-performance critical policy infrastructure.

3.2.1 In-kernel policy execution

To provide a memory view, PageFlex tracks various basic paging events with explicit kernel tracepoints. Policies then subscribe to (any subset of) these events and provide event handlers that PageFlex runs synchronously with the paging events in the kernel as eBPF programs. We discuss these components below.

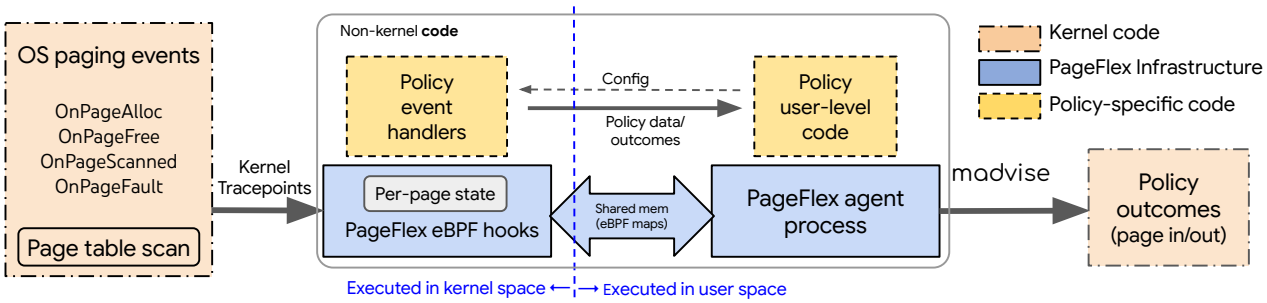


Figure 3: An overview of PageFlex's design showing various components. The policy control flows from left to right. The box in the middle indicates the components that support the policy code written outside the kernel code.

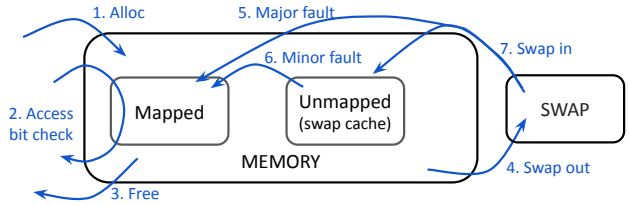


Figure 4: Paging paths (blue arrows) that PageFlex's events track for each page, as the page transitions between various memory states and the swap backend.

Kernel paging events. The left-most box in Figure 3 shows the set of in-kernel page events PageFlex provides for policy handlers to subscribe to. (Figure 4 shows all the page state transitions these events cover.) `OnPageAlloc` (arrows 1 and 7 in Figure 4) and `OnPageFree` (arrows 3 and 4) events are raised when a page enters or leaves the memory, either due to allocation or freeing by the application or being moved in from/out to the swap backend. These events are useful for keeping track of the current memory and swap state. For access information, PageFlex tracks two sets of events: for unmapped pages, `OnPageFault` (arrows 5 and 6) is raised when an unmapped page is requested by the process through a page fault (regardless of whether the page is mapped from the swap backend or the swap-cache)—it reports both swap-cache hit and miss events as both are useful access signals for prefetching (§2.3). For mapped pages, PageFlex periodically collects page access bits similar to `g-swap`. `OnPageScanned` (arrow 2) is raised for each page with information on whether it is accessed. Combined, these events provided the necessary information for all the policies we implemented (§6.3).

eBPF-based custom event handlers. Policies subscribe to (any subset of) the above events and provide custom handler functions that PageFlex runs inline with the events as eBPF programs. Allowing policy-specific handlers with eBPF avoids the need for indiscriminately tracing and communicating everything to user space. In fact, we will show that most policies can make their decisions in their event handlers and only export digested data (like policy outcomes to enforce or statistics for policy tuning in user space). The event handlers run synchronously with paging paths, potentially impacting application performance (e.g., slower page fault handling due to `OnPageFault` events). However, the overhead of eBPF programs is very little (<50 ns per eBPF invocation), allowing PageFlex to use them aggressively even in high-frequency paths. (We quantify the overhead of eBPF in §6.5.)

Low-overhead per-page state. Policies generally need to maintain per-page metadata as they learn from page accesses and make decisions over time. For example, `g-swap` keeps track of page ages for LRU ordering. As we will see in §4.1, most reclamation policies assign per-page scores that are updated over time based on page accesses. Such state can be maintained in user-space data structures (e.g., a hash table) or eBPF maps, however, these introduce overheads (i.e., lookups, additional memory). Fortunately, all the events traced in PageFlex are page-specific and the kernel page structures are already available for these events. Thus, PageFlex reserves a small space (4 bytes) in the kernel page structure for each page and provides safe and exclusive read/write access from the event handlers. This space also allows PageFlex event handlers to mark or track events for only a subset of pages based on policy-specific criteria. For example, online learning policies like LRB [42] may want to sample a subset of pages and track only those pages through their lifetime (e.g., send their access events to user space) for training their models. The state is limited (a 4-B state per each page incurs a fixed 0.1% memory cost, same as `g-swap`), however it is more than enough for most policies. By default, the state is reset when the page is swapped out, but policies can optionally persist the old state across reclamations if needed (at an additional memory cost of 0.1% incurred for swapped-out pages as well).

3.2.2 User-level policy control and enforcement

PageFlex employs a dedicated user-level agent process to initialize and run policies for the applications that opt in for PageFlex. The agent process loads the policy's eBPF handlers and lets policies configure them through eBPF maps. Policies

```
// Reclaim interface (per-page)
struct {} state; // 4B max
UpdateWeight(state, accessBit): {}
OnSwapIn(state, oldState): {}
  default { state = 0 }
OnSwapOut(oldState): {}
  default { oldState = 0 }

// Prefetch interface (per-page fault)
PredictTrend(page, isHit): {}
```

```
// Hyperbolic caching
struct { u8 hits; u8 age; } state;
UpdateWeight(state, accessBit):
    if (accessBit) state.hits++;
    state.age++;
    return (state.hits / state.age);

OnSwapIn(state, oldState): default
OnSwapOut(oldState): default
```

```
// Linux read-ahead
PredictTrend(page, isHit):
  if (isHit and page in prevWin) nHits++;
  // re-use linux function
  newWin = __swapin_nr_pages
        (lastPage, page, nHits);
  prevWin = newWin; lastPage = page;
  if (newWin) nHits = 0;
  return newWin; // to prefetch
```

Figure 5: PageFlex simplified interface for most policies (left) and example policy implementations (center and right).

can push events (e.g., page in/out decisions) to user space and handle them asynchronously with user-level handlers (e.g., issuing paging actions). Additionally, policies can perform any non-critical and complex functionality in their user-level code using the information passed from the event handlers, e.g., control loops used in page reclamation (§ 2.3). PageFlex enables two-way shared-memory communication between user space and the eBPF event handlers with eBPF maps.

Enforcing policy outcomes. PageFlex acts on the policy outcomes using `madvise` hints from the user-level agent to page in (for prefetching) or page out (for reclamation). Specifically, the agent process uses the `process_madvise()` wrapper to issue `madvise` hints against a different (process') address space. Using a hardened interface like `madvise` lets us avoid introducing new kernel infrastructure while still making the decisions outside the kernel. However, this introduces additional overhead due to extra user/kernel crossings when compared to the policy decisions made in pure kernel designs. E.g., in `g-swap` a kernel thread, *kreclaimd*, reclaims pages. We amortize these overheads by issuing `madvise` calls on batches of pages. (As shown in the results in §6.5, both prefetching and proactive reclamation paths do not directly affect application latency, allowing us to batch aggressively.)

Policy isolation. As in ghOSt [28], PageFlex provides separate "enclaves" for running different policies simultaneously for different groups of processes. Each enclave binds to a single Linux kernel cgroup and applies a single policy to all the processes in that cgroup. The enclave separation is provided in the event handlers (in eBPF), user-space code and in the communication between the two. This enables targeted policy application for select workloads while the rest continue to use the default kernel policy (while sharing the same swap backend). And, if at any point PageFlex fails (or during upgrades), the cgroup bindings are gracefully released without any effect on the attached processes. Additionally, because PageFlex uses the kernel's swap subsystem, existing kernel-based reclamation like `g-swap` can be configured to take over on failure, a feature `usefaultfd`-based designs cannot provide.

4 Policies in PageFlex

In this section, we first show how PageFlex enables writing prefetching and reclamation policies with varying degrees of specialization, followed by a discussion on the limits of its policy expressibility.

4.1 Unlocking policy customization

Proactive reclamation. We start with a goal of unlocking various eviction algorithms for hyperscaler reclamation systems like `g-swap` (described in §2.3) with PageFlex. To that end, PageFlex replaces `g-swap`'s per-page LRU age that defines eviction ordering with generic per-page *weights* that are policy-specific. PageFlex then lets the policies provide custom functions to update these weights—and the way weights are updated can implement a variety of policies. For example, an LRU ordering maintains the time since last access as the weight while, for LFU, it is the number of hits. Many other traditional eviction heuristics such as LRU-K [36], LRFU [33], LIRS [30], Hyperbolic [18], LHD [15], etc. can be similarly expressed as simple functions of the page accesses. Even more complex ones based on multiple queues [50] can be modeled by partitioning the weight space to represent different queues (our 4-B per-page state allows a 32-bit weight which is large enough in practice). PageFlex runs these policy-specific functions through its (eBPF) event handlers while maintaining the page weights in the low-overhead per-page state (described in §3.2), thereby achieving the flexibility to implement various eviction algorithms for `g-swap` without kernel changes.

Along with policy flexibility, PageFlex's mechanisms can also easily support policy-agnostic infrastructure that is required to support the reclamation policies. Like `g-swap`, PageFlex employs periodic page-table scans to collect page access bits and allow page weight updates, exports page-weight distributions using eBPF maps, and runs the control loop in the user-level agent to determine the eviction threshold. To simplify policies, PageFlex abstracts away this common infrastructure and provides the interface shown in Figure 5 for most polices (like above) that can work with simple update functions. The only required function is `UpdateWeight`, which is called periodically for each page after checking its access bit (provided as `accessBit`) and returns a new weight (see the example

for Hyperbolic caching [18] in the center). The function can maintain a small state (`state`) persisted as long as the page is in memory. By default, the `state` is reset once the page is swapped out. This works for most policies as they do not need the access history after a page gets reclaimed. Some policies, however, remember the order of recently reclaimed pages to maintain a ghost queue [50], while a learning-based policy may want to remember the tracked pages across reclamations, and `g-swap` remembers the page age when it was reclaimed, etc. To enable such policies, PageFlex optionally lets the policies customize the behavior at `OnSwapIn` and `OnSwapOut` to persist and retrieve the state (`oldState`). More specialized policies, however, may bypass this simplified interface and directly work with the core paging events described in § 3.2.

Prefetching. As discussed in §2.3, prefetching policies require page refault history for trend detection and predicting the next pages in the trend to prefetch, and the number of prefetching hits (through minor-faults) to adjust the predictions. PageFlex can provide both the signals with `OnPageFault` events. Similar to reclamation, PageFlex takes care of the policy-agnostic support like enforcing prefetching decisions while only requiring the policies to define a `PredictTrend` function, where they can detect any trend and return the pages to prefetch. The function could be marked to run directly in the eBPF event handler or in user-level code depending on complexity. Figure 5 (right) shows the Linux read-ahead policy implemented with PageFlex's simplified prefetching interface.

4.2 Application-specific policies

PageFlex further supports applying distinct paging policies to different continuous memory regions and execution phases within a user application. This capability is particularly beneficial for workloads with diverse memory access patterns. For example, two memory regions with distinct memory access patterns might exist in a graph processing application: graph edges are traversed sequentially, while vertices are accessed randomly. In such scenarios, applying separate prefetching (and reclamation) policies to the two regions can significantly enhance performance and memory savings.

Specialization in PageFlex is guided by application-provided hints. PageFlex offers an inter-process communication (IPC) interface, enabling authorized agents to provide these hints. Each hint message associates an existing PageFlex policy, along with specific parameters, to a designated memory region represented by memory addresses. With PageFlex's IPC client library, agents can be developed to monitor and analyze applications' memory access patterns (e.g., detecting sequentiality) and translate these observations into corresponding PageFlex hints (e.g., applying lookahead prefetching with a step size of *N*). Agents can be implemented within the application process or as independent, authorized processes running alongside the application. By providing hints

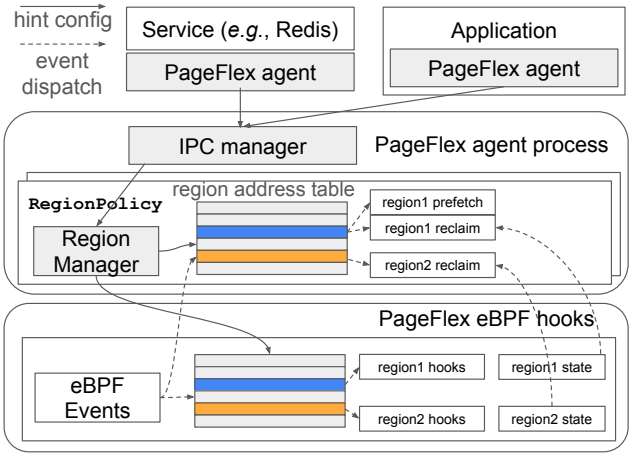


Figure 6: Control (solid line) and event dispatch (dotted line) paths for application specialization support.

to PageFlex for region creation/deletion or adjustments to prefetching/reclamation policies, applications can create specialized spatial and temporal and paging policies.

As illustrated by Figure 6, PageFlex enforces specializations through a new policy, *RegionPolicy*, which composes existing PageFlex policies and dynamically manages isolated sub-policies for specific memory address regions. Upon receiving a hint from an agent, PageFlex dynamically creates, updates, or removes sub-policies. Internally, PageFlex maintains a mapping table that associates memory regions with their corresponding sub-policies. To ensure correctness, the regions in this table are non-overlapping, and each region can be linked to at most one prefetching policy and one reclamation policy. PageFlex dispatches events to the appropriate sub-policies based on the virtual address of the accessed page, operating seamlessly across both user and kernel space.

4.3 Limits of policy expressibility

To avoid the CPU and memory overheads of exporting page access data to user space, PageFlex makes the design choice to limit the policy to eBPF programs which need to work with very limited state in the kernel. This may limit policies in some ways. First, writing complex policies could be challenging with eBPF due to verifier restrictions on program complexity (§ 2.6), limiting some policies such as LRB [42] which require ML model inference. Second, limited state such as the 32-bit per-page weights available to summarize a page's access history that are only accessible to eBPF code (as opposed to complex user-level data structures) could also limit the policies which work with more "information context". For example, a hypothetical policy that looks for spatial access patterns across the address space over time may be challenging to implement.

For such policies, PageFlex may still be used by shifting

the policy complexity to user space while the eBPF handlers merely export the necessary state instead of running the policy. However, in such cases, policies are left to account for the CPU/memory costs of exporting the state when weighing against the potential memory savings. It is worth noting that policy complexity generally translates to increased cost in making policy decisions to which paging policies are highly sensitive; as a result the increased cost may discourage such complex policies regardless of where they are implemented.

5 Implementation

PageFlex is implemented with a combination of Linux kernel changes, a user-space agent written in C++, and a collection of eBPF programs. We made 608 lines of changes to kernel code with most of it defining and introducing tracepoints at various paging events, and supporting the reserved field in the page `struct`. Excluding the policy-specific component, the agent constitutes 2,900 lines of C++ code and the eBPF programs are 700 lines of code to support the key components and the simplified policy interfaces.

Tracepoints. We add tracepoints in the kernel code to support PageFlex's events (sometimes at multiple places for each event) and attach eBPF programs to run the event handlers where needed. Page allocations, frees, swap-ins and swap-outs are traced from cgroup page charge and uncharge functions that already track all possible ways pages enter and leave memory. Page-fault events are traced from page-fault handling code. For page-access scans, we depend on a kernel thread similar to `g-swap`'s *kstaled* whose period can be externally controlled. Currently, the thread scans all physical pages in the system in each scan; in future, this can be made cgroup-specific by moving to the (already-upstreamed) MGLRU's page-table-based scans [8]. All tracepoints export the `memcg` id of the page so PageFlex can target required cgroups and a writable pointer to the reserved field in the page `struct` (described below).

Per-page persistable state. To provide low-overhead per-page state for eBPF code, we reserve a 4-byte field in the kernel page `struct` (associated with for each page in memory) exclusively for eBPF use. By default, eBPF disallows writing to kernel data structures exported through the tracepoints. To allow write access, we use writeable tracepoints [5] that allow explicitly whitelisting specific (section of) kernel data structures (in our case, the reserved 4-byte field in the page `struct`) to be writable by eBPF programs. This is safe because we make sure this field is never read by the kernel so any data written by the eBPF programs has no effect on the kernel's operation. To persist the state across swap-outs (the page `structs` are freed for swapped-out pages), we re-use swap cgroup maps that already maintain cgroup ids for swap entries for cgroup-level swap accounting. When a page is swapped in, we retrieve the old state and provide it to the policy through the `OnPageAlloc` event.

eBPF maps. PageFlex extensively uses eBPF maps for eBPF/user space communication. For each policy, we use a `BPF_MAP_TYPE_HASH` map for policy configuration, `BPF_MAP_TYPE_ARRAY` map for statistics, and `BPF_MAP_TYPE_RINGBUF` for event notifications to trigger user-level handlers if needed. Policies may employ maps where needed, too. For policies in § 4.1, we use histograms modeled on hash maps to communicate page weight distributions. Region-aware policy in § 4.2 maintains a hash map to store the VMA segment to sub-policy association, which is traversed to find the right policy handler at each event. (Current implementation reserves 256 slots per enclave.)

6 Evaluation

In this section, we evaluate PageFlex for its flexibility and performance overheads. Specifically, we focus on the following high-level questions:

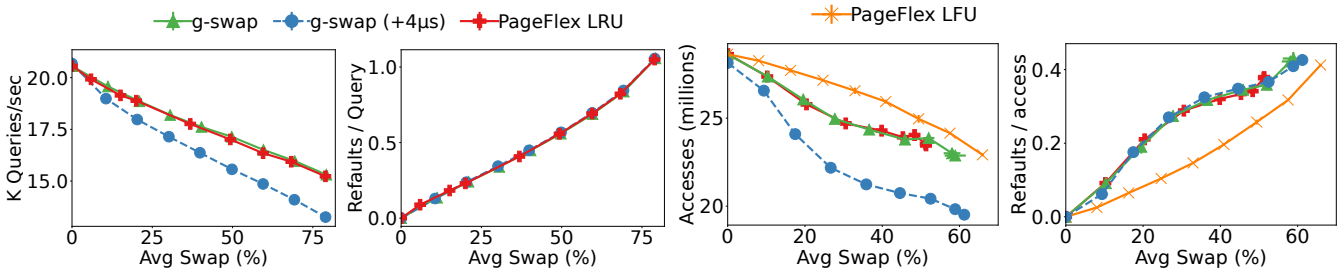
- Can PageFlex externalize current (kernel-based) policy implementations without negating their benefits?
- Is PageFlex flexible enough to support new policies? Specifically, can PageFlex enable specializing the policies, from general policy improvements that go beyond the current defaults like LRU to more tailored application-specific ones, and show improved memory savings?
- What overheads does PageFlex introduce, if any?

6.1 Testbed

We run experiments on a server fitted with two 24-core Intel Xeon E5-2696 CPUs each with 128 GB of memory. The server runs Linux 5.10 with PageFlex kernel changes discussed in §5. We use `g-swap` [32] as our kernel-based baseline for comparison. To ensure a fair comparison, we use the same kernel rebased with `g-swap`. Unless otherwise mentioned, we use zswap [7] as the swap backend. For a fast SSD-based swap emulation (with a 50-μs mean access latency), we use a block device supported by the Linux ublk driver using an SPDK Ublk target [11]. We use a combination of real-world benchmarks like Redis [4], RocksDB [22] and the GAP benchmark suite [14], and various synthetic workloads with a memory-access generator when evaluating application-level performance impact. For some long-running workloads, we first collect page-level access traces and replay them faster with the access generator (along with commensurate speedup in policy execution) where we need to collect a large number of data points; in such cases, we take care to only compare policy effectiveness (e.g., cache miss rate) and not raw performance (§ 6.3). Table 2 shows the list of workloads for which we collected page-level access traces by periodically scanning the page tables and logging the accessed

Trace group	#Traces	#Pages (M)	Max RSS (GB)	Span (hrs)	Scan interval (s)	Description
Redis	1	10	40	1	10	10-M keys, zipf skew = 0.5 workload
Memcached	4	1.3	5.2	1	60	10M 1024-B and 8-KB objs, sliding hotset
Database A	2	1	4	1	30	Proprietary SQL DB (like [44]) benchmark
Twitter KV	3	2.9–11.2	11.6–44.8	24-100	10	Twitter KV traces [49] played on Redis
Meta KV	4	2.8	11.2	24	10	MetaKV traces [9] played on Redis

Table 2: Traces used for evaluation of PageFlex policies. We collect the page-level access traces by periodically scanning the page tables, at a fine granularity (10-60 seconds) compared to their total running time (hours to days).



(a) Redis (b) Synthetic

Figure 7: Performance comparison of PageFlex's LRU with `g-swap`'s kernel-based implementations for (a) Redis with a zipf(0.5) client and (b) a memory access generator with an LFU-friendly workload. In each case, the left plot shows the raw application performance at various memory offloading targets and the right plot shows the refault rates for the same.

pages in each scan. We get these traces at a fine granularity (10–60 seconds) compared to their total running time which varies from hours to days. Where possible, we always batch the `madvise` calls to issue page-in or page-out hints for up to 64 pages. (We quantify the benefit in § 6.5).

6.2 Existing policies on PageFlex

We start with implementing existing policies on PageFlex to show that PageFlex can support these policies without causing any application slowdown compared to their functionally-equivalent kernel-based implementations.

PageFlex LRU vs `g-swap`. We use `g-swap`'s age-based reclamation as our kernel baseline and compare with PageFlex's LRU, which fully implements `g-swap` outside the kernel (§ 4.1). We run Redis with the memtier client, loaded with 10M keys with 1024-byte values (12 GB total memory), and measure its performance for requests drawn from a zipf(0.5) distribution. To compare policies at different degrees of memory offloading, we configure both systems to tune the age threshold based on a static memory reclamation target (i.e., a fraction of total process memory that should be swapped out).

Figure 7a (left) shows the Redis performance (queries per second) at various memory reclamation targets for both systems. The overall performance degrades with the increasing memory offloading as expected, as the refault rate per query (shown in Figure 7a right) increases. The performance of PageFlex's LRU is similar to `g-swap`'s LRU (within 0.98% of baseline) for all memory reclamation targets. The refault rates are also similar, showing their functional equivalence as well. To give a sense of the potential impact of `userfaultfd`-like overhead in the page swap-in path, we show the performance of `g-swap` modified to inject an additional 4-μs delay for each page fault, the overhead that `userfaultfd` introduces (§2.4). (Note that drawing an apples-to-apples comparison with real `userfaultfd` was infeasible, as it cannot use a kernel-based swap backend as PageFlexdoes; additionally, page access information is not yet available in `userfaultfd`). This result is significantly worse, slowing down the application by up to an additional 13.3% (for a similar refault rate), showing the importance of avoiding overheads in the page fault path. For reference, even a policy that randomly reclaims pages, which increases the refaults per query by 40% (to 1.4) at the rightmost point (80% swap), only incurs a 8.1% additional slowdown. Note that we only look at the direct impact to application performance here; we discuss the overheads added to the asynchronous policy work, which is similar across benchmarks, in §6.5.

To show another example where PageFlex can support a policy better suited for the workload, we use a synthetic memory access generator that generates page accesses from a skewed distribution interleaved with large sequential scans—a workload where pure recency-based eviction algorithms like LRU do poorly and LFU does better as it can filter out the noisy scans more effectively. We take measurements similar to Figure 7a, with the number of pages accessed in a fixed time (5-minute runs) as the performance metric. Figure 7b shows the results for the same configurations as above, as

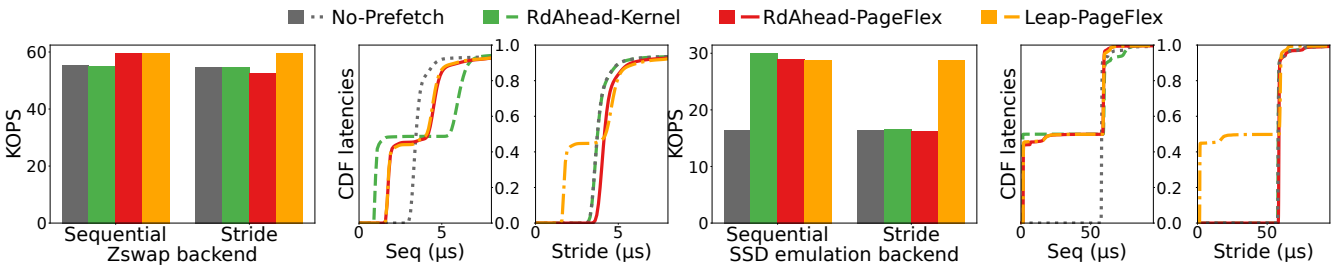


Figure 8: Page refault rate (pages/sec) and page fault latency distributions of a sequential and strided memory access patterns when working with various prefetching policies on two very different swap backends. Both kernel and PageFlex-based read-ahead improve the sequential performance but only Leap helps the strided one. The improvements are more significant with the emulated SSD backend.

well as LFU implemented with PageFlex. Both LRUs perform similarly worse compared to LFU, which degrades more gracefully with increasing memory offloading, due to the better miss rate shown in the right graph. For example, for a 10% slowdown (25.8 million accesses), LFU achieves 41% swap usage compared to LRU's 20% swap usage, providing 2× the memory savings. Emulated `userfaultfd` performance is again significantly worse than the baseline, potentially negating LFU's benefit (if it were to be implemented as a `userfaultfd`-based policy).

Linux default read-ahead with PageFlex. We externalize Linux's default read-ahead policy with PageFlex to quantify the overheads of running a prefetching policy on PageFlex. We implemented Linux's simple sequential trend detection and window size calculation in the `PredictTrend` function provided by our prefetching interface, which runs in the user-level code. The implementation consists of 61 lines, 41 of which were borrowed from the kernel code to perform window-size estimation.

To show policy equivalence and overheads, we use a synthetic benchmark with sequential and strided access patterns. The benchmark starts with a large dataset in the swap backend and promotes the data into memory with various fault patterns. Figure 8 (left) shows the rate at which the benchmark can promote the pages (with a 10-μs no-op work between the accesses) from a zswap backend. With sequential accesses, both kernel and PageFlex read-ahead prefetch half the pages as expected—we limit the prefetch window size to 1 in all cases. Figure 8 (second) shows the swap-in latency distribution. The no prefetching case (`No-Prefetch`) sees the standard zswap swap-in latency (3.5 μs at median) for most pages. For both read-aheads (`RdAhead-Kernel` and `RdAhead-PageFlex`), half of the pages see lower (minor-fault) latency due to a prefetching hit (1.08 μs for kernel and 1.85 for PageFlex at p25), whereas the other half see higher latency compared to `No-Prefetch` due to a miss. The kernel's miss latency (6.2 μs at p75) is twice as high as usual major fault, as the prefetches are synchronously issued with the original page fault and (unlike a block device-based swap backend) zswap's swap handling is also synchronous. PageFlex, however, prefetches asynchronously, incurring a miss latency closer to that of a major fault (`No-Prefetch`). For both hits and misses, PageFlex's latency is still slightly (0.5–1 μs) higher than the usual minor and major faults respectively—while a small fraction of it is PageFlex's eBPF overhead, we noticed that the interleaved `madvise` calls that do the prefetching from PageFlex's agent thread contribute to this overhead. Still, PageFlex's read-ahead is (inadvertently) faster than the kernel because of its asynchronous prefetching in the `sequential` case, but is slightly (3%) slower than the kernel, as expected, in the `strided` case with no prefetching.

The benefits of prefetching are more apparent with an SSD-like swap backend, where the cost of a major fault is much higher and prefetching makes a huge difference. Figure 8 (right half) shows the same results with an SSD emulation backend, where the swap-in latency is 58 μs at the median (for `No-Prefetch`) and both the read-aheads avoid this cost for half the prefetched pages. The kernel read-ahead promotes pages 82% faster than `No-Prefetch` for the `sequential` benchmark, and PageFlex's read-ahead is similarly (76%) faster. When not prefetching (in the `stride` case), PageFlex (at 16228 ops per sec) is only 0.8% slower than `No-Prefetch` (16364), adding very little overhead. In the figure, we also show the performance of the Leap [12] prefetching implemented with PageFlex (details in §6.3), which can capture strided patterns that the default read-ahead implementations cannot. In the `stride` case, only Leap prefetches (half) the pages as expected, and delivers higher performance over the both read-aheads on both the backends; the difference is starker in SSD emulation case with a 75.4% improvement (at 28,710 operations per second).

6.3 Beyond default policies

To demonstrate PageFlex's flexibility, we implemented two additional paging policies, Hyperbolic caching [18] and a version of Belady's MIN [16], as well as the Leap prefetching [12]. Table 3 shows the implementation complexity of

Kind	Policy	eBPF (LoC w/ simple interface)	User-level (LoC w/ simple interface)
Reclaim	LRU (`g-swap`)	6	0
	LFU	8	0
	Hyperbolic [18]	17	0
	Belady's MIN [16]	33	100
	Region-aware (§6.4)	95	204
Prefetch	Linux default	0	61
	Leap [12]	0	187

Table 3: Policies we implemented with PageFlex.

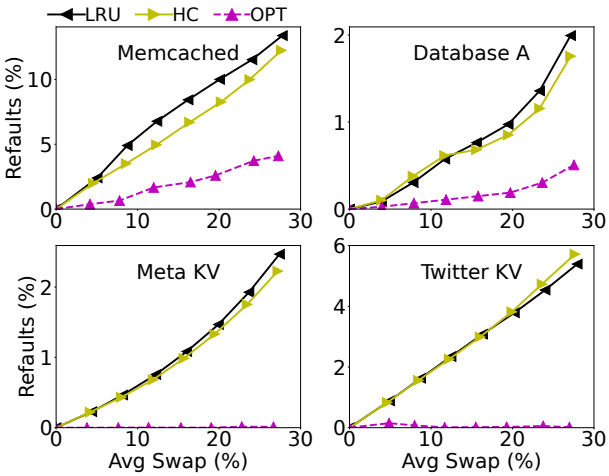


Figure 9: Refault rates for LRU, Hyperbolic (HC) and Belady's MIN policies with PageFlex for various page access traces, for memory offload targets up to 30%.

each.

Hyperbolic caching [18] is a recent cache eviction policy that introduces *hit density*, a new metric defined as the ratio of the number of accesses seen to the age of the page in the cache, to order the pages for eviction (lower hit density gets evicted first). Hyperbolic requires 17 LoC as shown in Figure 5 (center).

Belady's MIN [16]. We also implemented Belady's MIN [16] with a page access oracle based on the traces; this policy is not practical for real applications, but serves as a good representative for complex learning-based policies like LRB [42] from an implementation flexibility standpoint. Our MIN implementation re-uses our generic model with one difference: the `UpdateWeight` function performs a lookup to a table with distance to the next access for all pages (installed before each scan by the user-level code with access to the trace); one can imagine a policy that uses a learned ML model like LRB [42] to predict the same information.

We evaluate the performance of the above policies (i.e., refault or miss ratio at a given amount of offloaded memory) on the traces shown in Table 2. Figure 9 shows the observed refault rate (normalized to trace length) against average swap usage for one representative trace from each group (the results for other traces in the group are similar). Hyperbolic caching (HC) does slightly better than LRU for Memcached, Meta KV and in some regions of Database A. With Memcached, for example, Hyperbolic allows up to 5% more memory savings at a certain refault rates (e.g., 7%). For all the traces, there is a big gap between either of these policies and MIN, suggesting opportunity for better policies, as we also noted in §2.3.

Leap prefetching [12]. To show prefetching flexibility, we re-implemented the Leap prefetching with PageFlex. We implemented Leap's complex trend detection (§2.3) entirely in the `PrefetchTrend` function provided by our prefetching interface, where we re-used most of the trend detection logic from the original Leap implementation [1]. This required 187 lines of policy-specific code, out of which 160 were copied from the original Leap code. We discussed the resulting prefetching improvements at the end of §6.2.

6.4 Application-specific policies

We demonstrate PageFlex's flexibility in supporting specialized policies with spatial/temporal workload awareness with the help of two case studies.

Policy specialization in a key/value store. Meta's study [22] demonstrates that key/value store access patterns exhibit significant non-uniformity, with strong key-space and temporal locality. We reproduce the access patterns from the original study [22] using the db_bench tool, focusing on two scenarios: 1) *Region (key-space) hotness pattern*, where different memory regions are assigned varying eviction thresholds. We implement a PageFlex agent that monitors the hotness in different application-defined regions. This agent uses an LRU policy for all regions but sets distinct reclamation thresholds based on observed region hotness. We assume the KV store logically maps its key space onto the address regions (hence patterns), for example, by using key-range-based heap segregation or by re-organizing/compacting key/value pairs. 2) *Phase change pattern*, where the access pattern changes from random access to sequential scan during the runtime. We implement an agent that dynamically switches the reclamation policy from the default LRU to MRU accordingly. We then compare the average memory savings for these specialized policies with the default LRU policy, while keeping the KV performance target the same for all policies. As shown in Figure 10-left, the specialized policies achieve up to 36% more memory savings than the default LRU policy.

ExtMem [29] policies for PageRank. ExtMem introduced a specialized compound reclaim and prefetch policy designed for the PageRank workload within the GAP benchmark suite [14]. This policy targets a specific memory region within the application, the "edge array", applying sequential eviction and prefetching to optimize performance. To leverage this policy with PageFlex, we implemented an agent within the application that sends hints to PageFlex upon creation of

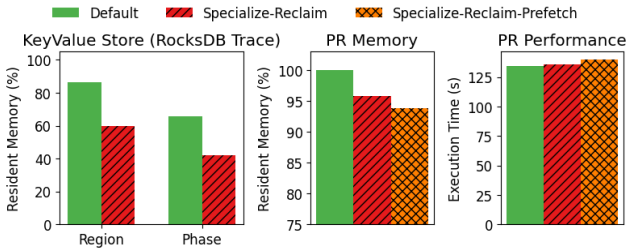


Figure 10: Specialized policies in KV store and GAPBS PageRank.

the edge array. This agent assigns a lookahead prefetching policy and an MRU reclamation policy to the edge array region. All other regions use the default LRU reclamation policy. This agent requires adding approximately 10 lines of code to the application, with no modifications required within PageFlex. This specialized policy reduces memory usage by 6.4% (Figure 10-middle) compared to a uniform LRU policy, with a performance overhead of less than 2% (Figure 10-right).

6.5 PageFlex Overheads

PageFlex runs most policy infrastructure asynchronously and overheads introduced in these paths do not directly cause application slowdown (as we have seen so far), but they still increase the resource usage, which could be a concern. We discuss these overheads below.

Page table scanning overhead. Recall that PageFlex employs a kernel thread to periodically scan page tables to inform page access information with `OnpageScanned` events. These scans are CPU-intensive and any increased overhead could limit how often a policy can choose to run these scans, affecting its efficacy. To isolate the overhead added by PageFlex's eBPF hooks, we compare the CPU usage for each scan in the Redis experiments from §6.2, where we compared PageFlex LRU and `g-swap`. While `g-swap`'s scans take 0.72 seconds (on 1 CPU core) on average to scan 12 GB of memory, PageFlex scans average 0.87 secs, a 17% increase due to the eBPF invocation for each page scanned (≈50 ns per invocation). Note that the relative overhead is constant regardless of the application's memory footprint. Unlike PageFlex, `userfaultfd` does not provide page access information.

Paging from user space. Unlike kernel-based policies, policy outcomes in PageFlex are enforced from the user-level agent, incurring extra syscall overhead for calls to `madvise`, which we amortize with batching. To quantify the overhead, we compare the work done by PageFlex's agent for reclaiming pages to that of `g-swap`'s *kreclaimd* thread which reclaims pages directly in the kernel. While *kreclaimd* spends an average of 4.3 μs to offload a page to zswap, PageFlex takes 6.9 μs to do the same from the agent, a 57% increase. However, after batching the offloading with one `process_madvise` call per 64 pages, the cost comes down to an amortized 4.9 μs per page, a 14% overhead. Note that, assuming similar paging costs, `userfaultfd` would incur a similar user-kernel crossing overhead for its syscalls, in addition to the cost of extra page copying per syscall.

7 Conclusion

Paging policies determine how much memory can be offloaded within a performance target. Existing deployments employ OS-based policies, making it slow and risky to evolve them in hyperscaler data centers. User-level designs such as `userfaultfd` and library-based approaches can provide better policy flexibility by completely externalizing paging, but they introduce practicality concerns such as significant performance overheads or incompatibility with existing deployments. To address these issues, PageFlex carefully separates the paging policy from the underlying mechanisms, and delegates the policy to eBPF programs, thereby providing policy flexibility while keeping the overheads low. We show that PageFlex enables writing prefetching and reclamation policies with varying degrees of specialization, and with virtually no impact on application performance.

Acknowledgments

We thank David Culler, Andrew Baumann, the anonymous reviewers, and our shepherd, Ankit Bhardwaj, for valuable feedback on earlier drafts of this manuscript.

References

[1] Leap prefetching changes to the Linux kernel. https://github.com/SymbioticLab/Leap.

[2] Querying data in Amazon S3 directly with user-space page fault handling. https://tech.nextroll.com/blog/data/2016/11/29/traildb-mmap-s3.html, 2016.

[3] Linux Userfaultfd. https://www.kernel.org/doc/html/latest/admin-guide/mm/userfaultfd.html, 2022.

[4] Redis - an open source, in-memory data store. https://redis.io/, 2023.

[5] Linux ebpf patch: writable contexts for bpf raw tracepoints. https://lwn.net/ml/bpf/20190419210409.5021-1-mmullins@fb.com/, 2024.

[6] Linux multi-gen LRU. https://www.kernel.org/doc/Documentation/mm/multigen_lru.rst, 2024.

[7] Linux zswap swap cache documentation. https://docs.kernel.org/admin-guide/mm/zswap.html, 2024.

[8] Merging the multi-generational LRU. https://lwn.net/Articles/894859/, 2024.

[9] Running cachebench with the trace workload. https://cachelib.org/docs/Cache_Library_User_Guides/Cachebench_FB_HW_eval, 2024.

[10] A thorough introduction to eBPF. https://lwn.net/Articles/740157/, 2024.

[11] ublk: the new and improved way to serve spdk storage locally! https://spdk.io/news/2023/03/28/ublk/, 2024.

[12] Hassan Al Maruf and Mosharaf Chowdhury. Effectively prefetching remote memory with Leap. In *Proceedings of the USENIX Annual Technical Conference*, July 2020.

[13] Emmanuel Amaro, Christopher Branner-Augmon, Zhihong Luo, Amy Ousterhout, Marcos K. Aguilera, Aurojit Panda, Sylvia Ratnasamy, and Scott Shenker. Can far memory improve job throughput? In *Proceedings of the 15th European Conference on Computer Systems, EuroSys 2020*, page 16. ACM, 2020.

[14] Scott Beamer, Krste Asanović, and David Patterson. The GAP benchmark suite. *arXiv preprint arXiv:1508.03619*, 2015.

[15] Nathan Beckmann, Haoxian Chen, and Asaf Cidon. LHD: Improving cache hit rate by maximizing hit density. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 389–403, Renton, WA, April 2018. USENIX Association.

[16] L. A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal*, 5(2):78–101, 1966.

[17] Ashish Bijlani and Umakishore Ramachandran. Extension framework for file systems in user space. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 121–134, Renton, WA, July 2019. USENIX Association.

[18] Aaron Blankstein, Siddhartha Sen, and Michael J. Freedman. Hyperbolic caching: Flexible caching for web applications. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 499–511, Santa Clara, CA, July 2017. USENIX Association.

[19] Irina Calciu, M. Talha Imran, Ivan Puddu, Sanidhya Kashyap, Hasan Al Maruf, Onur Mutlu, and Aasheesh Kolli. Rethinking software runtimes for disaggregated memory. In *ASPLOS*, pages 79–92, 2021.

[20] Blake Caldwell, Sepideh Goodarzy, Sangtae Ha, Richard Han, Eric Keller, Eric Rozner, and Youngbin Im. FluidMem: Full, flexible, and fast memory disaggregation for the cloud. In *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*, pages 665–677, 2020.

[21] Xuechun Cao, Shaurya Patel, Soo Yee Lim, Xueyuan Han, and Thomas Pasquier. FetchBPF: Customizable prefetching policies in linux with eBPF. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*, pages 369–378, Santa Clara, CA, July 2024. USENIX Association.

[22] Zhichao Cao, Siying Dong, Sagar Vemuri, and David HC Du. Characterizing, modeling, and benchmarking RocksDB key-value workloads at Facebook. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 209–223, 2020.

[23] Lei Chen, Shi Liu, Chenxi Wang, Haoran Ma, Yifan Qiao, Zhe Wang, Chenggang Wu, Youyou Lu, Xiaobing Feng, Huimin Cui, Shan Lu, and Harry Xu. A tale of two paths: Toward a hybrid data plane for efficient Far-Memory applications. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 77–95, Santa Clara, CA, July 2024. USENIX Association.

[24] Zhongjie Chen, Qingkai Meng, ChonLam Lao, Yifan Liu, Fengyuan Ren, Minlan Yu, and Yang Zhou. eTran: Extensible kernel transport with eBPF. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*, pages 407–425, Philadelphia, PA, April 2025. USENIX Association.

[25] Padmapriya Duraisamy, Wei Xu, Scott Hare, Ravi Rajwar, David Culler, Zhiyi Xu, Jianing Fan, Christopher Kennelly, Bill McCloskey, Danijela Mijailovic, Brian Morris, Chiranjit Mukherjee, Jingliang Ren, Greg Thelen, Paul Turner, Carlos Villavieja, Parthasarathy Ranganathan, and Amin Vahdat. Towards an adaptable systems architecture for memory tiering at warehouse-scale. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS 2023, page 727–741, New York, NY, USA, 2023. Association for Computing Machinery.

[26] Zhiyuan Guo, Zijian He, and Yiying Zhang. Mira: A program-behavior-guided far memory system. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 692–708, New York, NY, USA, 2023. Association for Computing Machinery.

[27] Steven M. Hand. Self-paging in the Nemesis operating system. In *USENIX Symposium on Operating Systems*

Design and Implementation (OSDI 99), pages 73–86. USENIX Association, 1999.

[28] Jack Tigar Humphries, Neel Natu, Ashwin Chaugule, Ofir Weisse, Barret Rhoden, Josh Don, Luigi Rizzo, Oleg Rombakh, Paul Turner, and Christos Kozyrakis. GhOSt: Fast & flexible user-space delegation of linux scheduling. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 588–604, New York, NY, USA, 2021. Association for Computing Machinery.

[29] Sepehr Jalalian, Shaurya Patel, Milad Rezaei Hajidehi, Margo Seltzer, and Alexandra Fedorova. ExtMem: Enabling application-aware virtual memory management for data-intensive applications. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*, pages 397–408, 2024.

[30] Song Jiang and Xiaodong Zhang. LIRS: an efficient low inter-reference recency set replacement policy to improve buffer cache performance. *SIGMETRICS Perform. Eval. Rev.*, 30(1):31–42, jun 2002.

[31] Kostis Kaffes, Jack Tigar Humphries, David Mazières, and Christos Kozyrakis. Syrup: User-defined scheduling across the stack. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 605–620, New York, NY, USA, 2021. Association for Computing Machinery.

[32] Andres Lagar-Cavilla, Junwhan Ahn, Suleiman Souhlal, Neha Agarwal, Radoslaw Burny, Shakeel Butt, Jichuan Chang, Ashwin Chaugule, Nan Deng, Junaid Shahid, Greg Thelen, Kamil Adam Yurtsever, Yu Zhao, and Parthasarathy Ranganathan. Software-Defined Far Memory in Warehouse-Scale Computers. *International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS*, pages 317–330, 2019.

[33] Donghee Lee, Jongmoo Choi, Jong-Hun Kim, S.H. Noh, Sang Lyul Min, Yookun Cho, and Chong Sang Kim. LRFU: a spectrum of policies that subsumes the least recently used and least frequently used policies. *IEEE Transactions on Computers*, 50(12):1352–1361, 2001.

[34] Michael Marty, Marc De Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C Evans, Steve Gribble, Nicholas Kidd, Gautam Kumar, Carl Mauer, Emily Musick, Lena Olson, Erik Rubow, Michael Ryan, Kevin Springborn, Paul Turner, Valas Valancius, Xi Wang, and Amin Vahdat. Snap: a Microkernel Approach to Host Networking. In *Proceedings of the ACM Symposium on Operating Systems Principles*, pages 399–413, 2019.

[35] Nimrod Megiddo and Dharmendra S. Modha. ARC: A Self-Tuning, low overhead replacement cache. In *2nd USENIX Conference on File and Storage Technologies (FAST 03)*, San Francisco, CA, March 2003. USENIX Association.

[36] Elizabeth J. O'Neil, Patrick E. O'Neil, and Gerhard Weikum. The LRU-K page replacement algorithm for database disk buffering. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, SIGMOD '93, page 297–306, New York, NY, USA, 1993. Association for Computing Machinery.

[37] Sujin Park, Diyu Zhou, Yuchen Qian, Irina Calciu, Taesoo Kim, and Sanidhya Kashyap. Application-Informed kernel synchronization primitives. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 667–682, Carlsbad, CA, July 2022. USENIX Association.

[38] Ivy Bo Peng, Marty McFadden, Eric W. Green, Keita Iwabuchi, Kai Wu, Dong Li, Roger Pearce, and Maya B. Gokhale. UMap: Enabling application-driven optimizations for page management. *CoRR*, abs/1910.07566, 2019.

[39] Yifan Qiao, Chenxi Wang, Zhenyuan Ruan, Adam Belay, Qingda Lu, Yiying Zhang, Miryung Kim, and Guoqing Harry Xu. Hermit: Low-Latency, High-Throughput, and transparent remote memory via Feedback-Directed asynchrony. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 181–198, Boston, MA, April 2023. USENIX Association.

[40] Amanda Raybuck, Tim Stamler, Wei Zhang, Mattan Erez, and Simon Peter. HeMem: Scalable tiered memory management for big data applications and real nvm. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 392–407, New York, NY, USA, 2021. Association for Computing Machinery.

[41] Zhenyuan Ruan, Malte Schwarzkopf, Marcos K. Aguilera, and Adam Belay. AIFM: High-Performance, application-integrated far memory. *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020*, pages 315–332, 2020.

[42] Zhenyu Song, Daniel S. Berger, Kai Li, and Wyatt Lloyd. Learning relaxed Belady for content distribution network caching. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 529–544, Santa Clara, CA, February 2020. USENIX Association.

[43] Indira Subramanian. Managing discardable pages with an external pager. In *USENIX Mach Symposium (USENIX Mach Symposium)*, Monterey, CA, November 1991. USENIX Association.

[44] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, Paul Bardea, Amruta Ranade, Ben Darnell, Bram Gruneir, Justin Jaffray, Lucy Zhang, and Peter Mattis. CockroachDB: The resilient geo-distributed SQL database. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 1493–1509, New York, NY, USA, 2020. Association for Computing Machinery.

[45] Brian R. Tauro, Brian Suchy, Simone Campanoni, Peter Dinda, and Kyle C. Hale. TrackFM: Far-out compiler support for a far memory world. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, ASPLOS '24, page 401–419, New York, NY, USA, 2024. Association for Computing Machinery.

[46] William Tu, Yi-Hung Wei, Gianni Antichi, and Ben Pfaff. Revisiting the open vSwitch dataplane ten years later. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM '21, page 245–257, New York, NY, USA, 2021. Association for Computing Machinery.

[47] Chenxi Wang, Yifan Qiao, Haoran Ma, Shi Liu, Yiying Zhang, Wenguang Chen, Ravi Netravali, Miryung Kim, and Guoqing Harry Xu. Canvas: Isolated and adaptive swapping for multi-applications on remote memory. In *Proceedings of the 19th USENIX Symposium on Networked Systems Design and Implementation*, April 2023.

[48] Johannes Weiner, Niket Agarwal, Dan Schatzberg, Leon Yang, Hao Wang, Blaise Sanouillet, Bikash Sharma, Tejun Heo, Mayank Jain, Chunqiang Tang, and Dimitrios Skarlatos. TMO: Transparent memory offloading in datacenters. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, page 609–621, New York, NY, USA, 2022. Association for Computing Machinery.

[49] Juncheng Yang, Yao Yue, and K. V. Rashmi. A large scale analysis of hundreds of in-memory cache clusters at Twitter. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 191–208. USENIX Association, November 2020.

[50] Juncheng Yang, Yazhuo Zhang, Ziyue Qiu, Yao Yue, and Rashmi Vinayak. FIFO queues are all you need for cache eviction. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 130–149, New York, NY, USA, 2023. Association for Computing Machinery.

[51] Zhe Yang, Youyou Lu, Xiaojian Liao, Youmin Chen, Junru Li, Siyu He, and Jiwu Shu. Lambda-IO: a unified IO stack for computational storage. In *Proceedings of the 21st USENIX Conference on File and Storage Technologies*, FAST'23, USA, 2023. USENIX Association.

[52] Anil Yelam, Stewart Grant, Saarth Deshpande, Nadav Amit, Radhika Niranjan Mysore, Amy Ousterhout, Marcos K. Aguilera, and Alex C. Snoeren. Eden: developer-friendly application-integrated far memory. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, April 2025.

[53] Wonsup Yoon, Jisu Ok, Jinyoung Oh, Sue Moon, and Youngjin Kwon. DiLOS: Do not trade compatibility for performance in memory disaggregation. In *Proceedings of the Eighteenth European Conference on Computer Systems*, EuroSys '23, page 266–282, New York, NY, USA, 2023. Association for Computing Machinery.

[54] Kan Zhong, Wenlin Cui, Xin Chen, Qiao Li, Zhe Yang, Youyou Lu, Xiaodan Yan, Siwei Luo, Qizhao Yuan, and Keji Huang. Revisiting swapping in user-space with lightweight threading. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 42(11):4205–4218, 2023.

[55] Yuhong Zhong, Haoyu Li, Yu Jian Wu, Ioannis Zarkadas, Jeffrey Tao, Evan Mesterhazy, Michael Makris, Junfeng Yang, Amy Tai, Ryan Stutsman, and Asaf Cidon. XRP: In-kernel storage functions with eBPF. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 375–393, Carlsbad, CA, July 2022. USENIX Association.

[56] Yang Zhou, Hassan M. G. Wassel, Sihang Liu, Jiaqi Gao, James Mickens, Minlan Yu, Chris Kennelly, Paul Turner, David E. Culler, Henry M. Levy, and Amin Vahdat. Carbink: Fault-Tolerant far memory. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 55–71, Carlsbad, CA, July 2022. USENIX Association.

ASTERINAS: A Linux ABI-Compatible, Rust-Based Framekernel OS with a Small and Sound TCB

Yuke Peng[1,*], Hongliang Tian[2,*], Junyang Zhang[3,4], Ruihan Li[3,4], Chengjun Chen[2], Jianfeng Jiang[2], Jinyi Xian[1], Xiaolin Wang[3,4], Chenren Xu[3,4], Diyu Zhou[3,4], Yingwei Luo[3,4,†], Shoumeng Yan[2,†], Yinqian Zhang[1,†]

[1]*SUSTech* [2]*Ant Group* [3]*Peking University* [4]*Zhongguancun Laboratory*

Abstract

How can one build a *feature-rich*, *general-purpose*, Rust-based operating system (OS) with a *minimal* and *sound* Trusted Computing Base (TCB) for *memory safety*? Existing Rust-based OSes fall short due to their improper use of `unsafe` Rust in kernel development. To address this challenge, we propose a novel OS architecture called *framekernel* that realizes Rust's full potential to achieve *intra-kernel privilege separation*, ensuring TCB minimality and soundness. We present OSTD, a streamlined framework for safe Rust OS development, and ASTERINAS, a Linux ABI-compatible framekernel OS implemented entirely in safe Rust using OSTD. Supporting over 210 Linux system calls, ASTERINAS delivers performance on par with Linux, while maintaining a minimized, memory-safety TCB of only about 14.0% of the codebase. These results underscore the practicality and benefits of the framekernel architecture in building safe and efficient OSes.

1 Introduction

Despite three decades of research into defending against memory safety bugs in operating systems (OSes) written in C, achieving true memory safety remains elusive. This was starkly demonstrated by the recent CrowdStrike outage [11], where millions of Windows PCs crashed due to an out-of-bounds memory access in a faulty driver. It is estimated that 60-70% of security vulnerabilities in system software written in C stem from memory safety issues [30].

In recent years, as the Rust programming language matures and becomes popular, the development of Rust-based, memory-safe OSes has gained momentum. Rust offers memory safety guarantees through innovative language features such as ownership, borrowing, and lifetimes, enabling safe memory management without relying on garbage collection. Many now see Rust as a potential successor to C and C++ as the dominant systems programming language. With the endorsement of Linus Torvalds, the Linux kernel has officially adopted Rust as its second programming language [18] and integrated the Rust for Linux [41] (RFL) project to facilitate writing "leaf" kernel modules in Rust. Additionally, new OS kernels like Tock [38], RedLeaf [49], and Theseus [16] are built from the ground up using Rust, further demonstrating Rust's potential in this domain.

Table 1: The `unsafe` keyword is widely utilized in the crates (or kernel modules) of existing Rust-based OSes. The statistics were derived from an analysis of the latest source code of these OSes at the time of writing.

Rust-based OSes	Linux	Tock	RedLeaf	Theseus
`Unsafe`-utilizing crates	6 / 11[1] (55%)	91 / 98 (93%)	36 / 58 (62%)	54 / 171 (32%)

[1] This includes the RFL crate and 10 notable Rust-written kernel modules [3].

While adopting Rust is a significant step toward achieving kernel memory safety, this is insufficient on its own since Rust-based OSes must include unsafe Rust code. The safety of the kinds of low-level controls required by kernel programming cannot be statically verified by the Rust compiler and thus is only allowed by Rust within special code blocks marked by the `unsafe` keyword. Despite the Rust language team dedicating an entire book [25] to the "dark arts of unsafe Rust", developers are still prone to misusing it, and the RustSec Advisory Database has recorded hundreds of bugs stemming from `unsafe` misuse [60].

To mitigate the risks associated with `unsafe` in Rust, two widely accepted best practices have emerged [12]: (1) use `unsafe` sparingly and (2) encapsulate `unsafe` code within safe abstractions. However, we have found that existing Rust-based OS kernels often fall short of these standards. Unsafe Rust code permeates a significant portion of a Rust-based OS. We observe that `unsafe`-utilizing crates make up 55%, 93%, 62%, and 32% of all crates in Linux, Tock, RedLeaf, and Theseus, respectively (as shown in Table 1). Although

*Co-first authors.

†Co-corresponding authors: Yinqian Zhang (yinqianz@acm.org); Shoumeng Yan (shoumeng.ysm@antgroup.com); Yingwei Luo (lyw@pku.edu.cn).

kernel developers generally view `unsafe` as a "necessary evil", we question whether such widespread use of `unsafe` is truly necessary. In particular, we challenge the necessity of `unsafe` in device drivers (as seen in all existing Rust-based OSes), which account for the majority of the codebase of a mature OS (70% in Linux [17]). Details about the pitfalls of `unsafe` handling in existing Rust OSes will be present in §2.

Given the limitations of prior work, we pose the question: is it possible to build a feature-rich, general-purpose, Rust-based OS kernel almost entirely in safe Rust? We introduce *framekernel* (§3), a novel OS architecture designed to achieve a minimal and sound TCB for a Rust-based OS. In the framekernel architecture, the entire OS resides in a single address space (as in a monolithic kernel) and is implemented in Rust. The kernel is logically divided into two parts: the *privileged* OS framework (akin to a microkernel) and the *de-privileged* OS services. Only the privileged framework is allowed to use `unsafe`, while the de-privileged services must be written in safe Rust completely. As the TCB, the privileged framework encapsulates all low-level, hardware-oriented `unsafe` operations behind safe APIs. Using these safe APIs, the de-privileged OS services can implement all kinds of OS functionalities, including device drivers. A framekernel minimizes the TCB size without incurring extra overheads due to hardware isolation. Thus, we claim that a framekernel combines the benefits of both a monolithic kernel and a microkernel (see Figure 1).

We enforce this *language-based, intra-kernel privilege separation* by systematically identifying *sensitive* OS resources, namely those that can be mis-programmed or misused to compromise memory safety – even with safe Rust. Thus, the design principle of a framekernel is to *keep sensitive OS resources within the privileged framework for soundness, while delegating insensitive OS resources to the de-privileged OS services for minimality*.

To realize the vision of framekernels, we develop OSTD (§4), the privileged OS framework required by a framekernel. OSTD provides a *small yet expressive* set of safe OS development abstractions, covering safe user-kernel interactions, safe kernel logic, and safe kernel-peripheral interactions. Of particular note is the *untyped memory* abstraction, which addresses the challenge of safely handling *externally-modifiable memory* (e.g., MMIO or DMA-capable memory) – a longstanding obstacle in safe driver development. In addition, we introduce *safe policy injection*, a technique that separates potentially complex "policy" components (e.g., task schedulers, page allocators, and slab allocators) from the core "mechanisms" of OSTD, thus containing the growth of OSTD's complexity over time. Furthermore, we define a set of key *safety invariants* that collectively ensure the soundness of OSTD's privilege separation.

To demonstrate the practicality and benefits of framekernels, we develop ASTERINAS (§5), a Linux ABI–compatible framekernel built on OSTD. ASTERINAS implements a rich

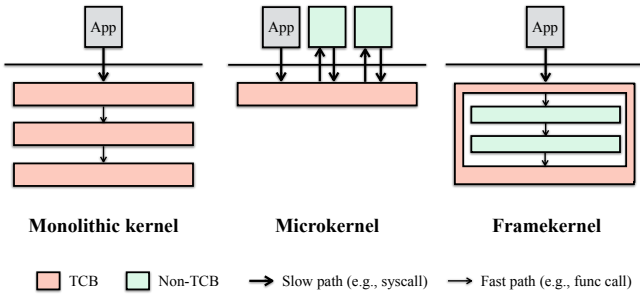


Figure 1: A framekernel combines the speed of a monolithic kernel and the security of a microkernel. The memory-safety TCB of a framekernel is reduced to the privileged OS framework (akin to a microkernel) without any communication overheads due to extra hardware-based isolation (similar to a monolithic kernel). We depict the TCB portion of a framekernel as a "frame" to highlight the fact all low-level interactions of non-TCB with the hardware (beneath the "frame") and the user space (above the "frame") is mediated by the TCB.

subset of Linux functionality: it supports over 210 Linux system calls, multiple file systems and socket types, and a variety of peripherals (e.g., disks and NICs). All these features are written in safe Rust, leveraging OSTD's APIs. ASTERINAS has been under development for three years. The repository of ASTERINAS and OSTD is open source [1, 2], containing over 100K lines of Rust code contributed by more than 50 individuals.

We conduct a thorough evaluation (§6) of ASTERINAS and OSTD across three dimensions: performance, TCB size, and soundness. ASTERINAS delivers performance on par with Linux: on LMbench, a syscall-intensive microbenchmark, ASTERINAS achieves a mean normalized performance score of 1.08 (relative to Linux, higher is better); for three I/O-intensive applications, Nginx, Redis, and SQLite, it delivers a normalized performance of 1.17, 1.31, and 0.85, respectively. The framekernel design also yields a lean TCB: ASTERINAS's TCB accounts for just 14.0% of its codebase, versus 43.8% in Tock, 62.4% in Theseus, and 66.1% in RedLeaf. Finally, to strengthen our confidence in OSTD's soundness, we develop KERNMIRI, a retrofitted version of Miri [48] for Rust OSes, which is used to detect potential safety issues in OSTD systematically.

Contributions. In summary, this paper makes the following contributions:

- We propose framekernel, a novel OS architecture that combines the benefits of both monolithic and microkernels by enforcing Rust-based intra-kernel privilege separation.
- We develop OSTD, a small and sound OS framework to facilitate OS development in safe Rust.
- We develop ASTERINAS, a highly-optimized, Linux ABI-compatible OS based on OSTD.

- We conduct extensive evaluations to show the performance, TCB size, and safety of both ASTERINAS and OSTD.

2 Background and Motivation

2.1 The Rusty Way to Safety

Rust is an efficient system programming language that offers strong guarantees for memory, type, and thread safety without compromising runtime performance, thanks to its strong type system and unique ownership model.

Ownership, borrowing, and lifetime. In Rust's ownership model, each value has a single owning variable, and the value's lifetime is tied to the owner's scope [10]. When the owner goes out of scope, the value is dropped. Ownership can be borrowed through references, subject to lifetime constraints enforced by the compile-time borrow checker.

Type system. The Rust compiler implements a tailored type system coupled with comprehensive compile-time checks [7, 9]. Utilizing type information, the compiler ensures that all accesses occur within valid lifetimes, generates correct memory offsets, and performs bounds checking, thereby guaranteeing both temporal and spatial memory safety.

Unsafe Rust. To offer additional expressive power, Rust provides the `unsafe` keyword, enabling programmers to bypass compile-time checks and thus shifting the responsibility for ensuring safety to those using `unsafe` code blocks [8].

Undefined behaviors. Undefined behaviors (UBs) in Rust refers to operations that compromise the language's correctness and safety guarantees, including memory safety, thread safety, and type safety. The Rust Reference book enumerates common UBs [6], e.g., data races, memory access based on dangling or misaligned pointers, out-of-bound memory access, breaking the pointer aliasing rules, and mutating immutable bytes. However, this list is neither comprehensive nor precise due to Rust's lack of a formalized specification. The Rust language team maintains an official UB detection tool called Miri [48], which, as part of the Rust toolchain, serves as a de-facto executable standard for UBs. Although Miri can effectively detect UBs in standard Rust applications, Miri is not applicable to Rust-based OSes as its design does not take into account the needs of low-level system programming, particularly in areas of memory management and hardware interaction. This work aims at systematic identification and prevention of UBs in the context of Rust OSes.

2.2 Rustification of Mainstream OSes

The efficiency and safety of Rust make it an attractive choice for developing OSes, leading to several mainstream OSes [29, 41, 54] integrating Rust into their codebases. The Rust for Linux (RFL) project [41] stands out as a prominent example.

RFL aims at establishing Rust as the second official language of the Linux kernel, allowing developers to write "leaf" kernel modules in safe Rust. Officially merged into Linux in 2022, RFL has laid the groundwork for Rust integration by 2024 [40]. Notably, three Rust-written device drivers, albeit basic, have already been added to the kernel tree [39].

Despite this progress, the unsafe nature of Linux remains. By design, RFL has to offer safe Rust abstractions over Linux's extensive legacy C APIs, resulting in substantial use of `unsafe` Rust. A recent study [39] notes that RFL already has 19K lines of Rust code upstream, with an additional 112K lines staged for upstream inclusion. A great portion of RFL is devoted to unsafe Rust code that interacts with the legacy C APIs. As RFL expands to cover more subsystems and their C APIs, we expect its codebase to grow to hundreds of thousands of lines. Therefore, the TCB size of RFL is substantial, even not considering Linux's huge C core and the countless C kernel modules around it.

> **Lesson Learned:** Constructing safe Rust abstractions on a legacy monolithic kernel inevitably requires a substantial use of `unsafe` Rust.

In addition to the inflated TCB size, the burden of a huge legacy codebase - its status quo and established philosophy - constrains the effectiveness of Rust and the soundness of RFL. This constraint leaves known soundness vulnerabilities unresolved in RFL. For example, Rust explicitly excludes memory leaks from its memory safety guarantees, permitting objects to be forgotten. However, it was discovered that forgetting RFL's mutex guards can trigger use-after-free vulnerabilities [55]. This vulnerability arises from conflicting API contracts between C and Rust regarding mutex unlocking obligations, so neither side is willing or capable to fix it. Another example of a soundness issue stems from Linux's (and by extension RFL's) tolerance for sleeping in atomic contexts such as spinlock or RCU-lock held regions. Sleep-in-atomic-context bugs may cause data races in RCU-protected memory accesses, undermining Rust API safety guarantees [31]. These unresolved soundness issues reflect the Linux community's traditional value of "pragmatism over safety" [31], creating fundamental tensions with Rust's security-first paradigm.

> **Lesson Learned:** Achieving sound memory safety requires a clean-slate OS that prioritizes safety above all else.

2.3 Clean-Slate Rust OSes

Clean-slate Rust OSes like Tock [38], RedLeaf [49], and Theseus [16] strive to fully leverage Rust's features to improve OS safety and reliability. However, they face a notable limitation: inadequate support for safe driver development. As shown in Table 2, device drivers in these systems frequently rely on `unsafe` code to manage low-level resources, such as raw data buffers, MMIO, I/O ports, and DMA regions. Given that drivers typically constitute the largest portion of

Table 2: Representative `unsafe` Rust code pattern in drivers

	Resource *access* within `unsafe` blocks	Resource *acquisition* within `unsafe` blocks
Tock	`// chips/nrf52:` `buf[idx] = *byte_ptr`	`// chips/nrf52:` `StaticRef::new(ptr)`
RedLeaf	`// lib/devices/ixgbe:` `ptr::read_volatile(addr)`	`// lib/devices/tpm:` `MmioAddr::new(base,len)`
Theseus	`// kernel/pci:` `pci_port.write(val)`	`// kernel/ixgbe:` `Box::from_raw(ptr)`

an OS codebase, extensive use of `unsafe` code significantly heightens the risk of memory safety vulnerabilities.

Lesson Learned: Safe driver development requires safe abstractions for acquiring and accessing low-level system resources.

Among the three Rust OSes, Theseus minimizes `unsafe` code through its `MappedPages` abstraction, which represents the *exclusive* ownership of a virtually-contiguous memory region backed by some *exclusively*-owned physical frames. This exclusiveness alone (seemingly) preserves the safety of Rust references (e.g., `&T`) borrowed from that range. However, this design overlooks *sensitive* or *externally-modifiable* memory. For example, using `MappedPages` to modify the sensitive memory of Local APIC may cause unpredictable CPU behavior (e.g., sending an IPI that resets a CPU). Similarly, Theseus drivers create Rust references (`&T`) to MMIO device memory, an unsound practice since such references assume no external modifications, which hardware may violate.

Lesson Learned: Rust OSes must develop safe abstractions for memory that is subject to external modifications, such as hardware, user programs, or DMA.

Finally, while language-level UBs are effectively addressed by Rust's safety guarantees, UBs stemming from execution environments or CPU architectures remain unresolved in existing Rust-based OSes. For example, a malicious device could corrupt kernel memory via DMA [44] or spoof interrupts to manipulate CPU trap handlers [63]. Similarly, a stack overflow could compromise the execution environment—an issue beyond the detection capabilities of safe Rust.

Lesson Learned: Rust OSes should safeguard against UBs not only at the language level but also at the architectural and environmental levels.

3 Framekernel Architecture

We introduce *framekernel*, a novel OS architecture that combines the benefits of both monolithic kernels and microkernels by fully leveraging the modern safe system language of Rust. In a framekernel OS, the entire OS kernel resides within a single address space, akin to a monolithic kernel, and is implemented in Rust. The kernel is logically divided into two parts: the *privileged OS framework* (similar to a microkernel) and the *de-privileged OS services*. Only the privileged framework may use Rust's `unsafe` features, while the de-privileged services are built entirely in safe Rust. The privileged framework encapsulates low-level, hardware-oriented unsafe operations into safe APIs, using which the de-privileged services implement most OS functionalities, including drivers, in safe Rust. Like a monolithic kernel, all components within a framekernel communicate efficiently (e.g., via function calls or shared memory). By restricting the privileged framework to a minimal set of functionalities, framekernels, like microkernels, reduce the size of the TCB, thereby enhancing safety, security, and reliability.

Design philosophy. At the heart of the framekernel architecture is *intra-kernel privilege separation*: although OS services operate in privileged CPU mode, their behaviors are constrained by safe Rust and the privileged framework, maintaining their de-privileged status. Only flaws within the privileged framework can jeopardize the kernel's memory safety. This intra-kernel privilege separation must uphold two key properties: soundness and minimality.

Soundness: The OS framework guarantees the absence of UBs under all circumstances, irrespective of interactions with OS services, user code, or peripheral devices.

This property aims to prevent all forms of UBs, which may originate at three levels (from high to low). First, **language-level UBs** in Rust are described in §2.1. Second, **environment-level UBs** arise when the code, stack, or heap is corrupted. Programming languages have implicit assumptions about their execution environments; UB occurs if those assumptions are compromised. Third, **architecture-level UBs** stem from incorrect use of CPU or hardware devices, such as improperly saving/restoring CPU registers, misconfiguring page tables, or allowing peripheral DMA to corrupt memory.

Minimality: A component is tolerated inside the OS framework only if moving it outside would prevent the implementation of OS services' required functionality or compromise soundness of the framework.

This property focuses on minimizing the principal component of the *runtime TCB* responsible for kernel memory safety. Naturally, the whole TCB extends beyond the OS framework itself, also including the Rust toolchain, the Rust core libraries, bootloader, and firmware involved in loading a framekernel, as well as the CPU plus some core devices (e.g., interrupt controller and IOMMU). The software and hardware outside the TCB, including the safe OS services, user-space programs, and peripheral devices (e.g., disks, NICs, and GPUs), are not trusted.

Towards effective privilege separation. Realizing intra-kernel privilege separation prompts a key question: *What must stay inside the framework (for soundness), and what can be moved outside (for minimality)?*

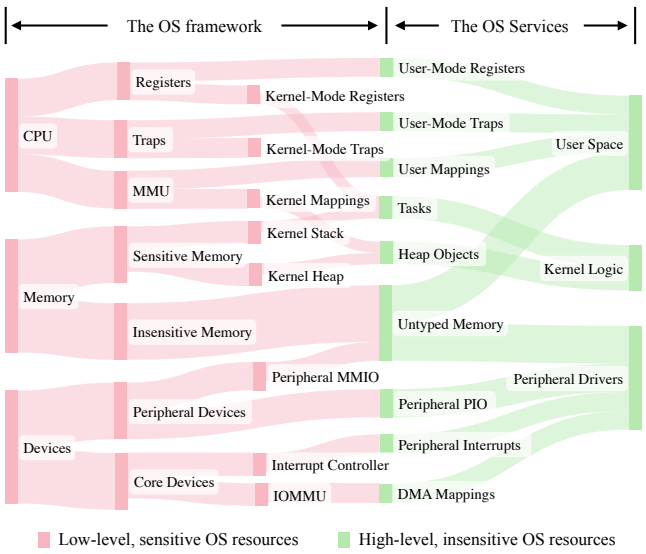


Figure 2: A blueprint for framekernel construction in a *resource*-centric view. The nodes represent OS resources, while the edges depict classification and encapsulation. The concept of untyped memory will be introduced in §4.2.

To answer this, we view framekernel construction from a *resource*-centric perspective, as shown in Figure 2. Fundamentally, an OS manages three classes of resources: CPU, memory, and devices. At first glance, these resources are all **sensitive** because the kernel code could misuse them (even in safe Rust) and break memory safety. However, deeper inspection shows that some subsets of these resources are actually **insensitive**. Thus, we *must keep sensitive resources inside the framework (for soundness) and should move insensitive ones outside (for minimality).*

All three classes of fundamental resources can be split into sensitive and insensitive subsets. **CPU** resources involve kernel-mode and user-mode control; the former is considered sensitive, while the latter (e.g., user-mode registers, user-mode traps) is insensitive because it cannot directly undermine kernel state. Furthermore, the user virtual address space is insensitive as manipulating the user virtual memory does not affect kernel memory safety. **Memory** resources include sensitive memory used for the kernel's code, stack, heap, and page tables, whereas memory dedicated to untrusted user processes or devices is insensitive because kernel safety does not depend on its integrity. **Devices** present a similar division, as core devices (e.g., APIC, IOMMU) are sensitive due to their machine-wide control, but peripheral devices (e.g., NICs, GPUs) are generally insensitive; a misconfigured core device can compromise the entire kernel, but failures in a peripheral device are typically confined to that device itself.

These observations inform a blueprint for framekernel construction (Figure 2). The blueprint *classifies* OS resources into sensitive and insensitive ones at a fine granularity, with low-level, sensitive ones *hidden* inside the privileged OS framework or *encapsulated* into high-level, insensitive ones. The net result is that the privileged OS framework only exposes to the de-privileged OS services a small set of insensitive resources that is sufficient to support the three primary needs of safe OS development: (1) safe user-kernel interactions, (2) safe kernel logic, and (3) safe kernel-peripheral interactions. This blueprint guides the design of OSTD.

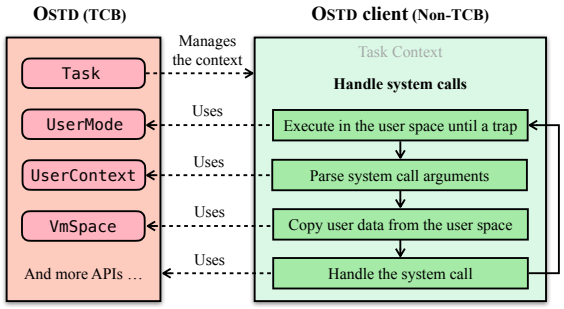


Figure 3: An example of OSTD APIs: system call handling

4 OSTD

In this section, we present OSTD, which is our implementation of the framework required by a framekernel. First, we provide an overview of OSTD's APIs. Then, we describe how OSTD manages physical memory pages (frames), a key foundation of OSTD's soundness. Next, we define the key safety invariants for OSTD to achieve the language-based, intra-kernel privilege separation. Lastly, we introduce the technique of safe policy injection, which confines the complexity growth of OSTD as it evolves.

4.1 Expressive APIs

OSTD provides a small set of expressive abstractions to meet the needs of safe OS development. These abstractions are implemented as OSTD APIs, which are utilized by OSTD clients – safe kernel code that interacts with the OSTD. For **safe user-kernel interactions**, it allows an OSTD client to jump into the user space and execute until a trap occurs (`UserMode`), manipulate user-mode CPU registers (`UserContext`), and manage user address spaces (`VmSpace`). For **safe kernel logic**, it includes synchronization primitives (e.g., `SpinLock`, `Rcu`, `Mutex`, `WaitQueue`, and `CpuLocal`) and efficient data collection types (e.g., `LinkedList` and `RbTree`). For **safe kernel-peripheral interactions**, it offers mechanisms to register interrupt handlers (`IrqLine`), perform MMIO and PIO (`IoMem` and `IoPort`), and create coherent or streaming DMA mappings (`DmaCoherent` and `DmaStream`). All three of these OS development aspects may allocate and access physical pages, referred to as **frames**, via `Frame` (one frame) or `Segment` (multiple contiguous frames).

We illustrate a simplified workflow of how system calls may be handled safely with OSTD in Figure 3. Another example on how device data is requested with OSTD is shown in Figure 10 in Appendix A. For a more concrete example, see our sample project "Write a Hello World OS Kernel in ∼100 Lines of Safe Rust with OSTD" [23].

4.2 Frame Management

OSTD employs a **frame metadata system** to track the state of each page frame, including a reference count and a customizable metadata field. All per-frame metadata are stored in a large static array, which is allocated and initialized in an early bootstrap phase. The reference count equals the total number of `Frame` and `Segment` instances that refer to that frame, enabling OSTD to manage frame lifetimes correctly.

Inv. 1: *Any newly-allocated `Frame` or `Segment` originates from currently unused memory.*

OSTD further supports custom per-frame metadata by taking a type parameter `M` in `Frame<M>` (or `Segment<M>`). The client can attach a metadata object of `M` to a `Frame<M>` object, enabling features like a page cache to efficiently maintain some extra per-frame states (e.g., the status of data synchronization between memory and disk). This mechanism is also used in page allocators and slab allocators (see §4.4).

Both `Frame` and `Segment` can represent either sensitive or insensitive memory. Internally, OSTD uses them to allocate sensitive memory for kernel resources such as page tables, stacks, or slabs. Externally, an OSTD client can request a special form of insensitive memory called **untyped memory** with `UFrame<M> = Frame<M: AnyUFrameMeta>` or `USegment<M> = Segment<M: AnyUFrameMeta>`, where the trait `AnyUFrameMeta` marks metadata types suitable for untyped memory usage.

Untyped memory deals with the fact that externally modifiable memory (e.g., user-mapped or DMA-capable memory) cannot uphold the strong guarantees of Rust references or the type safety of arbitrary Rust types. As such, we design `UFrame` (or `USegment`) to have a **read-write style interface** [24] that only permits copying **plain old data** (POD) [4] from or into it. A POD type (e.g., `u32`) can hold a value of any bit pattern, without invalidating any Rust invariants.

4.3 Privilege Separation

Intra-kernel privilege separation requires *preventing all sensitive resources from being tampered with by non-TCB entities*, including safe clients, user programs, and peripheral devices. OSTD achieves this by enforcing the following key invariants.

Inv. 2: *Kernel-mode CPU states cannot be tampered with by* OSTD *clients.*

To expose only user-mode CPU operations to clients, OSTD provides `UserMode`, `UserContext`, and `VmSpace`. The first two handle traps and registers at user privilege level, while the third manages user-mode virtual memory. Kernel-mode registers are hidden, and only safe portion of user-mode CPU registers are accessible. For instance, on x86, `UserContext` exposes only the non-sensitive subset of `RFLAGS`, excluding bits like `IF` or `IOPL` that control interrupts or I/O privilege.

Inv. 3: *Kernel-mode CPU states cannot be tampered with by peripheral devices.*

On x86 hardware, devices can spoof exceptions, traps, or inter-processor interrupts [63]. To protect against such attacks, OSTD configures the IOMMU to enable interrupt remapping, preventing rogue devices from influencing kernel control flow.

Inv. 4: *Sensitive memory cannot be tampered with by* OSTD *clients.*

All `Frame`s or `Segment`s allocated by OSTD clients originate from insensitive physical memory. Sensitive pages, such as those used for kernel code, stacks, or page tables, remain fully within OSTD's control.

Each `Task`'s stack includes a guard page to detect stack overflows. If the stack pointer touches the guard page, a page fault is triggered, preventing further execution and thwarting potential malicious behavior. Additionally, the OSTD-based OS enforces a compile-time stack usage analysis, ensuring each function's stack frame remains smaller than the guard page. This prevents malicious attempts to bypass the guard page and exploit stack overflows.

Inv. 5: *Sensitive memory cannot be tampered with by user programs.*

User-mode mappings are created through `VmSpace`, which can only take `UFrame` or `USegment` as inputs. This design rules out exposing sensitive memory to user space because untyped memory are, by definition, insensitive.

Inv. 6: *Sensitive memory (including I/O memory) cannot be tampered with by peripheral devices.*

OSTD leverages IOMMU to prevent peripheral devices from writing to unauthorized physical regions. Initially, no part of physical memory is DMA-accessible. Drivers can create DMA mappings (`DmaStream` or `DmaCoherent`) only over untyped memory (`UFrame` or `USegment`), so sensitive regions stay protected.

Inv. 7: *Sensitive I/O memory or ports cannot be tampered with by* OSTD *clients.*

Clients interact with MMIO and PIO through `IoMem` and `IoPort`. OSTD uses information from the architecture, firmware (e.g., ACPI tables on x86), and core device drivers to label ranges of I/O memory and ports as either sensitive or insensitive. `IoMem` and `IoPort` can only be instantiated for insensitive regions, preventing accidental or malicious misuse of sensitive I/O registers.

Table 3: Increased complexities in some Linux components

Linux Components	Early Version (2.1.23, 1997)	Latest Version (6.12.0, 2024)	
Task scheduler	1.6 KLoC	27.2 KLoC	17×
Slab allocator	1.6 KLoC	8.7 KLoC	6×
Frame allocator	1.2 KLoC	7.1 KLoC	6×

4.4 Safe Policy Injection

One's confidence in the correctness of any safe abstraction ultimately hinges on the size and complexity of its TCB. In this section, we explain how to contain OSTD—even if its implementation adopts increasingly sophisticated strategies and policies over time.

Specifically, OSTD consists of the following components, each of which makes decisions based on particular strategies or policies: (1) **Task scheduler** determines which task to run next, (2) **Frame allocator** decides how to allocate large chunks of memory, and (3) **Slab allocator** decides how to allocate smaller chunks of memory. These components can grow significantly when equipped with advanced features. For instance, their counterparts in Linux have grown in size and sophistication over the years, as summarized in Table 3.

Intuitively, a TCB should contain only mechanisms rather than policies for minimality. We therefore propose **safe policy injection**, a technique that removes complex policies from a Rust TCB without compromising functionality, efficiency, or soundness. To apply the technique, a developer identifies components inside the TCB that might use complex strategies or policies, then determines whether a complete policy implementation can be written in safe Rust. If it can, the developer designs abstractions for acceptable policies plus APIs to register those policies, a process we refer to as "injection".

Although the idea of safe policy injection appears simple, ensuring soundness can be challenging because these policies affect the behaviors of TCB code. For example, a safe-but-buggy scheduler could inadvertently schedule the same task on two CPUs at once – a catastrophic mistake for memory safety. Similarly, the choice of which memory page or slot to allocate next directly affects memory safety. All mainstream OSes include frame and slab allocators within their TCBs for precisely these reasons. The rest of this subsection describes how we overcome such challenges.

4.4.1 Task Scheduler

Our goal is to support advanced schedulers, such as Linux's Completely Fair Scheduler (CFS) [5], atop OSTD. We introduce to OSTD two new traits: `Scheduler` and `RunQueue`. Their APIs are summarized in Table 4. A type implementing `Scheduler` should be registered once at an early stage of kernel initialization (when no tasks exist). Whenever a task becomes runnable (e.g., is spawned or woken), OSTD hands it over to the scheduler via the `enqueue` method.

Table 4: Select APIs for task scheduler injection. `Scheduler` represents a task scheduler, and each CPU's local run queue is represented by `RunQueue`.

APIs	Descriptions
`Scheduler::enqueue(&self, task)`	Enqueue a task
`Scheduler::local_rq_with(&self, closure)`	Access to the local run queue with a closure
`RunQueue::update_curr(&mut self)`	Update current task
`RunQueue::pick_next(&mut self)`	Pick the next task
`RunQueue::dequeue_curr(&mut self)`	Remove the current task

Table 5: Select APIs for frame allocator injection. The `FrameAlloc` trait abstracts any injectable frame allocator.

APIs	Descriptions
`FrameAlloc::alloc(&self, layout)`	Allocate frames
`FrameAlloc::dealloc(&self, addr, size)`	Deallocate frames
`FrameAlloc::add_free_memory(&self, addr, size)`	Add a range of usable frames

We require schedulers to be SMP-friendly by keeping per-CPU run queues, accessible through `local_rq_with`. To replace the current task, OSTD calls `pick_next`, and if the current task becomes unrunnable (e.g., it sleeps), `dequeue_curr` removes it from the queue. At each scheduling event (e.g., sleep, yield, or timer tick), OSTD invokes `update_curr` to notify the scheduler, which can then update the scheduling information about the current task.

For efficiency, the scheduler API takes ownership of runnable tasks (`Arc<Task>`). Because the task objects are clonable, they can be stored in advanced data structures (e.g., red-black trees) or moved between CPU queues for load balancing. Each task may also carry custom data (`Box<dyn Any>`) for storing scheduling attributes, making these attributes cheaply accessible.

The client-provided `pick_next` method could return an invalid task. In particular, returning a task that is already running on another CPU may lead to severe consequences, as running one task on two CPUs corrupts its stack. We must thus preserve this key invariant:

Inv. 8: *A `Task` runs on at most one CPU at any given time.*

To enforce this, OSTD gives each `Task` a private flag, `is_running`, which is checked and set prior to a context switch. After switching, the previous task will have its flag cleared.

4.4.2 Frame Allocator

An injectable frame allocator is abstracted by the `FrameAlloc` trait, whose APIs are shown in Table 5. During the initialization phase, OSTD passes the information of all usable physical memory to the injected frame alloca-

Table 6: Select APIs for slab allocator injection. A `Slab` represents one or more memory pages arranged into fixed-size slots, with free slots represented by `HeapSlot`.

APIs	Descriptions
`Slab::new()`	Create a new slab
`Slab::alloc(&self)`	Allocate a new `HeapSlot`
`Slab::dealloc(&self, slot)`	Recycle a `HeapSlot`
`HeapSlot::into_box(self, val)`	Convert into a heap object

tor (`FrameAlloc::add_free_memory`). Subsequently, whenever an OSTD client requests a new `Frame` (or `Segment`), OSTD redirects the request to the injected allocator (`FrameAlloc::alloc`), which returns the address of the allocated memory. Conversely, when a `Frame` (or `Segment`) is dropped and its reference count is reduced to zero, the underlying frames will be returned to the injected allocator (`FrameAlloc::dealloc`).

A `Frame` (or `Segment`) must be created out of a *valid* and *unused* range of physical memory. But this safety precondition may be violated by an injected frame allocator due to potential logical bugs. To guard against such bugs, OSTD *only* turns memory ranges obtained from the injected allocator into `Frame`s through a safe constructor method, `Frame::from_unused(addr, size)`, which enforces Inv. 1 by leveraging the frame metadata system (§4.2).

4.4.3 Slab Allocator

We now describe how OSTD supports a custom slab allocator [14, 15] and its injection as the heap allocator. A **slab** is one or more contiguous pages partitioned into an array of fixed-size **slots** [14]. Each slot can hold a single object of a particular type or size. A **slab cache** pools these slots for rapid allocation and release. Typically, a per-CPU free list tracks empty slots, refilled from a global pool of slabs when needed. If all slabs are full, new slabs are allocated from free pages. Conversely, the allocator can free unused slabs to reclaim memory. An OS typically manages multiple slab caches for different slot sizes, all governed by a **slab allocator**.

We introduce two new abstractions in OSTD: `Slab` and `HeapSlot`, whose APIs are summarized in Table 6. These APIs perform type conversions critical to memory safety: from an unused memory page to a slab (`Slab::new`), then to a free slot (`Slab::alloc`), and ultimately to a heap object (`HeapSlot::into_box`). With these abstractions, developers can implement slab caches and the slab allocator in ASTERINAS using only safe Rust.

These abstractions maintain some key invariants. For example, a `Slab` owns its underlying pages, so dropping the `Slab` should free those pages – unless some slots are still occupied by active objects. To avoid use-after-free, each `Slab` tracks the number of active `HeapSlot`s it spawned. A panic is triggered if a `Slab` is dropped while any slot remains active:

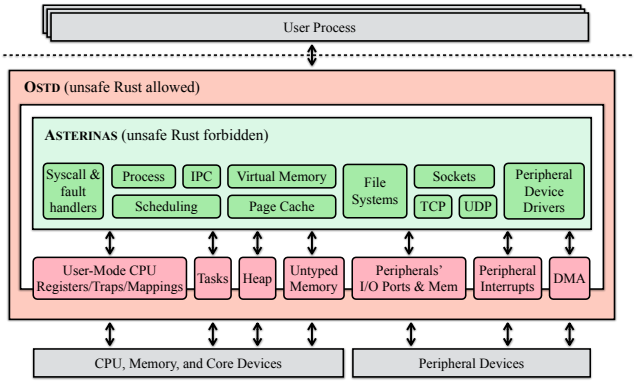


Figure 4: An overview of Asterinas.

Inv. 9: *A `HeapSlot` or any object derived from it must not outlive its parent `Slab`.*

Additionally, when `HeapSlot::into_box` is called, the method checks cheaply whether the slot's address and size can fit a requested object of type `T`:

Inv. 10: *An object is created from a `HeapSlot` only if the slot meets the object's size and alignment requirements.*

In practice, many kernel components rely on a global heap rather than creating their own slab caches. OSTD allows injecting a slab-based, global heap allocator, which can dispatch each heap allocation to the appropriate slab cache.

Overall, the safe policy injection technique allows OSTD to enjoy the performance advantage of advanced implementations without bloating its TCB size.

5 ASTERINAS

We develop ASTERINAS (see Figure 4), a framekernel-based OS built on top of OSTD. ASTERINAS implements a substantial subset of Linux features, including virtual memory, user processes, preemptive scheduling, IPC, a page cache, a virtual file system, and sockets, providing over 210 system calls. It supports various file systems (e.g., Ext2, exFAT32, OverlayFS, RamFS, ProcFS, and SysFS), socket types (e.g., TCP, UDP, Unix, and Netlink), and devices (e.g., Virtio Block, Virtio Network, Virtio Vsock, USB controller, and USB HID). Two CPU architectures are supported: x86-64 (tier-1) and RISC-V (tier-2). All the functionality is written in safe Rust by leveraging OSTD's APIs. ASTERINAS has been under development for three years (see Figure 7). The repository of ASTERINAS and OSTD is open-sourced [1, 2], contains over 100K lines of Rust contributed by over 50 individuals.

Towards smaller TCB. In pursuit of a smaller TCB, ASTERINAS (rather than OSTD) implements much of the OS infrastructure that Linux considers part of its core. For instance, ASTERINAS manages all interrupt bottom halves, such as softirq, tasklets, and work queues, by using an interrupt han-

dling hook provided by OSTD. OSTD enforces "atomic mode" to prevent client-provided callbacks from sleeping in interrupt context. ASTERINAS also maintains system time, monotonic time, and wall clocks by registering timer interrupts and reading the timestamp counter (TSC) through OSTD.

ASTERINAS takes advantage of OSTD's safe policy injection feature. It incorporates a Linux-style task scheduler with multiple scheduling classes—including a real-time scheduler and a rudimentary CFS [5]. ASTERINAS provides an efficient and scalable buddy system frame allocator, with per-CPU caching. ASTERINAS also features a slab allocator following the original design [14]. Our primary focus has been establishing the injection mechanism rather than extensively refining each scheduling or allocation policy; we plan to continue improving these components for greater maturity.

Performance optimization. Many performance bottlenecks in ASTERINAS and OSTD have been identified and addressed. Due to space constraints, we cannot cover every optimization in detail. In many instances, we have adapted strategies from Linux's optimized C implementations to align with the framekernel philosophy and the constraints of safe Rust.

One optimization in ASTERINAS is a pooling mechanism for DMA-able memory regions (akin to persistent mapping [61]), which may be requested frequently by device drivers. This approach minimizes the need for DMA mapping setup, requiring it only during initialization, thereby preserving IOTLB entries and enhancing the hit rates for IOMMU. In contrast, a dynamic mechanism would necessitate frequent unmaps, leading to IOTLB invalidation and a subsequent performance decline.

Thus far, our optimization efforts have been concentrated on single-core systems. Ongoing work is focused on improving SMP scalability through the use of SMP-friendly locks (e.g., RCU [45], MCS locks [35]) and the application of more fine-grained locking mechanisms. In this paper, our evaluation compares ASTERINAS directly with Linux in a single-core environment, with a comprehensive multi-core analysis planned for future work.

6 Evaluation

6.1 Performance Evaluation

We evaluate the performance of ASTERINAS. The experiments are performed on a machine with an Intel i7-10700 processor, 32GB of memory, and an Intel SSDPEKNW512GB solid state drive. The system software on the machine includes Ubuntu 22.04 (Linux version 6.8.0) and QEMU 9.1.0.

For comparison with ASTERINAS, we use Linux kernel version 5.15 as the baseline. Some Linux features that are missing in ASTERINAS, including CPU mitigations (`mitigations=off`) and huge pages (`hugepages=0`), are disabled to ensure fairness. Without disabling CPU mitigations, Linux's performance would be affected considerably.

Both the Linux and ASTERINAS kernels are tested within QEMU virtual machines (VMs) configured as follows: SMP set to 1, machine type set to q35 with the kernel IRQ chip in split mode, CPU specified as Icelake-Server, PCID disabled, and x2APIC and KVM enabled. Each VM is attached with a virtio-blk device (having a single queue with a length of 64) and a virtio-net device (a tap device with vhost support enabled), which will be used for block and network I/O benchmarks, respectively.

6.1.1 Micro-benchmarks

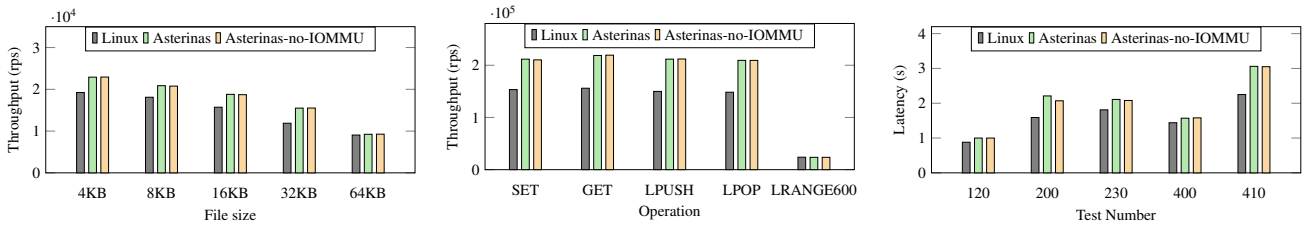
We run LMbench [46], a series of system call-intensive microbenchmarks. Specifically, the benchmarks are classified into five categories: process-related (Proc), memory-related (Mem), Inter-process communication-related (IPC), filesystem-related (FS), and network-related benchmarks (Net). The last column (Norm) of Table 7 shows the normalized performance—for throughput it shows results of Asterinas/Linux; for latency it shows results of Linux/Asterinas—and hence higher results suggest better performance of ASTERINAS. ASTERINAS is evaluated with IOMMU enabled and all reported results are averaged over ten runs of the benchmarks.

As shown in Table 7, the geometric mean of the normalized performance is 1.08, which means ASTERINAS slightly outperforms Linux in more benchmarks. This result suggests that ASTERINAS is comparable to Linux in performance, but does not necessarily mean ASTERINAS is better optimized. As a newly developed Rust kernel, ASTERINAS misses some of the advanced features and configurations in Linux. For example, in the network TCP tests, ASTERINAS uses the smoltcp [22] library, which lacks congestion control, enabling TCP to operate at full speed and resulting in faster performance compared to Linux. In the FS-related open and stat tests, Linux employs RCU-walk to achieve faster filename lookup, an optimization not implemented in ASTERINAS.

6.1.2 Macro-benchmarks

We evaluate ASTERINAS's performance with popular I/O-intensive applications, including Nginx (1.26.2), Redis (7.0.15), and SQLite (3.46.1). The results in Figure 5 report ASTERINAS's performance with IOMMU enabled (default) and disabled (for comparison).

- **Nginx.** We evaluate the throughput of Nginx using ApacheBench [26] with a concurrency level of 32 and a total of 200,000 requests. The results shown in Figure 5(a) indicate that the throughput of Nginx is higher on ASTERINAS (22,912 rps) than Linux (19,227 rps) when the requested file size is 4096 bytes (19% higher). When the file size is 64 KiB, ASTERINAS's throughput drops to 9,234 RPS, which is close to Linux's result. This drop may happen because ASTERINAS 's `sendfile` implementation is less optimized—it



(a) Nginx (b) Redis (c) SQLite

Figure 5: I/O-intensive application benchmarks with Ngnix, Redis, and SQLite.

Table 7: System call-intensive LMbench microbenchmarks.

	Command	Unit	Linux	Asterinas	Norm.[1]
Proc	`lat_syscall null`	μs	0.050	0.066 ± 0.001	0.76
	`lat_ctx 18`	μs	0.826	0.829 ± 0.019	1.00
	`lat_proc fork`	μs	59.20	57.46 ± 0.721	1.03
	`lat_proc exec`	μs	204.8	174.4 ± 2.156	1.17
	`lat_proc shell`	μs	319.3	294.3 ± 1.915	1.08
Mem	`lat_pagefault`	μs	0.109	0.100 ± 0.002	1.09
	`lat_mmap 4m`	μs	19.4	16.80 ± 0.422	1.15
	`bw_mmap`[2] `256m`	MB/s	15405	13197 ± 186.7	0.86
IPC	`lat_pipe`	μs	1.826	1.881 ± 0.009	0.97
	`bw_pipe`	MB/s	11133	14664 ± 1073	1.32
	`lat_fifo`	μs	1.825	1.938 ± 0.008	0.94
	`lat_unix`	μs	2.677	2.493 ± 0.023	1.07
	`bw_unix`	MB/s	7875	14183 ± 598.2	1.80
FS	`lat_syscall open`	μs	0.611	0.740 ± 0.020	0.83
	`lat_syscall read`	μs	0.081	0.088 ± 0.002	0.92
	`lat_syscall write`	μs	0.065	0.080 ± 0.003	0.81
	`lat_syscall stat`	μs	0.299	0.400 ± 0.009	0.75
	`lat_syscall fstat`	μs	0.263	0.231 ± 0.004	1.14
	`bw_file_rd`[3] `512m`	MB/s	10238	9198 ± 44.25	0.90
	`lmdd(Ramfs->Ramfs)`	MB/s	3219	2973 ± 18.60	0.92
	`lmdd(Ramfs->Ext2)`	MB/s	2490	2612 ± 57.08	1.05
	`lmdd(Ext2->Ramfs)`	MB/s	3453	2962 ± 30.77	0.86
	`lmdd(Ext2->Ext2)`	MB/s	2017	2626 ± 68.94	1.30
Net: Loop-back	`lat_udp`	μs	3.801	2.427 ± 0.009	1.57
	`lat_tcp`	μs	5.326	2.725 ± 0.016	1.95
	`bw_tcp 128`	MB/s	280.0	356.5 ± 78.94	1.27
	`bw_tcp 64k`	MB/s	6216	7647 ± 321.5	1.23
Net: VirtIO	`lat_udp`	μs	15.03	11.49 ± 0.139	1.31
	`lat_tcp`	μs	16.75	12.94 ± 0.135	1.29
	`bw_tcp 128`	MB/s	328.7	333.2 ± 5.141	1.01
	`bw_tcp 64k`	MB/s	1151	1116 ± 15.05	0.97
Mean					**1.08**

[1] Normalized performance. For throughput, use Asterinas/Linux; for latency, use Linux/Asterinas. The higher the better.
[2] With mmap_only.
[3] With io_only.

requires an extra copy to an intermediate buffer. For small files (4 KiB), ASTERINAS performs better than Linux because it lacks congestion control (using the smoltcp crate). However, as the file size increases, the overhead of redundant copying outweighs this advantage, causing ASTERINAS 's performance to fall behind Linux. The geometric mean of all the normalized performances (ASTERINAS/ Linux) is 1.17.

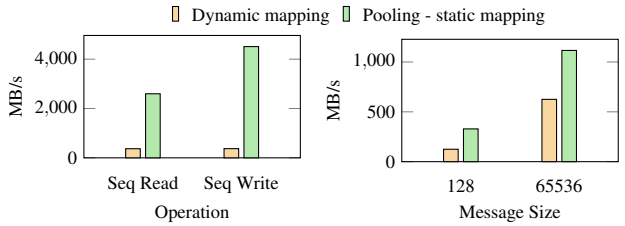
- **Redis.** We use the official Redis benchmark tool [53], which measures performance by executing a series of Redis commands and recording the throughput. Figure 5(b) shows a subset of representative commands, and the complete results can be found in Appendix B. The results indicate that ASTERINAS outperforms Linux. For instance, the throughput for GET operation on ASTERINAS is 218,670 rps, 40.2% higher compared to 155,994 rps on Linux. This result aligns with our LMbench findings, where ASTERINAS outperforms Linux for smaller packet sizes, as the Redis benchmark involves small message sizes in its operations. The geometric mean of all the normalized performance results is 1.31.

Table 8: Overhead due to OSTD's safety mechanism.

Operations	Sources of Safety Overheads	CPU Cycles Overhead / Total
Segment::read_bytes (4KB)	Boundary check	3/125(2.4%)
Segment::write_bytes (4KB)	Boundary check	2/239(0.8%)
IoMem::read_once (4 bytes)	Boundary check	170/10988(1.5%)
IoMem::write_once (4 bytes)	Boundary check	166/10666(1.6%)
KernelStack::new	Guard page creation	25/2950(0.8%)
Task::yield_now	Running flag check	1/167(0.6%)
FrameAlloc::alloc (1 frame)	Ownership check	12/180(6.7%)
Box::new (48 bytes)	Ownership check	1/148(0.7%)

- **SQLite.** We utilize SQLite's speedtest1 [51] as the benchmarking tool. In the test, we set the base size to 1000 to ensure data is written to the device (default is 100) and run the tests on an Ext2 mount over a virtio-blk device. Due to space limits, we only show representative results in Figure 5(c), whose test numbers are 120 (500000 unordered INSERTS with one index/PK), 200 (VACUUM), 230 (100000 UPDATES, numeric BETWEEN, indexed), 400 (700000 REPLACE ops on an IPK), and 410 (700000 SELECTS on an IPK). The complete results are shown in Appendix C. The geometric mean of all the normalized performances (Linux / ASTERINAS) is 0.85.

  In all these tests, ASTERINAS performs worse than Linux. The Vacuum test (200) shows the worst case, reaching only 72% of Linux's performance with IOMMU enabled. Once the IOMMU is disabled, the ratio will become 77%. To understand the root causes of Vacuum test, we used Strace [58] for performance diagnosis. The analysis revealed that Vacuum frequently makes `pwrite64` calls for writing 4-byte data. When disabling the SQLite journal, `pwrite64` with small data writes disappears, and ASTERINAS' performance increases to 80% and 83% of Linux's performance with IOMMU enabled and disabled, respectively. This analysis shows that ASTERINAS could benefit from optimization for both `pwrite64` operations with small data writes and IOMMU.



(a) Block device bandwidth (b) Network device bandwidth

Figure 6: IOMMU overhead with different mechanism

6.1.3 Overhead due to Safety Checks

The overhead introduced by the OSTD's safety mechanisms does not prevent ASTERINAS's performance from being comparable to a monolithic kernel. ASTERINAS has integrated several safety mechanisms into its OSTD to ensure system safety. To evaluate the cost of safety mechanisms, we directly invoke the interfaces corresponding to these safety mechanisms at the API level, measuring the CPU cycles required with and without the safety checks. The results are presented in Table 8. The "Allocate frame" test, which is related to the Frame allocator injection, exhibits the highest overhead at 6.7%, while the overhead of all other tests remains below 3%.

6.1.4 IOMMU Optimization

One optimization of IOMMU using the pooling mechanism leads to a significant performance gain. As shown in Figure 5, the results indicate that ASTERINAS does not introduce significant overhead for Nginx, Redis, and SQLite (in most cases). Both Nginx and Redis show almost no overhead, with the largest overhead observed in the SPOP test of Redis at 2.3%.

However, SQLite exhibits higher overhead in some cases (8.3%), primarily due to the incomplete pooling mechanism in the block device driver compared to the network device driver. Figure 6 further highlights the significance of pooling. We evaluate the bandwidth of block devices using FIO [13] and network devices using bw_tcp. Upon switching from polling to a dynamic mechanism, both network and block devices experienced considerable performance degradation.

6.2 TCB Evaluation

In this section, we evaluate the current TCB size of ASTERINAS and estimate its growth as the codebase evolves.

6.2.1 TCB Comparison

To assess the efficacy of ASTERINAS's implementation of the minimality principle, we compare the TCB size of ASTERINAS with RedLeaf [49], Theseus [16], and Tock [38].

Methodology. We employ a crate-level classification approach to evaluate the run-time TCBs of Rust-based OSes. In Rust, crates serve as the primary units for organizing and distributing code, with no additional isolation mechanisms within a crate. Consequently, a crate's interface naturally forms a trust boundary. So we consider a crate either belongs to the TCB, implying it must be trusted for security, or is excluded from the TCB, indicating it is secured by the Rust compiler. We adhere to the following rules to determine whether a crate is within the run-time TCB or not:

- **Rule 1**: The Rust toolchain is trusted. Consequently, crates provided by the Rust toolchain itself, like `alloc` and `core`, are *not* considered part of the run-time TCB.
- **Rule 2**: Crates containing `unsafe` code may potentially introduce soundness bugs, and as a result, they are considered part of the TCB.
- **Rule 3**: Crates that are dependencies of TCB crates should also be part of the TCB. This is because even if a crate doesn't use `unsafe` at all and thus doesn't introduce soundness issues on its own, the correctness of its APIs can still impact the soundness of the TCB crate that relies on it.

Since crates vary in size, evaluating the TCB size by counting the number of crates would be too simplistic and inaccurate. Therefore, a more refined metric is needed. However, directly comparing lines of code across crates is not ideal, as not all code within a crate is necessarily utilized during runtime. This is especially true for third-party dependencies, where often only a small fraction of the code is actually used at runtime.

To address this, we introduce a metric called **Linked Code Size (LCS)**, which measures the number of lines of code that are ultimately compiled and linked during the OS build.

We leverage the LLVM toolchain [42] to estimate the LCS for each OS. Specifically, we measure the number of source code lines that have corresponding statements in the generated LLVM IRs after link-time optimization. The content of the LLVM IR is closer to that of a binary file, primarily retaining the IR corresponding to executable statements within functions. It does not trace back to statements used for imports or struct definitions, which are not directly executable during runtime. This focus on executable IR provides a more accurate measure of the code that is relevant to the TCB during runtime operations.

Results. As illustrated in Table 9, the relative TCB size of ASTERINAS is a mere 14.0%, which is notably lower than that of other Rust-based OSes. While some OSes, such as RedLeaf, enforce restrictions that permit only safe Rust in specific crates, they still depend on numerous self-maintained crates that utilize `unsafe`, leading to a substantial portion of their code being part of the TCB (i.e., 66.1%).

Tock OS, being an embedded OS, is relatively lightweight, and thus its TCB size is slightly better than the other two. However, its TCB size (i.e., 43.8%) is still more than double of ASTERINAS. These results clearly underscore the effectiveness of the minimality principle that we adhere to.

Table 9: Comparison of TCB size

	RedLeaf	Theseus	Tock[1]	**Asterinas**
Total (LoC)	25992	70468	6628	**75285**
TCB (LoC)	17182	43978	2903	**10571**
Relative TCB	66.1%	62.4%	43.8%	**14.0%**

[1] Tock implementation of board `nrf52840dk`.

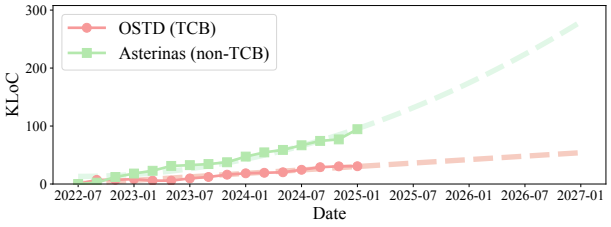


Figure 7: The codebase of ASTERINAS (non-TCB) has grown faster than OSTD (TCB) and is expected to remain so.

6.2.2 TCB Evolution

ASTERINAS has been under development for three years. We analyze and visualize the growth of its codebase (in KLoC). As shown in Figure 7, ASTERINAS (non-TCB) has experienced significant growth, while OSTD (TCB) has maintained a slower, steadier expansion. Curve fitting indicates that the TCB portion of the kernel will continue to grow in a controlled manner. This aligns with prior research on Linux codebase development [27, 57], which suggests that while overall codebases grow super-linearly with increasing complexity, core kernel functionality—excluding drivers—grows sub-linearly [34].

6.3 Safety Evaluation

To evaluate the soundness of OSTD, we develop a testing tool named KERNMIRI. This tool extends the official Rust UB detection tool, Miri [48], to support core OS concepts such as physical memory, page tables, etc. The overall architecture of KERNMIRI is depicted in Figure 8. It retains Miri's original UB detection logic (represented by grey boxes and lines) while introducing an additional 1,200 LoC to implement new components (indicated by green boxes and lines).

As illustrated in Figure 8, KERNMIRI enhances Miri by simulating physical memory and implementing a basic paging system. These improvements enable KERNMIRI to accurately interpret OS operations that demand fine-grained memory management. Moreover, KERNMIRI introduces additional shims to synchronize the OS state, which can be leveraged to direct KERNMIRI to activate a page table at a specific address or to inform KERNMIRI of changes in the state of a physical page, such as transitions between typed and untyped states.

KERNMIRI interprets the execution of OSTD and all its test cases. Following the workflow of OSTD's unit testing, KERNMIRI first interprets the initialization phase of OSTD to synchronize the system's initial memory state, and then

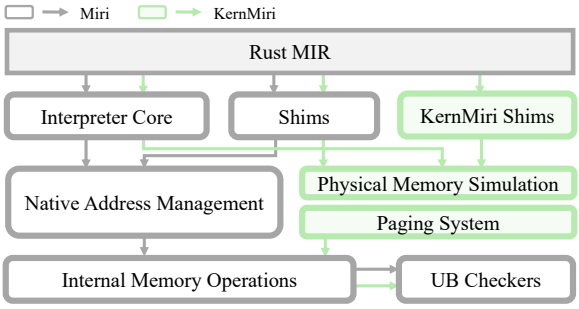


Figure 8: KERNMIRI, a UB detection tool for Rust OSes

Table 10: Coverage and efficiency of KERNMIRI on OSTD.

Modules	# Tests	Lines	Unsafe	Execution	
		Covered / Total	Covered / Total	Native	KernMiri
dma	12	385/443 (87%)	8/8 (100%)	0.25s	1.22s
frame	28	634/649 (98%)	41/41 (100%)	0.21s	3.14s
heap	6	278/319 (87%)	6/6 (100%)	0.01s	0.31s
kspace	8	287/323 (89%)	9/9 (100%)	0.04s	0.93s
page_table	34	931/1032 (90%)	46/46 (100%)	1.23s	34.83s
io	29	358/371 (97%)	23/23 (100%)	0.16s	3.12s
vm_space	17	662/672 (99%)	10/10 (100%)	0.28s	6.95s
All	**134**	**3535/3809** (93%)	**145/145**(100%)	**2.18s**	**50.50s**

executes each unit test of OSTD via interpretation. As such, the coverage of KERNMIRI depends on the unit test cases that have been developed for OSTD.

6.3.1 Coverage of KERNMIRI

We primarily use KERNMIRI to test and validate the `mm` module of OSTD, which is closely related to memory safety. This module also contains the majority of `unsafe` code in OSTD. Table 10 summarizes the measured metrics for each submodule of `mm` module separately, including the coverage of lines of code (Column `Line`), coverage of unsafe blocks (Column `Unsafe`) and interpreted execution time compared to native execution (Column `Execution`).

The results suggest that, with in total 134 unit tests, KERNMIRI covers all instances of `unsafe` code and over 90% on average of lines in all submodules. This confirms that the majority of the OSTD's memory operations is free of UBs. Furthermore, the few detected instances of UB have been addressed, with a detailed discussion provided in Section 6.3.2.

In addition, as shown in the last column of Table 10, KERNMIRI's interpreted execution takes approximately 25 times longer than normal execution. While significantly higher, this overhead is reasonable as a one-time cost for soundness evaluation. Thus, we conclude that KERNMIRI is a practical and effective tool for validating the soundness of Rust-based operating systems.

6.3.2 Case Studies

KERNMIRI has helped us detect several UBs that we would not be able to catch using other methods, significantly im-

```
/// Overview: 1 and 2 execute concurrently with 2 executing first. This
/// allows 1 to exchange successfully, leading to a race condition in
/// subsequent metadata modifications.

impl<M: FrameMeta> Frame<M> {
    pub fn from_unused(paddr: Paddr, metadata: M) -> Self {
        ...
① ---- ref_count.compare_exchange(0, 1, Acquire, Relaxed).expect(...);
        // Modify metadata.
    }
}

impl<M: FrameMeta> Drop for Frame<M> {
    fn drop(&mut self) {
② ---- let last_ref_cnt = self.ref_count().fetch_sub(1, Release);
        if last_ref_cnt == 1 {// Modify metadata.}
    }
}
```

(a) Data Race UB

```
/// Overview: The reference to the heap array is converted into a const
/// pointer for its first time pointer transition, yet the heap region
/// will be mutated later which results in mutability UB.

pub unsafe fn init() {
    static mut HEAP_SPACE: InitHeapSpace = InitHeapSpace([0; HEAP_SIZE]);
    HEAP_ALLOCATOR.init(HEAP_SPACE.0.as_ptr(), HEAP_SIZE);
}                                   └─ read-only transition
```

(b) Mutability UB

Figure 9: UB cases detected by KernMiri.

proves the soundness of our implementation of OSTD. We outline two representative UB cases in OSTD detected by KernMiri, as illustrated in Figure 9.

The first case involves a data race UB (Figure 9(a)). When creating a `Frame` using `from_unused` (see 4.4.2), if the `ref_count` corresponding to the physical address has just been decremented by a `drop` operation that has not yet completed, these two operations may concurrently modify the metadata, leading to data race UB.

The second case pertains to mutability UB (Figure 9(b)), where the code overlooks Rust's rules regarding the conversion of references to pointers. Specifically, when a reference is first converted into a pointer, the type of pointer determines the mutability constraints of the memory region. In this case a reference to the `HEAP_SPACE` is incorrectly converted into an immutable pointer during initialization. This conversion conflicts with subsequent mutable operations, resulting in UB.

7 Related Work

Safe language-based OSes. The development of operating systems using safe programming languages has been steadily increasing. Examples include Biscuit [19], implemented in Go; JX [28], developed in Java; MirageOS [43], built with OCaml; Verve [62], written in C#; and Singularity [33], which utilizes Sing# (an extension of C#). Rust has also become a popular choice for system development, as seen in projects like Theseus [16], Tock [38], RustyHermit [37], Redox [21], and rCore [52]. Asterinas is also implemented in Rust but goes further by leveraging Rust's type and memory safety features to achieve a minimal and sound TCB.

Intra-kernel privileged separation. Various mechanisms have been proposed to achieve privilege separation within the kernel. PerspicuOS [20] enforces separation using page table write protection and static analysis. RustyHermit-MPK [59], CubicleOS [56], and KDPM [36] employ Intel MPK for isolation. Lightweight Virtualized Domains [50] achieve separation through EPT and VMFUNC. RedLeaf [49], Tock [38], and Theseus [16] utilize Rust's safety features to create a new privilege level within the kernel. Asterinas achieves both soundness and minimality of intra-kernel privilege separation through a redefinition of the essential TCB functions.

Scheduler Injection. ghOSt [32] modifies Linux to delegate scheduling to userspace, incurring non-trivial overhead, while Enoki [47] adapts ghOSt's architecture but moves scheduler back to the kernel. In contrast, Asterinas offers (i) a smaller TCB—ghOSt relies on C-based run-queues, and Enoki can't catch all semantic bugs—and (ii) more flexible APIs, as Enoki schedulers must coordinate with Linux to avoid inconsistencies when managing run-queues, limiting independence.

8 Conclusions

This paper presents Asterinas, a Linux ABI-compatible OS kernel based on our novel *framekernel* architecture. By harnessing Rust's ownership and type-safety guarantees, a framekernel enforces *intra-kernel privilege separation* to achieve a small TCB in terms of memory safety. Using the APIs of OSTD (TCB), Asterinas (non-TCB) is implemented entirely in safe Rust, supporting over 210 Linux system calls. Our comprehensive evaluation shows that Asterinas delivers performance on par with Linux, demonstrating that a fully-featured, general-purpose OS can be both memory-safe and highly efficient.

9 Acknowledgments

The SUSTech authors are affiliated with the Research Institute of Trustworthy Autonomous Systems and in part supported by National Key R&D Program of China (No. 2023YFB4503902), CCF-AFSG RF20220012 and CCF-AFSG RF20230211. The Peking University authors are in part supported by National Science Foundation of China (No. 62032001). Asterinas has benefited from the collective expertise and support of numerous colleagues including Tao Xie, Bo An, Lingxin Kong, Yao Guo, Leye Wang, Huashan Yu, Jie Zhang, Wenfei Wu, Hui Xu, Zhaozhong Ni, Wei Zhang, Zhengyu He, Tao Wei, Lin Huang, Chuan Song, Yu Chen, Weijie Liu, Zhi Li, and many more. Special recognition is due to our open source contributors - Qing Li, Fabing Li, Shaowei Song, Qingsong Chen, Siyuan Hui, Zejun Zhao, Qihang Xu, Anmin Liu, Wang Siyuan, Wenqian Yan, Zhenchen Wang, Yingdi Shan, Hu Kang, Zhe Tang, Ruize Tang, and many more - whose technical dedication and collaborative spirit transformed this ambitious vision into a reality.

References

[1] The Asterinas Github repository. https://github.com/asterinas/asterinas.

[2] The ATC'25 artifact evaluation repository. https://github.com/asterinas/atc25-artifact-evaluation/.

[3] Rust for linux on wikipedia. https://en.wikipedia.org/wiki/Rust_for_Linux, 2024.

[4] cppreference . C++ named requirements: PODType (deprecated in C++20) - cppreference.com. https://en.cppreference.com/w/cpp/named_req/PODType.

[5] Linux . CFS Scheduler — The Linux Kernel documentation. https://www.kernel.org/doc/html/latest/scheduler/sched-design-CFS.html.

[6] The Rust Project Developers . Behavior considered undefined - The Rust Reference. https://doc.rust-lang.org/nightly/reference/behavior-considered-undefined.html.

[7] The Rust Project Developers . The borrow checker - Rust Compiler Development Guide. https://rustc-dev-guide.rust-lang.org/borrow_check.html.

[8] The Rust Project Developers . How Safe and Unsafe Interact - The Rustonomicon. https://doc.rust-lang.org/nomicon/safe-unsafe-meaning.html.

[9] The Rust Project Developers . Type checking - Rust Compiler Development Guide. https://rustc-dev-guide.rust-lang.org/type-checking.html.

[10] The Rust Project Developers . What is Ownership? - The Rust Programming Language. https://doc.rust-lang.org/book/ch04-01-what-is-ownership.html.

[11] America's Cyber Defense Agency. Widespread IT Outage Due to CrowdStrike Update. https://www.cisa.gov/news-events/alerts/2024/07/19/widespread-it-outage-due-crowdstrike-update, 2024.

[12] Vytautas Astrauskas, Christoph Matheja, Federico Poli, Peter Müller, and Alexander J. Summers. How do programmers use unsafe rust? *Proc. ACM Program. Lang.*, 4(OOPSLA):136:1–136:27, 2020.

[13] Jens Axboe. Flexible I/O Tester. https://github.com/axboe/fio, 2022.

[14] Jeff Bonwick. The Slab Allocator: An Object-Caching Kernel Memory Allocator. In *USENIX Summer*.

[15] Jeff Bonwick and Jonathan Adams. Magazines and Vmem: Extending the Slab Allocator to Many CPUs and Arbitrary Resources. In *USENIX ATC, General Track*.

[16] Kevin Boos, Namitha Liyanage, Ramla Ijaz, and Lin Zhong. Theseus: an experiment in operating system structure and state management. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 1–19. USENIX Association, November 2020.

[17] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. An empirical study of operating systems errors. *SIGOPS Oper. Syst. Rev.*, 35(5):73–88, October 2001.

[18] Jonathan Corbet. A first look at Rust in the 6.1 kernel. https://lwn.net/Articles/910762/, 2022.

[19] Cody Cutler, M. Frans Kaashoek, and Robert T. Morris. The benefits and costs of writing a POSIX kernel in a high-level language. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 89–105, Carlsbad, CA, October 2018. USENIX Association.

[20] Nathan Dautenhahn, Theodoros Kasampalis, Will Dietz, John Criswell, and Vikram Adve. Nested kernel: An operating system architecture for intra-kernel privilege separation. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, page 191–206, New York, NY, USA, 2015. Association for Computing Machinery.

[21] Redox Developers. Redox - your next(gen) os. https://www.redox-os.org/, 2024.

[22] Smoltcp Developers. Smoltcp: TCP/IP Stack for Embedded Rust. https://github.com/smoltcp-rs, 2025.

[23] The Asterinas developers. Example: Writing a Kernel in 100 Lines of Safe Rust - The Asterinas Book. https://github.com/asterinas/asterinas/blob/dec7ac1346649a0ef1a1256da258cd2f9f11ac4b/docs/src/ostd/a-100-line-kernel.md.

[24] The Asterinas developers. The reader-writer interface of untyped memory. https://github.com/asterinas/asterinas/blob/main/ostd/src/mm/frame/untyped.rs#L73.

[25] The Rust Project Developers. The Rustonomicon. https://github.com/rust-lang/nomicon, 2024.

[26] The Apache Software Foundation. ab - apache http server benchmarking tool. https://httpd.apache.org/docs/2.4/programs/ab.html, 2024.

[27] Michael Godfrey and Qiang Tu. Growth, evolution, and structural change in open source software. In *Proceedings of the 4th International Workshop on Principles of Software Evolution*, IWPSE '01, page 103–106, New York, NY, USA, 2001. Association for Computing Machinery.

[28] Michael Golm, Meik Felser, Christian Wawersich, and Jürgen Kleinoeder. The JX operating system. In *2002 USENIX Annual Technical Conference (USENIX ATC 02)*, Monterey, CA, June 2002. USENIX Association.

[29] Google. Android rust introduction. https://source.android.com/docs/setup/build/rust/building-rust-modules/overview.

[30] Internet Security Research Group. What is memory safety and why does it matter? https://www.memorysafety.org/docs/memory-safety/, 2025.

[31] Gary Guo. Klint: Compile-time detection of atomic context violations for kernel rust code. https://www.memorysafety.org/blog/gary-guo-klint-rust-tools/, 2023.

[32] Jack Tigar Humphries, Neel Natu, Ashwin Chaugule, Ofir Weisse, Barret Rhoden, Josh Don, Luigi Rizzo, Oleg Rombakh, Paul Turner, and Christos Kozyrakis. ghost: Fast & flexible user-space delegation of linux scheduling. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 588–604, New York, NY, USA, 2021. Association for Computing Machinery.

[33] Galen Hunt and Jim Larus. Singularity: Rethinking the software stack. *ACM SIGOPS Operating Systems Review*, 41(2):37–49, April 2007.

[34] Clemente Izurieta and James Bieman. The evolution of freebsd and linux. In *Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering*, ISESE '06, page 204–211, New York, NY, USA, 2006. Association for Computing Machinery.

[35] Theodore Johnson and Krishna Harathi. A simple correctness proof of the MCS contention-free lock. *Information Processing Letters*, 48(5):215–220.

[36] Hiroki Kuzuno and Toshihiro Yamauchi. Kdpm: Kernel data protection mechanism using a memory protection key. In *Advances in Information and Computer Security: 17th International Workshop on Security, IWSEC 2022, Tokyo, Japan, August 31 – September 2, 2022, Proceedings*, page 66–84, Berlin, Heidelberg, 2022. Springer-Verlag.

[37] Stefan Lankes, Jens Breitbart, and Simon Pickartz. Exploring rust for unikernel development. In *Proceedings of the 10th Workshop on Programming Languages and Operating Systems*, PLOS '19, page 8–15, New York, NY, USA, 2019. Association for Computing Machinery.

[38] Amit Levy, Bradford Campbell, Branden Ghena, Daniel B. Giffin, Pat Pannuto, Prabal Dutta, and Philip Levis. Multiprogramming a 64kb computer safely and efficiently. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 234–251, New York, NY, USA, 2017. Association for Computing Machinery.

[39] Hongyu Li, Liwei Guo, Yexuan Yang, Shangguang Wang, and Mengwei Xu. An Empirical Study of {Rust-for-Linux}: The Success, Dissatisfaction, and Compromise. pages 425–443.

[40] Hongyu Li, Liwei Guo, Yexuan Yang, Shangguang Wang, and Mengwei Xu. An empirical study of rust-for-linux: The success, dissatisfaction, and compromise. In Saurabh Bagchi and Yiying Zhang, editors, *Proceedings of the 2024 USENIX Annual Technical Conference, USENIX ATC 2024, Santa Clara, CA, USA, July 10-12, 2024*, pages 425–443. USENIX Association, 2024.

[41] Linux. Rust — The Linux Kernel documentation. https://docs.kernel.org/rust/index.html, 2024.

[42] LLVM Project. Llvm tools. https://llvm.org/docs/CommandGuide/, 2025. Accessed: 2025-01-14.

[43] Anil Madhavapeddy and David J. Scott. Unikernels: Rise of the virtual library operating system: What if all the software layers in a virtual appliance were compiled within the same safe, high-level language framework? *Queue*, 11(11):30–44, December 2013.

[44] A. Theodore Markettos, Colin Rothwell, Brett F. Gutstein, Allison Pearce, Peter G. Neumann, Simon W. Moore, and Robert N. M. Watson. Thunderclap: Exploring Vulnerabilities in Operating System IOMMU Protection via DMA from Untrustworthy Peripherals. In *Proceedings 2019 Network and Distributed System Security Symposium*, NDSS '19. Internet Society.

[45] Paul E. McKenney, Joel Fernandes, Silas Boyd-Wickizer, and Jonathan Walpole. RCU usage in the Linux kernel: Eighteen years later. *SIGOPS Oper. Syst. Rev.*, 54(1):47–63, August 2020.

[46] Larry W McVoy, Carl Staelin, et al. Lmbench: Portable tools for performance analysis. In *USENIX annual technical conference*, pages 279–294. San Diego, CA, USA, 1996.

[47] Samantha Miller, Anirudh Kumar, Tanay Vakharia, Ang Chen, Danyang Zhuo, and Thomas Anderson. Enoki: High velocity linux kernel scheduler development. In *Proceedings of the Nineteenth European Conference on Computer Systems*, EuroSys '24, page 962–980, New York, NY, USA, 2024. Association for Computing Machinery.

[48] Miri. Miri - an interpreter for rust's mid-level intermediate representation. https://github.com/rust-lang/miri, 2024.

[49] Vikram Narayanan, Tianjiao Huang, David Detweiler, Dan Appel, Zhaofeng Li, Gerd Zellweger, and Anton Burtsev. RedLeaf: Isolation and communication in a safe operating system. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 21–39. USENIX Association, November 2020.

[50] Vikram Narayanan, Yongzhe Huang, Gang Tan, Trent Jaeger, and Anton Burtsev. Lightweight kernel isolation with virtualization and vm functions. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '20, page 157–171, New York, NY, USA, 2020. Association for Computing Machinery.

[51] SQLite organization. Measuring and reducing cpu usage in sqlite. https://sqlite.org/cpu.html, 2024.

[52] rcore os. rcore. https://github.com/rcore-os/rCore, 2023.

[53] Redis. Redis benchmark. https://redis.io/docs/latest/operate/oss_and_stack/management/optimization/benchmarks/, 2024.

[54] The Register. Microsoft is busy rewriting core windows code in memory-safe rust. https://www.theregister.com/2023/04/27/microsoft_windows_rust/.

[55] Rust for Linux. Mutex::lock_noguard() may be unsafe · Issue #862 · Rust-for-Linux/linux. https://github.com/Rust-for-Linux/linux/issues/862.

[56] Vasily A. Sartakov, Lluís Vilanova, and Peter Pietzuch. Cubicleos: a library os with software componentisation for practical isolation. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, page 546–558, New York, NY, USA, 2021. Association for Computing Machinery.

[57] Walt Scacchi. Understanding open source software evolution. *Software Evolution and Feedback: Theory and Practice*, 9:181–205, 2006.

[58] Strace. Strace linux syscall tracer. https://strace.io/, 2024.

[59] Mincheol Sung, Pierre Olivier, Stefan Lankes, and Binoy Ravindran. Intra-unikernel isolation with intel memory protection keys. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '20, page 143–156, New York, NY, USA, 2020. Association for Computing Machinery.

[60] Rust Secure Code WG. RustSec Advisory Database. https://github.com/RustSec/advisory-db, 2024.

[61] Paul Willmann, Scott Rixner, and Alan L. Cox. Protection strategies for direct access to virtualized I/O devices. In *2008 USENIX Annual Technical Conference (USENIX ATC 08)*, Boston, MA, June 2008. USENIX Association.

[62] Jean Yang and Chris Hawblitzel. Safe to the last instruction: Automated verification of a type-safe operating system. In *PLDI*. Association for Computing Machinery, Inc., June 2010.

[63] Zongwei Zhou, Virgil D. Gligor, James Newsome, and Jonathan M. McCune. Building verifiable trusted path on commodity x86 computers. In *2012 IEEE Symposium on Security and Privacy*, pages 616–630, 2012.

A Another Example for OSTD APIs

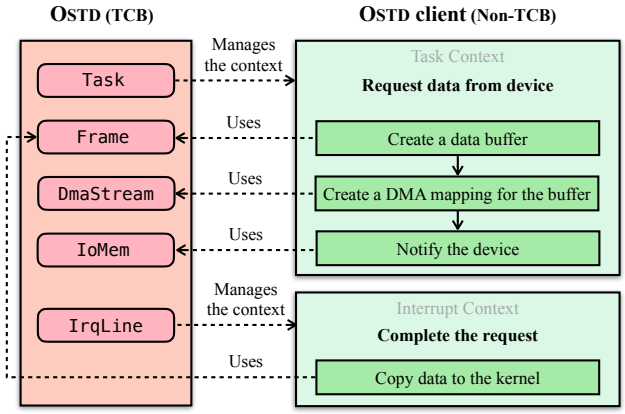


Figure 10: An example of OSTD API usage: request data from a device.

B Complete results for Redis benchmark

Table 11 displays the complete results from testing Redis using the redis-benchmark. The left column lists all the operations conducted in the benchmark, while the three columns

on the right present performance metrics for Linux, ASTERINAS, and ASTERINAS without IOMMU, respectively. The performance metrics are measured in requests per second (rps).

Operation	Linux(rps)	Asterinas(rps)	Asterinas no IOMMU(rps)
PING_INLINE	151022.44	213341.84	211694.22
PING_MBULK	157978.9	220975.64	218040.66
SET	153390.59	211647.77	210302.41
GET	155994.34	218670.04	219300.26
INCR	152132.93	219217.17	219302.27
LPUSH	149886.75	211691.68	211960.12
RPUSH	150504.81	214605.21	214053.79
LPOP	148347.63	209365.11	209308.7
RPOP	150713.57	210426.48	210138.93
SADD	156514.14	217682.08	217877.7
HSET	152276.36	209336.32	211663.65
SPOP	157350.85	217016.38	221988.13
ZADD	149385.73	206069.24	207479.97
ZPOPMIN	158360.84	219783.59	221895.17
LRANGE_100	92696.06	114471.67	113062.24
LRANGE_300	39268.41	39732.19	39629.18
LRANGE_500	27429.67	27843.41	27338.37
LRANGE_600	23876.49	23649.05	23674.88
MSET (10 keys)	125746.68	160040.56	157920.17

Table 11: Complete results of redis-benchmark.

Num.	Test Name	Linux (s)	Aster. (s)	Aster. no IOMMU(s)
100	500000 INSERTs into table with no index	0.27	0.33	0.32
110	500000 ordered INSERTS with one index/PK	0.43	0.49	0.49
120	500000 unordered INSERTS with one index/PK	0.88	1.00	1.00
130	25 SELECTS, numeric BETWEEN, unindexed	0.40	0.45	0.44
140	10 SELECTS, LIKE, unindexed	0.61	0.71	0.73
142	10 SELECTS w/ORDER BY, unindexed	1.17	1.35	1.34
145	10 SELECTS w/ORDER BY and LIMIT, unindexed	0.49	0.57	0.56
150	CREATE INDEX five times	0.95	1.16	1.13
160	100000 SELECTS, numeric BETWEEN, indexed	1.74	2.02	2.03
161	100000 SELECTS, numeric BETWEEN, PK	1.75	2.02	2.02
170	100000 SELECTS, text BETWEEN, indexed	1.72	2.06	2.03
180	500000 INSERTS with three indexes	2.14	2.41	2.42
190	DELETE and REFILL one table	2.09	2.38	2.38
200	VACUUM	1.59	2.21	2.07
210	ALTER TABLE ADD COLUMN, and query	0.04	0.04	0.04
230	100000 UPDATES, numeric BETWEEN, indexed	1.81	2.11	2.08
240	500000 UPDATES of individual rows	1.34	1.58	1.55
250	One big UPDATE of the whole 500000-row table	0.21	0.26	0.24
260	Query added column after filling	0.02	0.02	0.02
270	100000 DELETEs, numeric BETWEEN, indexed	2.26	2.63	2.58
280	500000 DELETEs of individual rows	2.19	2.6	2.58
290	Refill two 500000-row tables using REPLACE	3.85	4.31	4.22
300	Refill a 500000-row table using (b&1)==(a&1)	2.20	2.51	2.48
310	100000 four-ways joins	3.60	4.27	4.25
320	subquery in result set	7.14	8.3	8.35
400	700000 REPLACE ops on an IPK	1.44	1.57	1.58
410	700000 SELECTS on an IPK	2.25	3.06	3.05
500	700000 REPLACE on TEXT PK	1.66	1.82	1.85
510	700000 SELECTS on a TEXT PK	2.56	3.4	3.41
520	700000 SELECT DISTINCT	0.57	0.62	0.64
980	PRAGMA integrity_check	3.33	3.95	3.97
990	ANALYZE	0.20	0.22	0.22
	TOTAL	52.88	62.44	62.07

Table 12: Complete results of SQLite.

C Complete results for SQLite benchmark

Table 12 displays the complete results from testing SQLite with speedtest1. The first two columns indicate the test numbers along with their corresponding test names. The last three columns show the performance metrics for Linux, ASTERINAS, and ASTERINAS without IOMMU, measured in seconds.

Rex: Closing the language-verifier gap with safe and usable kernel extensions

Jinghao Jia*, Ruowen Qin*, Milo Craun†, Egor Lukiyanov†, Ayush Bansal*, Minh Phan*
Michael V. Le‡, Hubertus Franke‡, Hani Jamjoom‡, Tianyin Xu*, Dan Williams†

*University of Illinois Urbana-Champaign †Virginia Tech ‡IBM T.J. Watson Research Center

Abstract

Safe kernel extensions have gained significant traction, evolving from simple packet filters to large, complex programs that customize storage, networking, and scheduling. Existing kernel extension mechanisms like eBPF rely on in-kernel verifiers to ensure safety of kernel extensions by static verification using symbolic execution. We identify significant usability issues—safe extensions being rejected by the verifier—due to the *language-verifier gap*, a mismatch between developers' expectation of program safety provided by a contract with the programming language, and the verifier's expectation.

We present Rex, a new kernel extension framework that closes the language-verifier gap and improves the usability of kernel extensions in terms of programming experience and maintainability. Rex builds upon language-based safety to provide safety properties desired by kernel extensions, along with a lightweight extralingual runtime for properties that are unsuitable for static analysis, including safe exception handling, stack safety, and termination. With Rex, kernel extensions are written in *safe* Rust and interact with the kernel via a safe interface provided by Rex's kernel crate. No separate static verification is needed. Rex addresses usability issues of eBPF kernel extensions without compromising performance.

1 Introduction

Kernel extensibility is an essential capability of modern Operating Systems (OSes). Kernel extensions allow users with diverse needs to customize the OS without adding complexity to core kernel code or introducing disruptive kernel reboots.

In Linux, kernel extensibility has traditionally taken the form of loadable kernel modules. However, kernel modules are inherently *unsafe*—simple programming errors can crash the kernel. Despite the support of safe languages like Rust [23], there is no systematic support to ensure the safety of kernel modules—*unsafe* Rust code is allowed in kernel modules wherein checks to prevent errors are non-existent. Moreover, the vast, arbitrary interface exposed to kernel modules creates significant challenges in providing a safe Rust kernel abstraction to enforce safe Rust code [48, 76].

Recently, eBPF extensions have gained significant traction and become the *de facto* kernel extensions [5, 10]. Core to eBPF's value proposition is a promise of *safety* of kernel extensions, enforced by the in-kernel verifier. The verifier statically analyzes extension programs in eBPF bytecode, compiled from high-level languages (C and Rust). It performs symbolic execution against every possible code path in the bytecode to check safety properties (e.g., memory safety, type safety, termination, etc). The kernel rejects any extension the verifier fails to verify. Today, eBPF extensions have evolved far beyond simple packet filters (its original use cases [75, 77]) and are increasingly being used to construct large, complex programs that customize storage [40, 55, 102, 106], networking [107, 108], CPU scheduling [11, 60, 87], etc.

However, we observe that eBPF's static verifier introduces significant usability issues, making eBPF extensions hard to develop and maintain, especially for large, complex programs. For example, the eBPF verifier often incorrectly rejects *safe* extension code due to fundamental limitations of static verification and defects in the verifier implementation. When such false rejections happen, developers have no choice but to refactor or rewrite extension programs in ways that "please" the verifier. Such efforts range from breaking an extension program into multiple small ones, nudging compilers to generate verifier-friendly code, tweaking code to assist verification, etc (see §3). Some of the efforts also involve reinventing wheels and hacking eBPF bytecode, which creates significant cognitive overheads and makes maintenance difficult.

We argue that these usability issues are rooted in the gap between the programming language and the eBPF verifier, which we term the *language-verifier gap*. When writing eBPF programs, developers interact with the high-level language and naturally obey a *language contract* to align with the safety requirements of the language. The compiler also adheres to the language contract. Unfortunately, the verifier is not part of the language contract and has different expectations. As a result, verifier rejections may be surprising; the feedback (verifier log) is at the bytecode level and is hard to map to source code. As a result, developing eBPF extensions requires not only a deep knowledge of the high-level language and safety properties of kernel extensions but also a deep understanding of implementation details and quirks of the verifier.

Unfortunately, recent efforts to improve the eBPF verifier (e.g., via testing and verification [35, 97]) cannot fundamentally close the language-verifier gap because (1) they do not

address scalability issues of symbolic execution, so extension programs have to ill-fit the verifier's internal limits, and (2) it is unlikely that the language compiler (e.g., LLVM) and the eBPF verifier are always in synchronization, given their independent developments. Recent efforts to improve extension expressiveness via techniques like software fault isolation (e.g., KFlex [52]) largely inherit the eBPF verifier and, therefore, do not address the language-verifier gap.

We present Rex, a new kernel extension framework that closes the language-verifier gap and effectively improves the usability of kernel extensions, in terms of programming experience and maintainability. Rex builds up safety guarantees for kernel extensions based on safe language features. With Rex, safety properties are checked by the language compiler within the language contract. Rex drops the need for an extra verification layer and closes the language-verifier gap. We choose Rust as the safe language, as it is already supported by Linux [23] and offers desired language-based safety for practical systems programming [33, 36, 76, 82].

Rex kernel extensions are strictly written in *safe* Rust with selected features (unsafe Rust code is forbidden in Rex extensions). Rex transforms the promises of Rust into safety guarantees for extension programs with the following endeavors. First, to enable Rex extensions to be written entirely in safe Rust in the context of kernel extension, Rex develops a kernel crate and offers a safe kernel interface that wraps the existing eBPF kernel interface (eBPF helper functions and data types) with safe Rust wrappers and bindings. The kernel crate enforces memory safety, extends type safety, and ensures safe interactions with the kernel. Rex further enforces only safe Rust features through its compiler toolchains.

Moreover, Rex employs a lightweight extralingual runtime for safety properties that are hard to guarantee by static analysis. Specifically, Rex supports safe stack unwinding and resource cleanup upon Rust panics at runtime. Rex also checks kernel stack usage and uses watchdog timers to ensure termination with a safe mechanism. The Rex runtime is engineered with minimal overhead to achieve high performance.

We evaluate Rex on both its usability and performance. We show that by closing the language-verifier gap and offering Rust's rich built-in functionality, Rex effectively rules out the usability issues in eBPF. We further evaluate the usability by implementing the BPF Memcached Cache (BMC) [55] (a complex, performance-critical program written in eBPF) using Rex and show that Rex leads to cleaner, simpler extension code. We also conduct extensive macro and micro benchmarks. Rex extensions deliver the same level of performance as eBPF extensions—the enhanced usability does not come with a performance penalty.

Limitations. Rex's design comes with tradeoffs. To close the language-verifier gap and improve usability, Rex requires kernel extensions to be written in Rust, though its design principles apply to other safe languages. Rex brings the Rust toolchain into the Trusted Computing Base (TCB) and adds additional runtime complexity. Note that Rex extensions and eBPF extensions can co-exist—Rex and eBPF represent different tradeoffs. Rex targets large, complex kernel extensions for which usability and maintainability are critical, while small, simple extensions (e.g., packet filters) can still be written in eBPF, as their small amount of code and simple logic makes them less susceptible to the language-verifier gap.

Contributions. We make the following main contributions:

- A discussion of the language-verifier gap and its impact on the usability and maintainability of safe kernel extensions;
- Design and implementation of the Rex kernel extension framework, which closes the language-verifier gap by realizing safe kernel extensions upon language-based safety, together with efficient runtime techniques;
- The Rex project is at `https://github.com/rex-rs`.

2 Safety of Kernel Extensions

Safety is critical to OS kernel extensions—extension code runs directly in kernel space, and bugs can directly crash a running kernel. The eBPF verifier checks safety properties of extension programs in bytecode before loading them into the kernel to prevent programming errors such as illegal memory access. The verifier also checks the extension's interactions with the kernel via a bounded interface, defined by eBPF *helper functions*, to prevent resource leaks and deadlocks. We summarize the safety properties targeted by eBPF as follows:

- **Memory safety.** Kernel extensions can only access preallocated memory via explicit context arguments or kernel interface (helper functions), preventing NULL pointer dereferencing and corruption of kernel data structures. The eBPF verifier tracks the category of each value in the program to prevent dereference of invalid pointers (e.g., an arbitrary scalar value) as well as the size of the pointed memory to avoid out-of-bounds accesses.
- **Type safety.** When accessing data in memory, kernel extensions must use the correct types of data, avoiding misinterpretation of the data and memory corruption. The eBPF verifier checks the offset and size of memory accesses and ensures that they always match the underlying object.
- **Resource safety.** When acquiring kernel resources (e.g., locks, memory objects, etc.) through helper functions, kernel extensions must eventually invoke the appropriate interface to release the resources, preventing memory leaks or deadlocks that can crash or hang the kernel. In eBPF, the release of resources is checked by the verifier on all code paths. Since doing so is not enough to prevent deadlocks due to the possibility of circular wait, eBPF additionally restricts extensions to only hold one lock at a time.
- **Runtime safety.** Kernel extensions must terminate, with no infinite loops that can hang the kernel indefinitely. The verifier checks presence of backedges in the program and sets

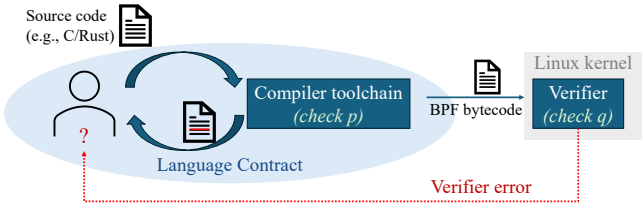


Figure 1: The language-verifier gap

a verification complexity limit on the number of bytecode instructions [92] it would explore to ensure termination.

- **Stack safety.** Kernel extensions must not overflow the limited and fixed-size kernel stack, avoiding kernel crashes or kernel memory corruptions. The verifier tracks stack usage of the program across function calls and rejects programs that use excessive amount of stack.

The eBPF verifier is complemented by the eBPF helper functions acting as the kernel interface, which allows interactions between the extension and the kernel. The helper function interface only contains around 200 routines, which is much more restrictive than that of the loadable kernel modules. The interface is designed to be high-level: helper functions are typically self-contained and encapsulate many low-level kernel operations, which in most cases eliminates the need of a protocol between extensions and the kernel. The simple and high-level nature of the interface makes it easy to reason about its safety—the eBPF verifier only needs to check value categories of the arguments passed for most helper functions. Helper functions are also the only means for extensions to acquire kernel resources. The interface defines both the acquisition and release routines for each resource, with the verifier ensuring the release routine is called before program exits.

Note that the above notion of safety in eBPF focuses on preventing programming errors that may crash or hang the kernel. Despite the discussions on whether security is a reasonable target [65, 73], in practice, eBPF and other extension frameworks (e.g., KFlex [52]) no longer pursue unprivileged use cases due to their inherent limitations (see detailed discussion in §4) [47, 53, 58]. Our work follows this safety model.

3 The Language-Verifier Gap

A fundamental problem of eBPF's safety verification mechanism is the *language-verifier gap* (illustrated in Figure 1). Developers mainly implement and maintain eBPF extensions in high-level languages (e.g., C and Rust) and compile them to eBPF bytecode; they agree to a contract with the high-level language (code has property p), which is enforced by the compiler. When a program fails to compile, developers receive feedback about how they violate the language contract. However, in eBPF, the compiled extension code will be further checked by the verifier. If correctly compiling code fails to pass the verifier (e.g., the code lacks property q, which is not part of the contract), it is difficult for developers to understand why the extension program fails despite obeying the contract.

Table 1: Patterns of common verifier workarounds

Category	Count
Refactoring extension programs into small ones	27
Hinting LLVM to generate verifier-friendly code	22
Changing code to assist verification	15
Dealing with verifier bugs	9
Reinventing the wheels	1

The language-verifier gap is further exacerbated when the verifier incorrectly rejects *safe* extension programs due to (1) scalability limitations of the symbolic execution used by the verifier, (2) conflicting analyses between the compiler and the verifier, and (3) the verifier's implementation defects. Today, the language-verifier gap forces developers to understand the verifier's internal implementation and its limitations and defects, and to revise extension code in ways that can pass the verifier at the bytecode level. Many such revisions are workarounds solely to please the verifier. Fundamentally, the language-verifier gap breaks the language abstractions and artificially forces extension programs in a high-level language to tightly couple with the low-level verifier implementation.

3.1 Verifier Workarounds

To understand the impact of the language-verifier gap, we analyzed commits related to revising eBPF programs to resolve verifier issues in popular eBPF projects, including Cilium [6], Aya [3], and Katran [16]. The commits were collected by searching through the commit logs of each project using keywords and manually inspected. In total, we collected 72 commits related to verifier issues. We also included two issues raised by BMC [55] and Electrode [107] in the discussion.

In all 72 commits, we confirmed that the original eBPF programs were safe but were rejected by the verifier due to defective or overly conservative safety checks. When the verifier rejects a safe program, the developer must find a *workaround*. Table 1 summarizes the workaround patterns.

Refactoring extension programs into small ones. The most common pattern (27 out of 72) is refactoring a large eBPF program, which the verifier rejects due to exceeding the verifier's internal limits into smaller ones. Since symbolic execution is hard to scale, the eBPF verifier imposes a series of limits on the complexity of extension programs (e.g., the number of bytecode instructions and branches [92]) to ensure verification completes at load time. The eBPF extension will be rejected if it exceeds any of these limits. Such rejections have no implication on the safety of the extension; rather, they are artifacts of scalability limitations of static verification.

We observe two standard practices of refactoring eBPF programs to work around verifier limits: (1) splitting eBPF

```
; return (void *)(unsigned long)ctx->data;
; LLVM generates a 32-bit load on ctx->data
2: (61) r9 = *(u32 *)(r7 +76)
; LLVM generates a 32-bit assignment, prompting
; the verifier to discard the pointer value type
3: (bc) w6 = w9
...
; now verifier treats it as an untrusted scalar
7: r2 = *(u8 *)(r6 +22)
Error: R6 invalid mem access 'inv'
```

(a) Verifier log showing an invalid memory access, which is hard to diagnose and does not directly map to the source code in C

```
static __always_inline void *
ctx_data (const struct __sk_buff *ctx) {
  void *ptr;
  /* prevent LLVM from generating 32-bit move */
  asm volatile (
    "%0 = *(u32 *)(%1 + %2)"
    : "=r"(ptr)
    : "r"(ctx),
      "i"(offsetof(struct __sk_buff, data))
  );
  return ptr;
}
```

(b) Inline assembly code created to work around the verification failure by preventing the compiler optimization

Figure 2: An example of the language-verifier gap from Cilium [41], where a safe eBPF extension is incorrectly rejected by the verifier (2a) and developers had to work around the problem by creating inline assembly code (2b).

programs into smaller ones and (2) rewriting eBPF programs with reduced complexity the verifier can handle.

We use BMC [55] as an example to explain these practices. BMC uses eBPF to implement in-kernel caches to accelerate Memcached. Conceptually, only two extension programs are needed (at ingress and egress, respectively). However, to satisfy the verifier limit, BMC developers had to split BMC code into *seven* eBPF programs connected via tail calls.[1] Such splitting creates an unnecessary burden on the implementation and maintenance of BMC; it also creates performance issues when states need to pass across tail calls (using maps).

Despite the smaller size of each program after splitting, BMC programs that iterate over the packet payload in a loop cannot easily pass the verifier. While the programs correctly check for the bounds of the payload, the programs result in an excessive number of jump instructions and exceed the verifier's complexity limit. As a workaround, developers must bind the size of the data BMC can handle further to pass the verifier. §7.1 revisits this example in more depth.

Hinting LLVM to generate verifier-friendly code Another common pattern is to change source code in ways that nudge the compiler (LLVM) to generate verifier-friendly bytecode. In several cases, LLVM generates eBPF bytecode that fails the verifier due to complex, often undocumented expectations of the verifier. Figure 2a shows a case from Cilium [41] that accesses a pointer field (`ctx->data`) in a socket buffer, defined as a 32-bit integer in the kernel `uapi` interface. LLVM generates a 32-bit load on `data` and assigns its value to another 32-bit register. While `data` is defined as 32-bit, under the hood it represents a pointer to the start of the packet payload. The 32-bit assignment made the verifier interpret the pointer as a scalar and incorrectly reject the program when it tries to access memory through the scalar. As a workaround, developers encapsulated access to `data` in inline assembly (Figure 2b) to prevent LLVM from generating 32-bit move as an optimization (LLVM does not optimize inlined assembly). The verifier then treats the register as a pointer rather than a scalar.

In another case [38], developers were forced to use `volatile` when loading from a 32-bit integer pointer and only using its upper 16 bits. Without `volatile`, LLVM optimized the code to only load the upper 16 bits from the pointer, which the verifier perceives as a size mismatch violation.

In fact, many eBPF programs today can only pass the verifier if compiled with `-O2` optimization—the verifier has a hardwired view of eBPF extension bytecode, which the compiler cannot generate with other levels, including `-O0`.

Changing code to assist verification. In this pattern, developers had to assist the verifier manually. A common pattern is refactoring the code into new functions when the verifier loses track of values in eBPF programs. It is often unclear what code needs to be refactored to pass the verifier, which significantly burdens developers. In a case from Cilium [86], developers had to refactor a network policy check into a separate function to change the program control flow that caused verifier to lose track of certain values in the program. We discuss examples of this category in more details in §A.

Dealing with verifier bugs. The language-verifier gap is further exacerbated by verifier bugs [35, 63, 72, 85], as developers need to acquire knowledge of the verifier's expectations and deficiencies. Moreover, different kernel versions can have different verifier bugs. Dealing with verifier bugs and maintaining compatibility across kernel versions is non-trivial. In a Cilium case [57], the verifier rejected a correct program with valid access to the context pointer due to the verifier's incorrect handling of constant pointer offsets. The verifier bug was known, but the fix was not present in all kernel versions. Cilium developers had to tweak their program to avoid the bug-triggering, yet correct, context pointer access so the code could verify on all kernel versions.

Reinventing the wheels. Developers may need to reimplement existing functions to pass the verifier. In Aya, the default definition of the `memset` and `memcpy` intrinsics provided by the language toolchain failed to pass the verifier [50]. Aya eventually implemented its own version for both intrinsics, using a simple loop to iterate over the data to avoid ever tripping the verifier. This case reflects a key challenge of using eBPF for large, complex extension programs, as developers may need to re-implement many standard, nontrivial library functions.

[1] Since BMC, the limit has increased, but the fundamental gap remains.

3.2 Implications

Our analysis shows that the language-verifier gap causes severe usability issues in developing and maintaining eBPF kernel extensions. eBPF developers have to implement arcane fixes and change their mental model to meet the verifier's constraints. If an eBPF extension fails to verify, the verifier log rarely pinpoints the root causes and cannot help trace back to the source code. Since it is hard to require compilers like LLVM to follow the eBPF verifier's implementations, we expect the language-verifier gap will continue to exist, especially for large, complex extension programs.

4 Key Idea and Safety Model

The key idea of Rex is to realize *safe* kernel extensions without a separate layer of static verification. Our insight is that the desired safety properties of kernel extensions can be built on the foundation of language-based properties of a safe programming language like Rust, together with extralingual runtime checks. In this way, the in-kernel verifier can be dropped, and the language-verifier gap can be closed. Rex extensions are strictly written in a *safe* subset of Rust. We choose Rust as the safe language for kernel extensions (instead of other languages like Modula-3 [34] and Sing# [54]) because Rust is already supported by Linux [45] and offers desired language features for practical kernel code [36,71,82]. Rex enforces the same set of safety properties eBPF enforces (§2). Hence, Rex extensions fundamentally differ from unsafe kernel modules.

Safety Model. Rex follows eBPF's non-adversarial safety model—the safety properties focus on preventing programming errors from crashing/hanging the kernel instead of malicious attacks. Like eBPF, Rex extensions are installed from a trusted context with root privileges on the system. Rex extensions can only be written in safe Rust with selected features and language-based safety is enforced by a trusted Rust compiler (§5.1). Unlike Rust kernel modules that can use unsafe Rust, the language-based safety of Rex extensions is strictly enforced. Other safety properties that are not covered by language-based safety (e.g., termination) are checked and enforced by the lightweight Rex runtime.

While historically eBPF supported unprivileged mode [47] and there are research efforts in supporting unprivileged use cases for kernel extensions [64, 65, 73], in practice, eBPF and other frameworks (e.g., KFlex [52]) no longer pursue it [53, 58]. The reasons come from inherent limitations of securing eBPF or kernel extensions in general.

First, it is hard for the eBPF verifier to prevent transient execution attacks like Spectre attacks completely, without major performance and compatibility overheads (see [53]). Specifically, new Spectre variants are being discovered; though many of them are bugs in hardware, they cannot be easily detected and fixed by static analysis [68]. Sandboxing techniques cannot completely prevent Spectre attacks either, e.g., SafeBPF [73] only prevents memory vulnerabilities, while

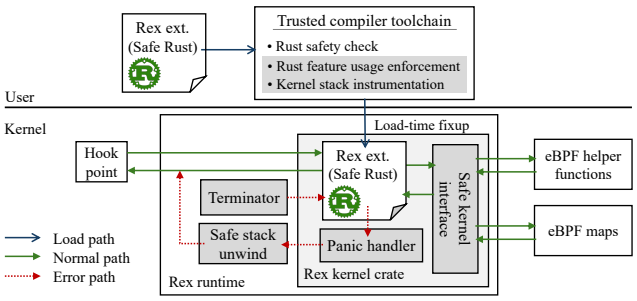


Figure 3: Overview of the Rex kernel extension framework. The gray boxes are Rex components.

BeeBox [65] only focuses on two Spectre variants and requires manual instrumentation of helper functions. For these reasons, the Linux kernel and major distributions also have moved away from unprivileged eBPF [25, 58, 80].

Second, eBPF chose not to be a sandbox environment (like WebAssembly or JavaScript) that does not know what code will be run [53]. Instead, the development of eBPF assumes that "*the intent of a BPF program is known* [53]."

Lastly, the constantly reported verifier vulnerabilities [63, 88, 91] indicate that a bug-free verifier is hard in practice.

Trusted Computing Base (TCB). With Rex's safety model, the TCB consists of the Rust toolchain, the Rex kernel crate, and the Rex runtime. Rex has to trust the Rust toolchain for its correctness to deliver language-based safety. We believe the need to trust the Rust toolchain is acceptable and does not come with high risks with our safety model. Recent work on safe OS kernels [36, 76, 82] makes the same decision to establish language-based safety by trusting the Rust toolchains. The active effort on extensive fuzzing and formal verification of the Rust compiler [12, 24, 66, 67, 69, 70] may further reduce the risk. Certainly, we acknowledge that the existing Rust compiler, such as rustc [30], is larger than the eBPF verifier.

5 Rex Design

The key challenge of the Rex design is to provide safety guarantees of kernel extensions (listed in §2) on top of Rust's safe language features (adopting a safe language alone is insufficient, as in Rust kernel modules).

Figure 3 gives an overview of the Rex framework. To realize language-based safety, Rex enforces kernel extensions to be strictly written in *safe* Rust with selected features. The Rex compiler toolchain rejects any Rex program that uses unsafe language features. Although this safe subset of Rust already provides inherent language-based safety within Rex extensions, eliminating undefined behaviors, safety of extensions is only achieved with the presence of safe kernel interactions provided by the *Rex kernel crate*. The kernel crate is trusted and bridges Rex extensions with unsafe kernel code. Rex builds on top of the eBPF helper function interface to provide a safe kernel interface for Rex extensions to interact with the

kernel using safe Rust wrappers and bindings. The safe interface encapsulates the interaction across the foreign function interface in the kernel crate. We reuse the eBPF helper interface, because it is designed for kernel extensions with a clearly defined programming contract and separates extensions from the kernel's internal housekeeping (e.g., RCU [32]).

Rex employs a lightweight extralingual in-kernel runtime that checks safety properties that are hard for static analysis. The runtime enforces program termination, kernel stack safety, and safe handling of runtime exceptions (e.g., Rust panics).

Although Rex reuses the existing hook points of eBPF, it assumes no nesting of extension programs, where another program executes on top of an existing program (e.g., by kprobing a helper function). This simplifies the reasoning of shared states and data races. At the same time, we note that uncontrolled program nesting is also unsafe in existing eBPF.

5.1 Safe Rust in Rex

Rex only allows language features that are safe in the context of kernel extensions. First, Rex excludes any *unsafe* Rust code as it misses important safety checks from the Rust compiler and can violate various safety properties (§2). Second, Rex forbids Rust features that interfere with Rust's automatic management of object lifetimes, which include `core::mem::{forget,ManuallyDrop}` and the `forget` intrinsic. These features are considered safe in Rust but violate resource safety of kernel extensions by facilitating resource leakage. Third, language features that cannot be supported in the kernel extension context are excluded by Rex. This group contains the `std` [26] crate and dynamic allocation support (not available in `no_std` build [20]), the floating point and SIMD support (generally cannot be used in kernel space), and the `abort` intrinsic (triggers an invalid instruction). Note that dynamic allocation may be supported by hooking the `alloc` [1] crate to the kernel allocator. We plan to explore the use of dynamic allocation in extensions in future works (§8).

To enforce the restrictions, Rex configures the Rust compiler and linter to reject the use of prohibited features. Specifically, we set compiler flags [17] to forbid unsafe code and unstable Rust features, which include SIMD and intrinsics. For individual langauge items such as `core::mem::forget`, we configure the Rust linter [7] to detect and deny their usage. We further remove floating point support by setting the target features [8] in Rex compilation. The `std` library and dynamic allocation are already unavailable in `no_std` environment used by Rex and therefore warrant no further enforcement.

5.2 Memory safety

Rex enforces that extensions access kernel memory safely. There are two common memory access patterns, depending on the ownership of the memory region: (1) memory owned by the extension (e.g., a stack buffer) is sent to the kernel through helper functions, and (2) memory owned by the kernel (e.g., a kernel struct) is accessed by the extension.

Memory owned by extensions. A Rex extension can allocate memory on the stack and send it to the kernel (e.g., asking the kernel to fill a stack buffer with data) via existing eBPF helper functions. Rex ensures no unsafe memory access and thus prevents stack buffer overflow and kernel crash (e.g., corruption of the return address on the stack).

Unlike eBPF, which checks a memory region with its size on every invocation of a helper function, in Rex, the strict type system of Rust already prevents unsafe access. Rex leverages the generic programming feature of Rust [13] to ensure that the size sent through the helper function interface is always valid. For helper functions that take in pointer and size as inputs, the Rex kernel crate creates an adaptor interface that parametrizes the pointer type as a generic type parameter. The interface queries the size of the generic type from the compiler and invokes the kernel interface with this size as an argument. Since Rust uses *monomorphization* [18], the concrete type and its size are resolved at compile time, adding no runtime overhead. In this way, the size is guaranteed to match the type statically and the kernel will never make an out-of-bound access. This works for both scalar types and array types. We use Rust's *const generics* to allow a constant to be used as a generic parameter [13] to encode array lengths.

Memory owned by the kernel. The kernel can provide extensions with a pointer to kernel memory (e.g., map value pointers and packet pointers). The extension must not have out-of-bound memory accesses. In eBPF, the verifier checks uses of kernel pointers with a static size, e.g. map value pointers (maps store the size of values); for pointers without a static size, like packet data pointers, the verifier requires extensions to explicitly check memory boundaries.

In Rex, pointers with static sizes are handled through the Rust type system. The kernel map interface of Rex encodes the key and value types through generics and returns such pointers to extension programs as safe Rust references. To manage pointers referring to dynamically sized memory regions, the Rex kernel crate abstracts such pointers into a Rust *slice* with dynamic size. Rust slices provide runtime bounds checks (§5.5), which allow the checking to happen without explicit handling by the extension.

Rust slices are in principle similar to `dynptr` in eBPF [96], but provide more flexibility. eBPF `dynptr`s are pointers to dynamically sized data regions with metadata (size, type, etc); however, access to the `dynptr`'s referred memory must be of a static size. Rust slices allow dynamically sized access to the underlying memory, benefiting from its runtime bounds checks. Moreover, the `bpf_dynptr_{read,write}` helpers do not implement a zero-copy interface like that available in Rust slices. While `bpf_dynptr_{data,slice}` helpers allow extensions to obtain data slices without copying, they again require explicit checks of the bound of the slice. As a tradeoff, eBPF `dynptr`s avoid runtime overheads of dynamic bounds checks, which we find negligible in our evaluation (§7.2).

5.3 Extended type safety

Rex extends Rust's type safety to allow extension programs to safely convert a byte stream into typed data. This pattern is notably found in networking use cases, where extensions need to extract the protocol header from a byte buffer in the packet as a struct. Safety of such transformations is beyond Rust's native type safety because they inevitably involve unsafe type casting. eBPF allows pointer casting; the verifier ensures: (1) the program does not make a pointer from a scalar value, and (2) the new type fits the memory boundary.

Rex also enforces the above two properties so that the reinterpreting cast (dubbed "transmute" in Rust) is safe. Rex extends Rust's type safety to cover such casts. To satisfy (1), Rex ensures the target Rust type of casting does not contain raw pointers or safe references in any transitively reachable fields. Rex introduces a Rust *auto trait* [2], `rex::NoRef`. The Rust compiler automatically implements an auto trait on a type unless the type is explicitly opted out via a *negative implementation* [19] or the type contains a field on whose type the trait is not implemented. We negatively implement `rex::NoRef` on the raw pointer and safe reference types of Rust, which ensures any type transitively containing a pointer or a reference will not have an implementation of the trait generated by the Rust compiler. Note that polymorphic types without statically known fields are not a problem because they take the form of *trait objects* [31] in Rust and can only exist behind pointers that already do not have an implementation of `rex::NoRef`. By requiring the target type of casting to implement `rex::NoRef` via trait bounds [4], (1) is effectively satisfied. To satisfy (2), Rex performs explicit bound checking in the transmute interface to ensure the target type always fits.

5.4 Safe resource management

Rex extensions are ensured to acquire and release resources properly to avoid leaks of kernel resources (e.g., refcounts and spinlocks). Different from eBPF where the verifier checks all possible code paths to ensure the release of acquired resources, Rex uses Rust's Resource-Acquisition-Is-Initialization (RAII) pattern [21]—for every kernel resource a Rex extension may acquire, the Rex kernel crate defines an RAII wrapper type that ties the resource to the lifetime of the wrapper object.

For example, when the program obtains a spinlock from the kernel, the Rex kernel crate constructs and returns a *lock guard*. The lock guard implements the RAII semantics through the `Drop` trait [9] in Rust, which defines the operation to perform when the object is destroyed. In the case of the lock guard, its `drop` handler releases the lock. Rex uses compiler-inserted `drop` calls at the end of object lifetime during normal execution, and implements its own resource cleanup mechanism (§5.5) for exception handling. The use of RAII automatically manages kernel resources to ensure safe acquisition and release. Extension programs do not need to explicitly release the lock or drop the lock guard.

5.5 Safe exception handling

While certain Rust safety properties are enforced statically by the compiler, the others are checked at runtime and their violations trigger exceptions (i.e., Rust panics). To handle exceptions in userspace, Rust uses the Itanium exception handling ABI [14] to unwind the stack. A Rust panic transfers the control flow to the stack unwinding library (e.g., llvm-libunwind), which backtracks the call stack and executes cleanup code and catch clauses for each call frame. Unfortunately, this ABI is unsuitable for kernel extensions:

- Unlike in userspace where failures during stack unwinding crash the program,[2] stack unwinding in kernel extensions cannot fail—kernel extensions must not crash the kernel and must not leak kernel resources.
- Unwinding generally executes destructors for all existing objects on the stack, but executing untrusted, user-defined destructors (via the `Drop` trait [9] in Rust) is unsafe.

Rex implements its own exception handling framework with two main components: (1) graceful exit upon exceptions, which resets the context, and (2) resource cleanup to ensure release of kernel resources (e.g., reference counts and locks).

Graceful exit. Rust invokes the panic handler upon a panic, which diverges the control flow from normal execution path. To gracefully exit the program upon a panic, the control flow must be redirected back to the normal execution path and with the correct context (e.g., stack pointer). To ensure such a graceful exit, Rex implements a small runtime (Figure 4), which consists of a program dispatcher and a landingpad in the kernel, as well as a panic handler in the Rex kernel crate. The dispatcher takes the duty of executing the extension program (like the eBPF dispatcher). It saves the stack pointer of the current context into per-CPU memory, switches to a dedicated program stack (§5.6), sets the termination state (§5.7), and then calls into the program. If the program exits normally, it returns to the dispatcher, which switches the stack back and clears the termination state. Under exceptional cases where a Rust panic is triggered, the panic handler releases kernel resources currently allocated by the extension, and transfer control to the in-kernel landingpad to print debug information to the kernel ring buffer and return a default error code to the kernel. Then, the landingpad redirects control flow to a predefined label in the middle of the dispatcher, where it restores the old value of the stack pointer from the per-CPU storage. This effectively unwinds the stack and resets the context as if the extension returned successfully.

Resource cleanup. Correct handling of Rust panics requires cleaning up resources acquired by the extension. However, static approaches that rely on the verifier to pre-compute resources to be released during verification (e.g., object table in [52]) do not apply to Rex due to the language-verifier gap.

[2] Theseus [36] implements stack unwinding in the kernel. But, it assumes that unwinding never fails; faults in unwinding result in kernel failures.

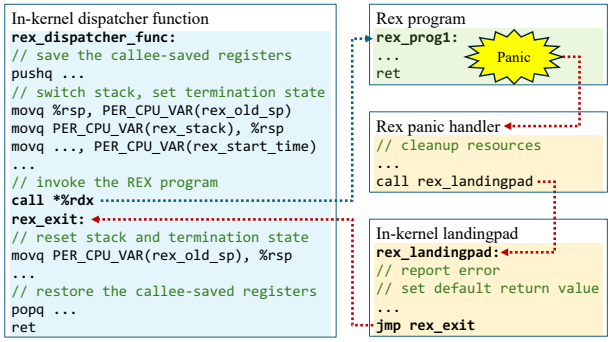


Figure 4: Exception handling control flow in Rex

Our insight is that extensions can only obtain resources by explicitly invoking helper functions. So, Rex records the allocated kernel resources during execution in a per-CPU buffer, which is in principle like the global heap registry in [82]. Upon a panic, the panic handler takes the responsibility to correctly release kernel resources, which involves traversing the buffer and dropping recorded resources.

We implement the cleanup code as part of the panic handler in the Rex kernel crate, as it is responsible for coordinating helper function calls that obtain kernel resources. Implementing the cleanup mechanism in the kernel crate ensures safety: as the code is called upon panic, it must not trigger deadlocks or yet another Rust panic to fail panic handling. To this end, we avoid using functions that may panic as much as possible and explicitly ensure the panic conditions cannot be met in case their usage is inevitable. This careful design of the Rex kernel crate frees the cleanup code and `drop` handlers of locks and panic-triggering code. Kernel functions invoked by such code may still hold locks internally, but they are self-contained and do not propagate to Rex (deadlocks in kernel functions is out of the scope of Rex). Rex does not execute user-supplied `drop` handlers upon panic, as they are not guaranteed to be safe (i.e., free of a nested panic) under panic handling context.

Rex implements a crash-stop failure model—a panicked extension is removed from the kernel. Any used maps and other Rex extensions sharing the maps will also be removed recursively. This prevents extensions sharing the maps from running in a potentially inconsistent state—exception handling in Rex already ensures the kernel is left in a good state.

5.6 Kernel stack safety

Kernel extensions should never overflow the kernel stack. Unlike userspace stacks which grow on demand with a large maximum size, the stack in kernel space has a fixed size (4 pages on x86-64). The eBPF verifier checks stack safety by calculating stack size via symbolic execution. However, it is reported that stack safety is broken in eBPF due to the difficulties of statically analyzing indirect tail calls [91] and uncontrolled program nestings [43].

Our insight is that stack safety can be enforced at compile time to avoid runtime overhead if the extension program has no indirect or recursive calls, as the stack usage can be statically computed. Otherwise, it is easy to check stack safety at runtime. Rex, therefore, takes a hybrid approach and selects between static and dynamic checks based on the situation.

Static check. The static check is done by a Rex-specific compiler pass (§6). If the extension has no indirect or recursive calls, its total stack usage can be calculated by traversing its global static callgraph and sum up the size of each call frame. We turn on fat LTO and use a single Rust codegen unit [8] for Rex programs to ensure the compiler always has a global view across all translation units.

Runtime check. For extensions with indirect or recursive calls, it is hard to calculate the stack usage from the callgraph due to the presence of unknown edges (indirect calls) and cycles (recursive calls). Under these cases, Rex performs runtime checks. The Rex compiler pass first ensures each function in the program takes less than one page (4K) of stack. This is more relaxed than the frame size warning threshold (2K) in Linux and ensures enough stack to handle Rust panics. Before each function call in the extension, the compiler inserts a call to the `rex_check_stack` function from the kernel crate to check the current stack usage: if the stack usage exceeds the threshold, it will trigger a Rust panic and terminate the program safely (§5.5).

To manage stack usage of Rex extensions effectively, Rex implements a dedicated kernel stack for each extension. The dedicated stacks are allocated per-CPU and virtually mapped at kernel boot time with a size of eight pages. Before executing a Rex extension, the dispatcher (Figure 4) saves the stack pointer of the current context, and then sets the stack and the frame pointer (already saved with other callee-saved registers) to the top of the dedicated stack. When the extension exits, the original stack and frame pointers are restored.

Rex sets the stack usage threshold to be four pages for extension code; it reserves the next four pages with following considerations: (1) helper functions are not visible at compile time but they also account for stack usage during execution; we use four pages as the de facto stack size used by the kernel itself, and (2) since stack usage of each function is limited to one page of stack, in the worse case, the remaining stack space is at least three pages when `rex_check_stack` triggers a Rust panic. Since the panic handler is implemented in the kernel crate and does not change with programs, this worse-case guarantee empirically leaves enough space for panic handling and stack unwinding. Rex's dynamic approach achieves stronger stack safety than that of eBPF.

5.7 Termination

Termination is an important property of kernel extensions. In eBPF, an extension with a back edge or exceeds the instruction limit will be rejected, regardless whether it eventually terminates. Since it is challenging to statically reason about termination of an arbitrary program, the eBPF termination

semantic creates many false positives and greatly contributes to the language-verifier gap. KFlex [52] lifts the back edge restriction by inserting cancellation points in eBPF bytecode on all back edges during verification, which triggers termination at runtime. However, back edge analysis is non-trivial outside eBPF bytecode and is unreliable for general Rust programs.

Instead of following eBPF's termination semantic that contributes to the language-verifier gap, Rex employs a termination semantic based on extension execution time. Rex implements termination support in its runtime, which interrupts and terminates extensions that run for too long. Rex limits the run time of extensions by leveraging kernel timers as watchdogs. Rex builds the runtime on the high resolution timer (`hrtimer`) subsystem in Linux [56]. Since `hrtimer` callbacks execute in hardware timer interrupts, they are capable of interrupting the contexts in which most extensions execute (soft interrupts and task context [42]). Since hardware timer interrupts are periodically raised by the processor, regardless whether an `hrtimer` is present, executing timer callbacks in this existing hardware timer interrupt adds no extra interrupt or context switch, keeping the watchdog overhead minimal.

Rex sets one timer for each CPU to avoid inter-core communication, in contrast to using a single, global timer to handle programs from all CPUs. Each timer only needs to monitor extensions running on the core. Rex arms the timers at kernel boot time, which are triggered periodically with a constant timeout, and re-armed each time after firing.[3]

Rex implements its watchdog logic in the timer handlers. When a timer fires, its handler suspends any soft interrupt or task context, and saves its registers. The handler then checks the current CPU on whether the termination timeout of the Rex extension in the stopped context has been reached. This is done by comparing the extension start time (stored as a per-CPU state as shown in Figure 4) with the current time. If the extension exceeds the threshold, the timer handler overwrites the saved instruction pointer register to the panic handler (§5.5). After returning from the timer interrupt, the extension executes its panic handler, which cleans up kernel resources and gracefully exits. Rex sets both the timer period and runtime threshold to the default RCU CPU stall timeout (Rex extensions run in RCU locks as they use eBPF hook points).

Rex defers termination when the extension is running kernel helper functions to avoid disrupting the kernel's internal resource bookkeeping; it also does not terminate an extension if it is in the panic handler. Rex uses a per-CPU tristate flag to track the state of an extension: (1) executing extension code, (2) executing kernel helpers or panic handlers, and (3) termination requested. A helper call changes the state from 1 to 2. When executing the timer handler, if the flag is at state 2, the termination handler modifies it to state 3 without changing the instruction pointer. When a helper returns, if the flag is at state 3, the panic handler is called to gracefully exit.

A corner case of this design is deadlock. Since spinlock acquisition in Rex is implemented by a kernel helper function, a deadlocked program will never return from the helper, and therefore will never be terminated properly. Rex follows eBPF's solution toward deadlocks, where a program can only take one lock at a time. This is achieved by using a per-CPU variable to track whether the program currently holds a lock—a program trying to acquire a second lock will trigger a Rust panic. We note that if the ability of holding multiple locks at the same time is desired, the kernel spinlock logic can be modified to check the termination state of Rex programs during spinning and terminate a deadlocked program accordingly.

Limitation. Rex uses hard interrupts, and thus cannot interrupt extensions that are already executing in hard or non-maskable interrupts [42] (e.g., hardware perf-event programs). Such extensions are not targeted by Rex, as they are supposed to be small, simple, and less likely to encounter the language-verifier gap. Note that Rex extensions and eBPF extensions are not mutually exclusive and can co-exist.

Moreover, the termination of a timed-out Rex extension may be delayed if the extension is already interrupted by another event when the timer triggers (the extension registers will not be available to the timer handler). Rex needs to wait for a triggering of the timer that directly interrupts the extension. However, to date, we have never encountered such delayed termination in our experiments (§7).

6 Implementation

We implement Rex on Linux v6.11. Rex currently supports five eBPF program types (tracepoint, kprobe, perf-event, XDP, and TC) and shares their in-kernel hookpoints. Rex only includes helpers for kernel interactions. Helpers introduced due to contraints posed by the eBPF verifier (e.g., `bpf_loop`, `bpf_strtol`, and `bpf_strncmp`) are entirely excluded by Rex.

Kernel crate. The Rex kernel crate is implemented in 3.5K lines of Rust code, among which 360 lines are unsafe Rust code. The kernel crate contains the following components:

- *Helper function interface* in Rex is implemented on top of eBPF helpers, with wrapping code that allows Rex extensions to invoke helpers with safe Rust types.
- *Kernel data-type bindings* are generated for the extension to access kernel data types defined in C. Rex uses rust-bindgen [22] to automatically generate kernel bindings and integrates it into the build process of extensions. Rex programs need to be rebuilt for each kernel they target to account for ABI differences in kernel data types.
- *Program context* in Rex is wrapped in a Rust struct, which marks the context as private and implements getter methods for its public fields.

Kernel support. Rex implements the extension load code and the runtime in the kernel in 2.2K lines of C code on

[3] Disarming the timer when no extension is running saves CPU cycles, but incurs high overhead due to timer setup on the hot path of extension execution, especially for frequently triggered extensions (e.g., XDP extensions [5]).

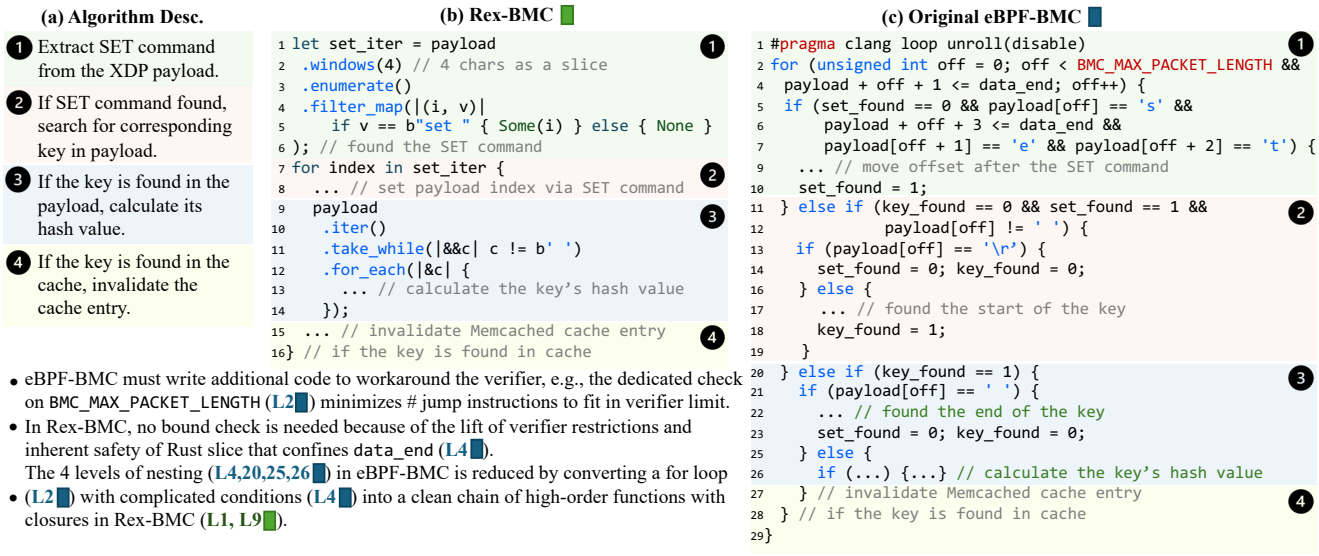


Figure 5: Cache invalidation implementation of Rex-BMC and eBPF-BMC; Rex leads to cleaner, simpler code.

vanilla Linux. To load an extension, the kernel parses the ELF executable of the extension and locates all the `LOAD` segments in the executable. It then allocates new pages and maps the `LOAD` segments into the kernel address space based on the size and permissions of the segments. The load function is responsible for fixups on the program code to resolve referenced kernel helpers and eBPF maps. The Rex runtime in the kernel consists of the stack unwinding mechanisms (§5.5), support for dedicated kernel stack (§5.6) and termination (§5.7).

Compiler support. Rex implements a compiler pass for Rex-specific compile-time instrumentations on the stack (§5.6). We take advantage of Rust's use of LLVM [29] as its default code generation backend and implement the pass in LLVM. A Rex-specific compiler switch is also added to the Rust compiler frontend (rustc [30]) to gate the Rex compiler pass.

7 Evaluation

We evaluate Rex in terms of its usability and performance (with both macro and micro benchmarking).

7.1 Usability

Measuring usability is challenging. We evaluate Rex in two ways: (1) heuristic evaluation on whether it saves workarounds to the language-verifier gap, and (2) our dog-fooding experience of using Rex to implement a large, complex extension (BMC [55]). Overall, we find that Rex enables developers to write simpler and cleaner code.

Eliminating workarounds. Because Rex introduces no language-verifier gap, none of the workarounds in §3 is needed in writing Rex extensions.

- Rex extensions have no limit on program size and complexity. There is no need to artificially refactor extension programs into smaller or simpler ones (§3.1).
- There is no need to artificially make Rex extensions verifier-friendly (§3.1). In fact, by decoupling static analysis from the kernel, Rex can enable new analysis (e.g., by allowing compilers to optimize for extra analysis/verification [99]).
- For the same reason, developers no longer need to tweak code to assist verification (§3.1).
- Developers no longer need to manage different verifier bugs across kernel versions (§3.1). The Rust compiler can have bugs and break safety guarantees, but it is arguably easier to upgrade than the kernel for fixes.
- Rex enables developers to use rich builtin intrinsics defined by the Rex toolchain without reinventing wheels (§3.1).

Case study: Rex-BMC We rewrite BMC [55] as a Rex kernel extension (Rex-BMC), which was originally written in eBPF extensions (eBPF-BMC). Rex-BMC is not a line-by-line translation of eBPF-BMC, because Rex provides a more friendly programming experience (e.g., no need to split programs due to the verifier limit; see §3.1). Rex-BMC covers all safety aspects discussed in §5. The extension accesses packet payloads as Rust slices that provide memory safety (§5.2) and safe exception handling (§5.5), leverages extended type safety (§5.3) from Rex to reinterpret payload bytes into Memcached headers, and uses spinlocks that are safely managed via RAII (§5.4). Kernel stack checks (§5.6) and termination support (§5.7) are also covered by Rex-BMC as they are not bound to specific program implementations. Rex-BMC does not contain indirect calls, and thus does not invoke runtime stack checks. Additionally, Rex-BMC uses four helper functions from the Rex kernel crate to perform map accesses and packet manipulations. In this section, we discuss Rex-BMC from the usability perspective and measure its performance in §7.2.

Our experience shows that Rex enables cleaner and simpler extension code, compared to eBPF. Essentially, Rex enables

us to focus on key program logic without the overhead of passing the verifier. For example, we no longer need to divide code into in parts, add auxiliary code to help the verifier, dealing with tail calls and state transfer, etc. In addition, we can directly use Rust's builtin language features and libraries (e.g., iterators and closures). As one metric, Rex-BMC is written in 326 lines of Rust code. In comparison, eBPF-BMC is written in 513 lines of C code (splitting into seven programs).

Figure 5 compares the code snippets of eBPF-BMC and Rex-BMC that implement cache invalidation, respectively, as a qualitative example. The checks in eBPF-BMC code, required by the eBPF verifier, including these for offset and `data_end` limits, are now being enforced via the inherent language features of Rust, such as slices with bound checks in Rex (L2 and L10). The check on `BMC_MAX_PACKET_LENGTH`, which serves as a constraint to minimize the number of jump instructions to circumvent the eBPF verifier, is no longer needed. Other checks for identified SET commands and loops states can be implemented with built-in functions and closures in an easy and clean way (L4–L6 and L11).

Moreover, with the elimination of program size and complexity limits in Rex-BMC, developers no longer have to save the computation state in a map across tail calls, which leads to a clearer and more efficient implementation.

Note that the usability benefit does not come from the expressiveness difference between Rust and C, but from closing the language-verifier gap via Rex. Evidently, the cleaner code of Rex-BMC would fail the verifier if it were to be compiled into eBPF (e.g., via Aya [3]): the compiler is unable to generate verifier-friendly code for convenient language features such as slices, and the verifier complexity limits will always be an issue. Rex allows us to fully leverage Rust's expressiveness without being constrained by verification issues.

7.2 Macro benchmark

Rex's usability benefits do not come with a performance cost. We show that Rex extensions deliver comparable performance as eBPF extensions. Rex-BMC achieves a throughput of 1.98M requests per second (RPS) on 8 cores, which is slightly higher than eBPF-BMC (1.92M).

Our setup consists of a server machine and a client machine. The server machine runs the Rex custom kernel based on Linux v6.11.0 on an AMD EPYC 7551P 32-Core processor with 112 GB memory without SMT and Turbo. The client machine runs a vanilla v6.11.0 Linux kernel on an AMD Ryzen 9 9950X processor with 96 GB memory. Both machines are equipped with Mellanox ConnectX-3 Pro 40GbE NICs and are connected back-to-back using a single port.

We evaluate the throughput of (1) Memcached which binds multiple UDP sockets to the same port [55], (2) Memcached with eBPF-BMC, and (3) Memcached with Rex-BMC. For each setup, we vary the number of CPU cores for Memcached server and NIC IRQs and pin one Memcached thread onto each available core. We use the same workloads as in BMC [55], albeit with a smaller number of Memcached keys.

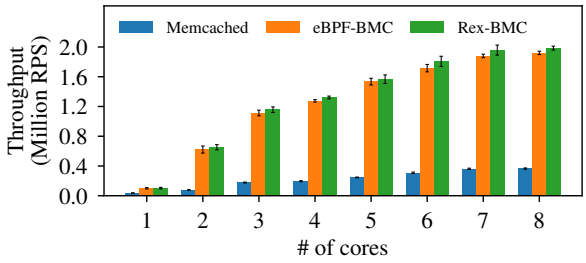


Figure 6: Throughput of Memcached, with eBPF-BMC, and with Rex-BMC under different number of cores.

Figure 6 shows the throughput of the three setups under different numbers of CPU cores. Memcached processes all requests in userspace with the overhead of the kernel network stack, achieving only 37K RPS on a single core and 365K RPS on 8 cores. Both eBPF-BMC and Rex-BMC achieve a much higher throughput as they process a large fraction of requests at NIC driver level without going through the kernel network stack. With 8 cores, eBPF-BMC and Rex-BMC achieve a throughput of 1.92M and 1.98M, and a performance benefit of 5.26x and 5.43x, respectively. The slight performance improvement over eBPF is attributable to the elimination of overheads of tail calls and associated state-passing via maps, along with optimizations in the rustc frontend and x86 backend, despite the overhead of additional runtime checks.

7.3 Micro benchmark

Several of Rex's design decisions could introduce overheads, despite being invisible in the Rex-BMC evaluation. We use specially designed microbenchmarks to stress our design and measure overheads. We show that overheads exist in some pessimistic cases, but have negligible impact in real-world scenarios. All experiments are performed on the same server machine in the Rex-BMC experiments (§7.2).

Setup and teardown. Entering and exiting a Rex program requires Rex-specific operations (Figure 4). Rex's use of a dedicated stack requires saving the stack pointer and setting the new stack and frame pointer to the dedicated stack (and restoring to the saved values after the extension exits). Rex also needs to set up the per-CPU state used by its termination mechanism (§5.7). In total, these operations add eight instructions on the execution path in Rex. To measure the overhead, we implement an empty extension program in both eBPF and Rex and record their execution time (including the program dispatcher). As shown in Table 2, the measured execution time of the empty Rex and eBPF programs only differ in around a nanosecond on average.

Exception handling Rex's safe cleanup for exception handling requires recording allocated resource at runtime (§5.5), which, compared to eBPF, adds overhead. We measure the overhead using a program that acquires and then immediately releases an eBPF spinlock. Since the acquired spinlock needs

Table 2: Time to execute an empty extension program and to acquire and release a spinlock in eBPF and Rex (nanosecond)

Extension	Empty prog runtime	Spinlock runtime
eBPF	42.1 ± 4.1 ns	130.4 ± 20.3 ns
Rex	42.6 ± 5.8 ns	183.1 ± 27.5 ns

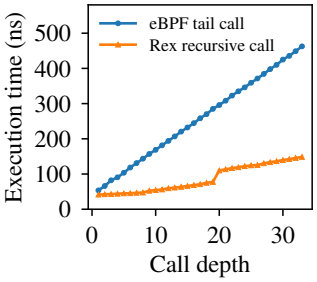


Figure 7: eBPF tail call and Rex recursive call time

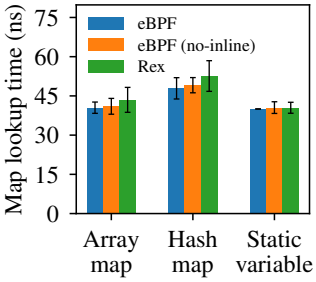


Figure 8: Map lookup time under various setups

to be released upon Rust panics, Rex's cleanup mechanism records it in its per-CPU buffer. Additionally, Rex sets up a per-CPU state flag to indicate execution of a helper function (§5.7). The program is implemented in both eBPF and Rex and the time used to acquire and release the spinlocks are measured. Table 2 shows that the runtime difference between eBPF and Rex is roughly 50 nanoseconds. We argue that such runtime difference is negligible in real world use cases, as exemplified by Rex-BMC (§7.2) that uses fine-grained locking (one spinlock per map entry).

Stack check. Stack checks are added before function calls in Rex extensions that contain indirect or recursive calls (§5.6). We implement recursive extension programs in both eBPF and Rex to measure the overhead. The recursive function calls itself for a controlled number of times. In Rex, we pass the call depth as the argument to the recursive function; since eBPF does not support recursive functions, we use eBPF tail calls to implement the logic—since it is inconvenient to pass arguments to tail-called programs (§3.1), we use a static variable to set the call depth. Figure 7 plots execution time of the recursive programs with call depths from 1 to 33 (eBPF cannot do more than 33 tail calls). Rex is roughly 3x faster than eBPF. The overhead of eBPF is due to its runtime check on tail-call count limit and accessing the static variable, which is a map in eBPF (not a register as in Rex's normal calls).

Map access Map access in Rex is expected to have more overhead than in eBPF. First, Rex implements wrapping code to enforce safety of helper function calls (§6). Moreover, the eBPF JIT compiler inlines the helper function for map lookup at load time as a performance optimization; however, inlining is not available in Rex (no JIT in Rex). We measure map lookup time of Rex, compared with eBPF, with and without inlining, including array maps, hash maps, and static variables. In eBPF, static variables are converted into maps; we use a static Rust atomic variable in Rex, as the counterpart of a static variable map in eBPF. Figure 8 shows the lookup time of different maps in eBPF and Rex, respectively. We find that inlining map lookups in eBPF are $\sim$0.5*ns* faster on array maps and $\sim$1.2*ns* faster on hash maps. An additional slowdown of 2*ns*–4*ns* is present in Rex over non-inlined eBPF, due to the wrapping code. Static variables in eBPF are always accessed via direct load without invoking a helper. Hence, their access latency is almost the same to accessing Rust atomic variables.

8 Discussion

Verification without language-verifier gaps. Rex currently uses language features of Rust to ensure safety of kernel extensions. This approach defers the checking of some safety properties to the runtime (e.g., termination, integer errors). It may be possible to minimize the amount of runtime errors by incorporating Rust-based verification techniques, e.g., ensuring freedom of panics [15, 24, 69, 70, 105]. Certainly, push-button verification techniques that use symbolic execution such as PanicCheck [105] are likely to re-introduce the language-verifier gap. We suspect that using verification techniques that combine proofs and programming [27, 28, 69, 70], such as Verus for Rust may allow Rex to reduce runtime errors *without* the language-verifier gap.

Dynamic memory allocation. eBPF has recently supported dynamic allocation [51] that allows extension programs to request memory from the kernel using allocation kfuncs [46]. Rex currently does not support dynamic memory allocation. We plan to integrate memory allocation [1] of Rust with the eBPF all-context allocator [44], granting Rex dynamic allocation. Dynamic allocation enhances programmability of extension programs and opens up more advanced use cases [52]. It also makes more components of the Rust standard library available, notably the collection and smart pointer types with automatic memory management.

Kernel crate maintenance. The Rex kernel crate inevitably needs to use unsafe Rust, as it directly interacts with kernel functions and variables defined in C. As a principle, unsafe Rust code must not be used for escaping safety checks but only when it is the last resort (mostly for foreign function interface, FFI). This keeps the scope of unsafe Rust at its minimum—the Rex kernel crate only leverages unsafe Rust necessary for FFI interaction and contributes to about 10% (360 lines) of kernel crate code. We explicitly check and ensure the preconditions of unsafe code blocks (e.g., argument requirements of eBPF helper functions) for the soundness of the Rex crate. As unsafe code is isolated from extension programs and managed at a central location by trusted maintainers, we are not particularly concerned about its maintainability.

9 Related Work

Aya-rs [3] provides an infrastructure that allows developers to write eBPF programs in Rust and compile to eBPF bytecode.

Aya is fundamentally different from Rex in that it does not rely on the Rust language for safety properties, but relies on the in-kernel eBPF verifier. Programming in Aya ignores the safety aspect of Rust and only treats it as a frontend language for eBPF—unsafe code blocks that directly use raw pointers are common. Differently, in Rex, one can safely use the high-level language features of Rust exposed by Rex. These features are important to write more intuitive, cleaner, and more optimized code for large, complex kernel extensions.

Improving eBPF. eBPF has evolved from simple use cases like packet filtering [75, 77] into a general-purpose kernel extension language and programming framework that enables many innovative projects [11, 40, 55, 59, 60, 79, 87, 90, 106–108]. Recent work is making active progress to improve the correctness and security of the eBPF infrastructure, including fuzzing and bug finding [61,78,94,95], formal verification [84, 85, 98, 100], sandboxing [65, 72, 73], and integrating with hardware protection mechanisms [74, 104]. eBPF's design, which relies on an in-kernel static verifier for extension safety, inevitably creates the language-verifier gap (§3). In contrast, Rex provides an alternative to develop and maintain large, complex kernel extensions directly with high-level language safety, avoiding the language-verifier gap.

Other frameworks. The idea of building safe OS extensibility using safe languages was proposed by SPIN [34] and revisited by Singularity [62], Tock [71], and a few recent discussions [39, 63, 76]. However, adopting them in practice is challenging as they are based on clean-slate OS designs. Rex develops a practical kernel extension framework for Linux, taking the opportunity of recent support of Rust as a safe language for OS code. It addresses the key challenges of enforcing safe code only, interfacing with unsafe C code, and providing safety guarantees beyond language-based safety.

Prior works such as VINO [89] explore extension safety through Software Fault Isolation (SFI) with unsafe languages. SFI is in principle implemented as runtime checks, which not only causes programming errors to be completely uncaught before deployment but also leads to non-trivial overhead. While recent works [81, 103] have demonstrated low-overhead SFI techniques, we note that Rex represents a different design tradeoff and complements SFI. By leveraging language-based safety, Rex pushes a good set of safety checks into compile time, allowing programming errors to be detected before execution. It strikes a balance between static checks and runtime protection to close the language-verifier gap.

Similar to SFI, hardware-based isolation mechanisms such as Intel MPK [49] offer alternative paths toward extension safety [74, 104]. However, such mechanisms alone usually do not satisfy all the safety requirements of kernel extensions, e.g., Intel MPK does not handle safety of kernel interactions as it cannot ensure a lock acquired will be properly released.

KFlex [52] is a recent kernel extension framework built on top of eBPF. KFlex aims to improve the flexibility of eBPF to let developers express diverse functionality in extensions. It employs an efficient runtime by co-designing it with the existing eBPF verifier: (1) its SFI elides checks already done by the verifier for efficiency, and (2) its termination mechanism uses the verifier to statically compute the kernel resources acquired by the extension. Rex made the same design choice as KFlex to use a lightweight runtime for safety properties that are hard to check statically. Unlike KFlex, which is co-designed with the eBPF verifier, Rex eliminates the verifier to close the language-verifier gap.

BCF [93] is a recent proposal to enhance eBPF's in-kernel verification with help from user space, asking for proof when the verifier fails to reason about certain program properties. The idea echoes proof-carrying code [83] which asks a program to attach a formal proof that its code obeys the safety policy. BCF leverages the eBPF verifier's range analysis and symbolic execution for proof generation but still requires developers to specify safety conditions to aid the generation. Its uses of the verifier still lead to the language-verifier gap.

Rust for OS kernels. Rust has been embraced by modern OSes [23, 101] as practical language which leads to safer code. Recent work shows the promises to build new OS kernels using Rust [36, 39, 71, 82]. We claim no novelty of using Rust as a language. In fact, a safe language alone does not lead to system safety, as exemplified by Rust kernel modules [48]. Rex shows an example of how to build upon language-based safety to enable and enforce safe kernel extension programs.

10 Conclusion

We build Rex, a new kernel extension framework that closes the language-verifier gap. We believe that closing the gap is essential to the programming experience and maintainability of kernel extensions, especially those that embody large, complex programs for advanced features. Rex provides a solution that allows kernel extensions to be developed and maintained in a high-level language, while providing desired safety guarantees as in existing frameworks like eBPF.

Acknowledgement

We appreciate the help from our shepherd Diyu Zhou, as well as the anonymous reviewers, on improving this paper. We thank Zhenyu Huang, Manvik Nanda, Quan Hao Ng, Raj Sahu, Wentao Zhang, and Md Sayeedul Islam for their participation in the project. We also thank Jiyuan Zhang, Di Jin, Hao Lin, James Bottomley, and Darko Marinov for their feedback and discussion. We used Rex to develop Machine Problems (MPs) for CS 423 (Operating System Design) at the University of Illinois Urbana-Champaign in Fall 2022, 2023, and 2024. We thank students in CS 423 for being the beta users of Rex, which provides valuable feedback for us to improve the usability of Rex. This work was funded in part by NSF CNS-1956007, NSF CNS-2236966, and an IBM-Illinois Discovery Accelerator Institute (IIDAI) grant.

References

[1] alloc - Rust. https://doc.rust-lang.org/alloc/index.html.

[2] auto_traits. https://doc.rust-lang.org/beta/unstable-book/language-features/auto-traits.html.

[3] Aya-rs. https://aya-rs.dev/.

[4] Bounds - Rust By Example. https://doc.rust-lang.org/rust-by-example/generics/bounds.html.

[5] BPF and XDP Reference Guide. https://docs.cilium.io/en/latest/bpf/index.html.

[6] Cilium - Cloud Native, eBPF-based Networking, Observability, and Security. https://cilium.io/.

[7] Clippy Lints. https://rust-lang.github.io/rust-clippy/master/index.html.

[8] Codegen Options - The rustc book. https://doc.rust-lang.org/rustc/codegen-options/index.html.

[9] Drop in core::ops - Rust. https://doc.rust-lang.org/core/ops/trait.Drop.html.

[10] eBPF implementation that runs on top of Windows. https://github.com/microsoft/ebpf-for-windows.

[11] Extensible Scheduler Class. https://docs.kernel.org/next/scheduler/sched-ext.html.

[12] Fuzzing. https://rustc-dev-guide.rust-lang.org/fuzzing.html.

[13] Generic parameters. https://doc.rust-lang.org/reference/items/generics.html.

[14] Itanium C++ ABI: Exception Handling ($Revision: 1.22 $). http://itanium-cxx-abi.github.io/cxx-abi/abi-eh.html.

[15] Kani Rust Verifier. https://github.com/model-checking/kani.

[16] Katran - A high performance layer 4 load balancer. https://github.com/facebookincubator/katran.

[17] Lints - The rustc book. https://doc.rust-lang.org/rustc/lints/index.html.

[18] Monomorphization. https://rustc-dev-guide.rust-lang.org/backend/monomorph.html.

[19] negative_impls. https://doc.rust-lang.org/beta/unstable-book/language-features/negative-impls.html.

[20] no_std - The Embedded Rust Book. https://docs.rust-embedded.org/book/intro/no-std.html.

[21] RAII - Rust By Example. https://doc.rust-lang.org/rust-by-example/scope/raii.html.

[22] rust-bindgen. https://github.com/rust-lang/rust-bindgen.

[23] Rust for Linux. https://rust-for-linux.com/.

[24] Rust Verification Tools. https://project-oak.github.io/rust-verification-tools/.

[25] Security Hardening: Use of eBPF by unprivileged users has been disabled by default. https://www.suse.com/support/kb/doc/?id=000020545.

[26] std - Rust. https://doc.rust-lang.org/std/index.html.

[27] The Coq Proof Assistant. https://coq.inria.fr/.

[28] The Dafny Programming and Verification Language. https://dafny.org/.

[29] The LLVM Compiler Infrastructure. https://llvm.org/.

[30] The rustc book. https://doc.rust-lang.org/stable/rustc/.

[31] Trait objects. https://doc.rust-lang.org/reference/types/trait-object.html.

[32] What is RCU? – "Read, Copy, Update". https://docs.kernel.org/RCU/whatisRCU.html.

[33] BALASUBRAMANIAN, A., BARANOWSKI, M. S., BURTSEV, A., PANDA, A., RAKAMARIĆ, Z., AND RYZHYK, L. System Programming in Rust: Beyond Safety. In *Proceedings of the 16th ACM Workshop on Hot Topics in Operating Systems (HotOS'17)* (May 2017).

[34] BERSHAD, B. N., SAVAGE, S., PARDYAK, P., SIRER, E. G., FIUCZYNSKI, M. E., BECKER, D., CHAMBERS, C., AND EGGERS, S. Extensibility, Safety and Performance in the SPIN Operating System. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP'95)* (Dec. 1995).

[35] BHAT, S., AND SHACHAM, H. Formal Verification of the Linux Kernel eBPF Verifier Range Analysis. https://sanjit-bhat.github.io/assets/pdf/ebpf-verifier-range-analysis22.pdf. (May 2022).

[36] BOOS, K., LIYANAGE, N., IJAZ, R., AND ZHONG, L. Theseus: an Experiment in Operating System Structure and State Management. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI'20)* (Nov. 2020).

[37] BORKMANN, D. bpf: Fix verifier issue in fib_redirect. https://github.com/cilium/cilium/commit/efb5d6509fea263bd6d36998f8e524d9942b8a79. (Mar. 2023).

[38] BORKMANN, D. bpf: fix verifier's ctx port access in post bind hooks. https://github.com/cilium/cilium/commit/394e72478a8d120dab0bff2c41db77695877ce57. (Mar. 2023).

[39] BURTSEV, A., NARAYANAN, V., HUANG, Y., HUANG, K., TAN, G., AND JAEGER, T. Evolving Operating System Kernels Towards Secure Kernel-Driver Interfaces. In *Proceedings of the 19th ACM Workshop on Hot Topics in Operating Systems (HotOS'23)* (June 2023).

[40] CAO, X., PATEL, S., LIM, S. Y., HAN, X., AND PASQUIER, T. FetchBPF: Customizable Prefetching Policies in Linux with eBPF. In *Proceedings of the 2024 USENIX Annual Technical Conference (USENIX ATC'24)* (July 2024).

[41] CHAIGNON, P. bpf: Avoid 32bit assignment of packet pointer. https://github.com/cilium/cilium/commit/847014aa62f94e5a53178670cad1eacea455b227. (May 2023).

[42] CHAIKEN, A. IRQs: the Hard, the Soft, the Threaded and the Preemptible. In *Embedded Linux Conference Europe (ELCE'16)*. https://events.static.linuxfound.org/sites/events/files/slides/Chaiken_ELCE2016.pdf. (Oct. 2016).

[43] CHINTAMANENI, S., SOMARAJU, S. R., AND WILLIAMS, D. Unsafe kernel extension composition via BPF program nesting. In *Proceedings of the 2nd ACM SIGCOMM 2024 Workshop on eBPF and Kernel Extensions (eBPF'24)* (Aug. 2024).

[44] CORBET, J. A BPF-specific memory allocator. https://lwn.net/Articles/899274/. (June 2022).

[45] CORBET, J. A first look at Rust in the 6.1 kernel. https://lwn.net/Articles/910762/. (Oct. 2022).

[46] CORBET, J. Calling kernel functions from BPF. https://lwn.net/Articles/856005/. (May 2021).

[47] CORBET, J. Reconsidering unprivileged BPF. https://lwn.net/Articles/796328/. (Aug. 2019).

[48] CORBET, J. Rust-for-Linux developer Wedson Almeida Filho drops out. https://lwn.net/Articles/987635/. (Aug. 2024).

[49] CORBET, J. Memory protection keys. https://lwn.net/Articles/643797/, May 2015.

[50] DUBERSTEIN, T. bpf: Remove builtin global functions. https://github.com/aya-rs/aya/pull/698. (July 2023).

[51] DWIVEDI, K. K. bpf: Introduce bpf_obj_new. https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?id=958cf2e273f0929c66169e0788031310e8118722. (Nov. 2022).

[52] DWIVEDI, K. K., IYER, R., AND KASHYAP, S. Fast, Flexible, and Practical Kernel Extensions. In *Proceedings of the 30th ACM Symposium on Operating Systems Principles (SOSP'24)* (Nov. 2024).

[53] EDGE, J. BPF and security. https://lwn.net/Articles/946389/. (Oct. 2023).

[54] FAHNDRICH, M., AIKEN, M., HAWBLITZEL, C., HODSON, O., HUNT, G., LARUS, J., AND LEVI, S. Language Support for Fast and Reliable Message-based Communication in Singularity OS. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys'06)* (Apr. 2006).

[55] GHIGOFF, Y., SOPENA, J., LAZRI, K., BLIN, A., AND MULLER, G. BMC: Accelerating Memcached using Safe In-kernel Caching and Pre-stack Processing. In *Proceedings of the 18th USENIX Symposium on Networked Systems Design and Implementation (NSDI'21)* (Apr. 2021).

[56] GLEIXNER, T., AND MOLNAR, I. hrtimers - subsystem for high-resolution kernel timers. https://docs.kernel.org/timers/hrtimers.html.

[57] GRAF, T. bpf: Workaround for verifier bug in proxy hairpin code. https://github.com/cilium/cilium/commit/e38a92115620125b19c8761f35f6709e71c34511. (May 2019).

[58] GUPTA, P. bpf: Disallow unprivileged bpf by default. https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?id=8a03e56b253e9691c90bc52ca199323d71b96204. (Oct. 2021).

[59] HØILAND-JØRGENSEN, T., BROUER, J. D., BORKMANN, D., FASTABEND, J., HERBERT, T., AHERN, D., AND MILLER, D. The eXpress data path: fast programmable packet processing in the operating system kernel. In *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies (CoNEXT'18)* (Dec. 2018).

[60] HUMPHRIES, J. T., NATU, N., CHAUGULE, A., WEISSE, O., RHODEN, B., DON, J., RIZZO, L., ROMBAKH, O., TURNER, P., AND KOZYRAKIS, C. ghOSt: Fast & Flexible User-Space Delegation of Linux Scheduling. In *Proceedings of the 28th ACM Symposium on Operating Systems Principles (SOSP'21)* (Oct. 2021).

[61] HUNG, H.-W., AND AMIRI SANI, A. BRF: Fuzzing the eBPF Runtime. In *Proceedings of the 2024 ACM International Conference on the Foundations of Software Engineering (FSE'24)* (July 2024).

[62] HUNT, G., LARUS, J., ABADI, M., AIKEN, M., BARHAM, P., FAHNDRICH, M., HAWBLITZEL, C., HODSON, O., LEVI, S., MURPHY, N., STEENSGAARD, B., TARDITI, D., WOBBER, T., AND ZILL, B. An Overview of the Singularity Project. Tech. Rep. MSR-TR-2005-135, Microsoft Research. (Oct. 2005).

[63] JIA, J., SAHU, R., OSWALD, A., WILLIAMS, D., LE, M. V., AND XU, T. Kernel extension verification is untenable. In *Proceedings of the 19th ACM Workshop on Hot Topics in Operating Systems (HotOS'23)* (June 2023).

[64] JIA, J., ZHU, Y., WILLIAMS, D., ARCANGELI, A., CANELLA, C., FRANKE, H., FELDMAN-FITZTHUM, T., SKARLATOS, D., GRUSS, D., AND XU, T. Programmable System Call Security with eBPF. *arXiv:2302.10366* (Feb. 2023).

[65] JIN, D., GAIDIS, A. J., AND KEMERLIS, V. P. BeeBox: Hardening BPF against Transient Execution Attacks. In *Proceedings of the 33rd USENIX Security Symposium (USENIX Security'24)* (Aug. 2024).

[66] JUNG, R., DANG, H.-H., KANG, J., AND DREYER, D. Stacked borrows: an aliasing model for Rust. In *Proceedings of the 47th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL'20)* (Jan. 2020).

[67] JUNG, R., JOURDAN, J.-H., KREBBERS, R., AND DREYER, D. RustBelt: securing the foundations of the Rust programming language. In *Proceedings of the 45th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL'18)* (Jan. 2018).

[68] KIM, T. H., RUDO, D., ZHAO, K., ZHAO, Z. N., AND SKARLATOS, D. Perspective: A Principled Framework for Pliable

and Secure Speculation in Operating Systems. In *Proceedings of the ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA'24)* (June 2024).

[69] LATTUADA, A., HANCE, T., BOSAMIYA, J., BRUN, M., CHO, C., LEBLANC, H., SRINIVASAN, P., ACHERMANN, R., CHAJED, T., HAWBLITZEL, C., HOWELL, J., LORCH, J. R., PADON, O., AND PARNO, B. Verus: A Practical Foundation for Systems Verification. In *Proceedings of the 30th ACM Symposium on Operating Systems Principles (SOSP'24)* (Sept. 2024).

[70] LATTUADA, A., HANCE, T., CHO, C., BRUN, M., SUBASINGHE, I., ZHOU, Y., HOWELL, J., PARNO, B., AND HAWBLITZEL, C. Verus: Verifying Rust Programs using Linear Ghost Types. In *Proceedings of the 2023 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'23)* (Apr. 2023).

[71] LEVY, A., CAMPBELL, B., GHENA, B., GIFFIN, D. B., PANNUTO, P., DUTTA, P., AND LEVIS, P. Multiprogramming a 64kB Computer Safely and Efficiently. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP'17)* (Oct. 2017).

[72] LIM, S. Y., HAN, X., AND PASQUIER, T. Unleashing Unprivileged eBPF Potential with Dynamic Sandboxing. In *Proceedings of the 1st ACM SIGCOMM 2023 Workshop on eBPF and Kernel Extensions (eBPF'23)* (Sept. 2023).

[73] LIM, S. Y., PRASAD, T., HAN, X., AND PASQUIER, T. SafeBPF: Hardware-assisted Defense-in-depth for eBPF Kernel Extensions. *arXiv:2409.07508* (Sept. 2024).

[74] LU, H., WANG, S., WU, Y., HE, W., AND ZHANG, F. MOAT: Towards Safe BPF Kernel Extension. In *Proceedings of the 33rd USENIX Security Symposium (USENIX Security'24)* (Aug. 2024).

[75] MCCANNE, S., AND JACOBSON, V. The BSD Packet Filter: A New Architecture for User-level Packet Capture. In *Proceedings of the 1993 Winter USENIX Conference* (Jan. 1993).

[76] MILLER, S., ZHANG, K., ZHUO, D., XU, S., KRISHNAMURTHY, A., AND ANDERSON, T. Practical Safe Linux Kernel Extensibility. In *Proceedings of the 17th ACM Workshop on Hot Topics in Operating Systems (HotOS'21)* (May 2021).

[77] MOGUL, J. C., RASHID, R. F., AND ACCETT, M. J. The Packet Filter: An Efficient Mechanism for User-level Network Code. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles (SOSP'87)* (Nov. 1987).

[78] MOHAMED, M. H. N., WANG, X., AND RAVINDRAN, B. Understanding the Security of Linux eBPF Subsystem. In *Proceedings of the 14th ACM SIGOPS Asia-Pacific Workshop on Systems (APSys'23)* (Aug. 2023).

[79] MORES, K., PSOMADAKIS, S., AND GOUMAS, G. eBPF-mm: Userspace-guided memory management in Linux with eBPF. *arXiv:2409.11220* (Sept. 2024).

[80] MURRAY, A. Unprivileged eBPF disabled by default for Ubuntu 20.04 LTS, 18.04 LTS, 16.04 ESM. https://discourse.ubuntu.com/t/unprivileged-ebpf-disabled-by-default-for-ubuntu-20-04-lts-18-04-lts-16-04-esm/27047. (Mar. 2022).

[81] NARAYAN, S., GARFINKEL, T., JOHNSON, E., YEDIDIA, Z., WANG, Y., BROWN, A., VAHLDIEK-OBERWAGNER, A., LEMAY, M., HUANG, W., WANG, X., SUN, M., TULLSEN, D., AND STEFAN, D. Segue & ColorGuard: Optimizing SFI Performance and Scalability on Modern Architectures. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'25)* (Mar. 2025).

[82] NARAYANAN, V., HUANG, T., DETWEILER, D., APPEL, D., LI, Z., ZELLWEGER, G., AND BURTSEV, A. RedLeaf: Isolation and Communication in a Safe Operating System. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI'20)* (Nov. 2020).

[83] NECULA, G. C., AND LEE, P. Safe Kernel Extensions Without Run-Time Checking. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation (OSDI'96)* (Oct. 1996).

[84] NELSON, L., GEFFEN, J. V., TORLAK, E., AND WANG, X. Specification and verification in the field: Applying formal methods to BPF just-in-time compilers in the Linux kernel. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI'20)* (Nov. 2020).

[85] NELSON, L., WANG, X., AND TORLAK, E. A proof-carrying approach to building correct and flexible BPF verifiers. In *Linux Plumbers Conference* (Sept. 2021).

[86] RAJAHALME, J. datapath: Use inline function to keep policy and l4policy checks separate. https://github.com/cilium/cilium/commit/142c0f7128c7fac22eb18b2c21a56433f19a5ef8. (Apr. 2023).

[87] RHODEN, B. eBPF Kernel Scheduling with Ghost. In *Linux Plumbers Conference (LPC'22)*. https://lpc.events/event/16/contributions/1365/. (Sept. 2022).

[88] SAHU, R., AND WILLIAMS, D. When BPF programs need to die: exploring the design space for early BPF termination. In *Linux Plumbers Conference (LPC'23)*. https://lpc.events/event/17/contributions/1610/. (Nov. 2023).

[89] SELTZER, M. I., ENDO, Y., SMALL, C., AND SMITH, K. A. Dealing With Disaster: Surviving Misbehaved Kernel Extensions. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation (OSDI'96)* (Oct. 1996).

[90] SKARLATOS, D., AND ZHAO, K. Towards Programmable Memory Management with eBPF. In *Linux Plumbers Conference (LPC'24)*. https://lpc.events/event/18/contributions/1932/. (Sept. 2024).

[91] SOMARAJU, S. R., CHINTAMANENI, S., AND WILLIAMS, D. Overflowing the kernel stack with BPF. In *Linux Plumbers Conference (LPC'23)*. https://lpc.events/event/17/contributions/1595/. (Nov. 2023).

[92] STAROVOITOV, A. bpf: verifier (add docs). https://git.kernel.org/pub/scm/linux/

kernel/git/stable/linux.git/commit/?id=51580e798cb61b0fc63fa3aa6c5c975375aa0550. (Sept. 2014).

[93] SUN, H., AND SU, Z. Lazy Abstraction Refinement with Proof. In *Linux Plumbers Conference (LPC'24)*. https://lpc.events/event/18/contributions/1939/. (Sept. 2024).

[94] SUN, H., AND SU, Z. Validating the eBPF Verifier via State Embedding. In *Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI'24)* (July 2024).

[95] SUN, H., XU, Y., LIU, J., SHEN, Y., GUAN, N., AND JIANG, Y. Finding Correctness Bugs in eBPF Verifier with Structured and Sanitized Program. In *Proceedings of the 19th ACM European Conference on Computer Systems (EuroSys'24)* (Apr. 2024).

[96] VERNET, D. More flexible memory access for BPF programs. https://lwn.net/Articles/910873/. (Oct. 2022).

[97] VISHWANATHAN, H., SHACHNAI, M., CHAIGNON, P., NAGARAKATTE, S., AND NARAYANA, S. Agni: Fast Formal Verification of the Verifier's Range Analysis. In *Linux Plumbers Conference (LPC'24)*. https://lpc.events/event/18/contributions/1937/. (Sept. 2024).

[98] VISHWANATHAN, H., SHACHNAI, M., NARAYANA, S., AND NAGARAKATTE, S. Verifying the Verifier: eBPF Range Analysis Verification. In *Proceedings of the 35th International Conference on Computer Aided Verification (CAV'23)* (July 2023).

[99] WAGNER, J., KUZNETSOV, V., AND CANDEA, G. - OVERIFY: Optimizing Programs for Fast Verification. In *Proceedings of the 14th USENIX Workshop on Hot Topics in Operating Systems (HotOS'13)* (May 2013).

[100] WANG, X., LAZAR, D., ZELDOVICH, N., CHLIPALA, A., AND TATLOCK., Z. Jitk: A Trustworthy In-Kernel Interpreter Infrastructure. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI'14)* (Oct. 2014).

[101] WESTON, D. The journey towards default security. In *BlueHat IL 2023*. https://www.youtube.com/watch?v=8T6ClX-y2AE. (Mar. 2023).

[102] YANG, Z., LU, Y., LIAO, X., CHEN, Y., LI, J., HE, S., AND SHU, J. λ-IO: A Unified IO Stack for Computational Storage. In *Proceedings of the 21st USENIX Conference on File and Storage Technologies (FAST'23)* (Feb. 2023).

[103] YEDIDIA, Z. Lightweight Fault Isolation: Practical, Efficient, and Secure Software Sandboxing. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'24)* (Apr. 2024).

[104] ZHANG, P., WU, C., MENG, X., ZHANG, Y., PENG, M., ZHANG, S., HU, B., XIE, M., LAI, Y., KANG, Y., AND WANG, Z. HIVE: A Hardware-assisted Isolated Execution Environment for eBPF on AArch64. In *Proceedings of the 33rd USENIX Security Symposium (USENIX Security'24)* (Aug. 2024).

[105] ZHANG, Y., LI, P., DING, Y., WANG, L., MENG, N., AND WILLIAMS, D. Broadly Enabling KLEE to Effortlessly Find Unrecoverable Errors. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP'24)* (Apr. 2024).

[106] ZHONG, Y., LI, H., WU, Y. J., ZARKADAS, I., TAO, J., MESTERHAZY, E., MAKRIS, M., YANG, J., TAI, A., STUTSMAN, R., AND CIDON, A. XRP: In-Kernel Storage Functions with eBPF. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI'22)* (July 2022).

[107] ZHOU, Y., WANG, Z., DHARANIPRAGADA, S., AND YU, M. Electrode: Accelerating Distributed Protocols with eBPF. In *Proceedings of the 20th USENIX Symposium on Networked Systems Design and Implementation (NSDI'23)* (Apr. 2023).

[108] ZHOU, Y., XIANG, X., KILEY, M., DHARANIPRAGADA, S., AND YU, M. DINT: Fast In-Kernel Distributed Transactions with eBPF. In *Proceedings of the 21st USENIX Symposium on Networked Systems Design and Implementation (NSDI'24)* (Apr. 2024).

A Appendix

We discuss more details of the example mentioned in §3.1 where developers have to change their code to assist the eBPF verifier. Figure 9a shows a code example from Cilium [86], which originally used a `goto` statement to combine the code path of `policy` and `l4policy` to avoid duplicated code. However, the combined code, which assigns `l4policy` to `policy`, later causes the verifier to incorrectly believe that `policy`, which is a pointer variable, is instead a scalar and reject the program. As a workaround, developers had to refactor the policy check code into an inlined function to separate the code path to pass the verifier.

Developers also have to teach the verifier by providing additional information. Figure 9b shows an example in Cilium [37] where the verifier lost track of `nh_params.nh_family`, a 32-bit scalar value on the stack, and mistakenly treated it as a spilled 64-bit pointer, leading to an invalid size error on the load. As a workaround, developers passed `fib_params->l.family` directly instead of going through `nh_params.nh_family` to let the verifier know the scalar value.

```
1    if (likely(l4policy && !l4policy->wildcard_dport)) {
2      *match_type = POLICY_MATCH_L4_ONLY;
3  -   policy = l4policy;
4  -   goto policy_check_entry;
5  +   return __account_and_check(ctx, l4policy, ...);
6    }
7
8    if (likely(policy && !policy->wildcard_protocol)) {
9      *match_type = POLICY_MATCH_L3_PROTO;
10 -   goto policy_check_entry;
11 +   return __account_and_check(ctx, policy, ...);
12   }
```

(a) The assignment of `policy` to `l4policy` (L3) and the `goto` (L4) causes the verifier to misinterpret the `policy` pointer as a scalar, requiring the policy check logic to be refactored into its own function.

```
1    #ifndef ENABLE_SKIP_FIB
2    ...
3    if (likely(ret == BPF_FIB_LKUP_RET_NO_NEIGH)) {
4      nh_params.nh_family = fib_params->l.family;
5      ...
6    } else {
7      return DROP_NO_FIB;
8    } ...
9    skip_oif:
10   #else
11   *oif = DIRECT_ROUTING_DEV_IFINDEX;
12 - nh_params.nh_family = fib_params->l.family;
13   #endif /* ENABLE_SKIP_FIB */
14   ...
15 - dmac = nh_params.nh_family == AF_INET ? ...;
16 + dmac = fib_params->l.family == AF_INET ? ...;
```

(b) The verifier mistakenly treats the 32-bit on-stack scalar `nh_params.nh_family` as a pointer. Developers are forced to refactor the code and avoid its use (L12 and L15).

Figure 9: Examples that developers had to work around the language-verifier gap by refactoring already safe extensions

Barre: Empowering Simplified and Versatile Programmable Congestion Control in High-Speed AI Clusters

Yajuan Peng *[1], Haoran Wei *[2], Xiaolong Zhong[2], Junkai Huang[2], Haohan Xu[2], Zicheng Wang[2], Yang Bai[2], Zhuo Jiang[2], Jianxi Ye[2], Xiaoliang Wang , Xiaoming Fu †[1], and Huichen Dai † [2]

[1]*Shanghai Key Laboratory for Intelligence Information Processing, Fudan University, China*
[2]*ByteDance*

Abstract

Network interface cards (NICs) and switches have entered the 400 Gbps era. RoCEv2 networks face significant challenges in congestion management, particularly under high-throughput workloads. While advanced congestion control algorithms have been proposed, their deployment in large-scale data centers remains hindered by complex parameter tuning and dependency on sophisticated hardware features. In this paper, we present Barre, a simple yet highly effective congestion control scheme designed for modern AI/HPC clusters operating at 400 Gbps. By leveraging commodity hardware and standard network functionalities, Barre achieves near-optimal performance in fairness, congestion responsiveness, and scalability with minimal overhead. Deployed in our 400 Gbps RoCE cluster for over a year and supporting up to 10,000 GPUs, Barre improves AI training task throughput by an average of 9.6%. Furthermore, we demonstrate that Barre's core principles can be seamlessly applied to enhance DCQCN, a widely deployed congestion control algorithm, underscoring its practicality and versatility.

1 Introduction

The leading tech companies are building new data centers to support current and next-generation high-performance computing and AI models. The networks in these data centers play a crucial role in interconnecting thousands of GPUs, providing the infrastructure to train large models with hundreds of billions of parameters, e.g. LLAMA 3.1 405B. These cutting-edge large-scale network infrastructures have utilized high-performance switches with 51.2 Tbps of throughput capacity and network interface cards (NICs) with 400Gbps interconnect capabilities [14, 27, 44].

Despite the widespread adoption of high-bandwidth networking infrastructure, the Machine FLOP Utilization (MFU) in large-scale LLM training remains suboptimal. This limitation arises primarily due to network congestion during the transmission of vast amounts of data across multiple compute nodes, which causes long delays for the parameter updates and gradient exchanges required in the training process. In current AI clusters, RoCEv2 (RDMA over Converged Ethernet v2) is widely applied for inter-node communication transport. Data Center Quantized Congestion Notification (DCQCN) [60] is the de facto congestion control algorithm for RDMA NICs. However, DCQCN faces significant challenges in achieving low-latency performance under dynamic traffic patterns due to its intricate parameter tuning requirements [6, 35, 51]. A key limitation of DCQCN is its sluggish congestion response, stemming from prolonged control loops, which perpetuates dependence on Priority Flow Control (PFC) for backpressure mitigation. In high-bandwidth networks, relaxed DCQCN configurations exacerbate congestion, increasing PFC pauses by 2–3 times while aggressive tuning yields marginal improvements in flow completion time (FCT) at the cost of degraded throughput under collective communication patterns [14].

Although many advanced congestion control (CC) algorithms have been proposed, their adoption in production AI clusters remains limited across industry deployments [10]. For the hardware dependency, HPCC [31] mandates In-band Network Telemetry (INT) support in NICs and switches, creating deployment barriers in heterogeneous environments. Swift [28] requires computationally intensive operations (e.g., square root calculations) for congestion signal processing, introducing non-trivial overhead. Poseidon [52] also identified the performance issues when deploying these algorithms. For example, HPCC may still react to congestion at any hop with high inflight bytes, even if the flow itself is not a major contributor to the congestion. For Swift, the estimation of end-to-end latency is easily affected by link length. Notably, though numerous CC algorithms have been proposed [1, 2, 4, 13, 16, 23, 34, 48, 57], these algorithms need more effort to enhance the compatibility with commercial off-the-shelf hardware environments of 400Gbps and beyond.

In this paper, we investigate the new performance require-

*Equal contribution.
†Corresponding author: Huichen Dai, email: daihuichen@bytedance.com and Xiaoming Fu, email: fu@ieee.org

ments and challenges for congestion control introduced by emerging AI clusters. Then we characterize the support for customized CC algorithms provided by the state-of-the-art AI network infrastructures, along with their inherent limitations. Instead of introducing a new congestion control algorithm, we aim to propose a new framework called Barre, which guides the design of congestion control that is simple and flexible to support the continuous rapid growth of network capacity (400Gbps and 800 Gbps) and the fast-evolving traffic patterns in new models. Barre fully leverages the capabilities provided by modern NICs, addressing the challenges posed by their limitations to achieve effective integration of hardware, software, and algorithms.

To meet the demand for designing customized protocols in cloud operators, a new trend has emerged that introduces programmability into the hardware. The emergence of programmable smartNICs has enabled new paradigms in congestion control for AI-optimized networks. NICs offer Programmable Congestion Control (PCC) capabilities [45], providing hardware-accelerated event APIs that enable customized congestion management solutions for mission-critical AI workloads. Building on these advancements, Barre implements a rate-based congestion control that adheres to the AIMD principles, utilizes the NIC's hardware-accelerated 1μs CNP feedback and dynamically adjust its rate update with built-in RTT probing: (1) *Fast Increase* ensures optimal bandwidth utilization while sustaining low queue length of switches both in mice flows and elephant flows; (2) *Dual-lock* addresses the issue of delayed congestion signal feedback in congested scenarios, ensuring timely congestion responses; (3) *Inflight Monitor* serves as a defensive mechanism against extreme congestion, preemptively reducing the sending rate based on inflight byte control, akin to window-based control. Additionally, we propose RTT-based enhancement to the implementation, which obtained lower latency and improvement of average throughput in end-to-end testing. This tight hardware-software co-design enables precise congestion response while maintaining network stability.

We evaluate Barre on a 256-GPU cluster equipped with 400 Gbps Bluefield-3 SuperNICs. Experimental results show that under NCCL AlltoAll test, Barre reduces the queue length in switches by 16.45% on average (up to 21.79%), and improves the overall throughput of AI training task at an average of 9.6%. Under large-scale test, compared to DCQCN, Barre reduced latency by an average of 55.89%, and improved bandwidth utilization by 15%. Compared to InfiniBand (IB), Barre is able to achieve almost the same latency and throughput. Additionally, we apply insights from Barre to optimize DCQCN, resulting in an enhanced variant that achieves faster convergence while improving fairness and stability under Incast traffic scenarios. Barre has been deployed in our production environment for over 10K GPUs equipped with 400G BlueFiled-3 SuperNICs, with communication across four levels of switches (S3). Over one year of monitoring, Barre achieves maximal throughput with no PFC triggered, and achieves comparable performance with IB for training tasks.

In summary, we make the following contributions:

- We re-examine the performance requirements and challenges that emerging workloads and AI network infrastructures pose for CC algorithms, characterize the merits and limitations of various CC design choices within the context of current NICs and switches, and provide design guidance to realize precise congestion control and fully leverage hardware capacity in modern high-speed AI clusters.
- By leveraging the programmable congestion control capabilities and CC event interfaces provided by BlueField-3 SuperNIC, we propose a CC scheme named Barre. Bare is an event-based algorithm that primarily relies on TX Event, CNP events, and RTT events of the NIC, rather than using ACK events and Timer events. This approach results in low overhead and is more suitable for software and hardware processing in high-speed AI clusters.
- We evaluated Barre on a 256-GPU 400 Gbps cluster with NCCL AlltoAll test, and demonstrated that Barre achieves performance comparable to InfiniBand while utilizing commercially available RoCE NICs (RNICs). Its innovative technique improves flexibility, compatibility, and performance. Inspired by the idea we can guide and enhance the performance of existing congestion controls, e.g. DCQCN, with low deployment complexity.

2 Background

2.1 Dilemma of Advanced CC Algorithms

Congestion control (CC) is one of the most critical techniques for network performance and fair bandwidth allocations. In our investigation, we have observed that despite the proliferation of advanced congestion control algorithms, HPC and AI data centers continue to predominantly employ DCQCN [15, 61], a protocol introduced over a decade ago. The design of DCQCN is grounded in the hardware capabilities and network characteristics prevalent around 2015, rendering it particularly challenging and inefficient to fine-tune for today's high-speed AI training clusters [15, 59]. This leads to the question, *why advanced CC algorithms are not widely adopted in modern data centers?*

Current state-of-the-art CC algorithms generally achieve near-maximal throughput on bottleneck links during congestion [28,31,33,52]. These algorithms employ different congestion signals and leverage diverse process mechanisms, which results in a great disparity in the cost of algorithm implementation (§7). Nonetheless, the majority of CC algorithms require functionalities that are either incompatible with commercial hardware or impose a high computational complexity that exceeds the capability of commercial NICs. This mismatch brings significant challenges for practical deployment and

Table 1: Collective Communication in Parallel Modes

Collectives	Parallel Patterns	Traffic pattern	Network congestion	Incast Scale	Key Requirement
AllReduce	Data Parallel	Tree or Ring	1 to 1, or many to 1	<20	Full bisection bandwidth
AllGather		Ring	1 to 1	=1	Bandwidth
AlltoAll	Expert Parallel	Fullmesh	N to N, N is very large	>500	
SendRecv	Pipeline Parallel	Point to Point	1 to 1, or many to 1	<20	Strict latency

scalability of these advanced CC mechanisms in industrial network environments. Taking HPCC [31] as an example, its deployment necessitates support for In-band Network Telemetry (INT) across all NICs and switches, a capability that remains generally unmet by most commercial NICs and switches. Furthermore, the computationally intensive operations, including square root calculations and complex multiplicative/division operations [28], can significantly prolong congestion signal processing latency and consequently reduce the maximum number of concurrent flows supportable by the congestion control mechanism under fixed hardware constraints.

2.2 Demand for Effective Congestion Control in Modern HPC/AI Clusters

Large-scale distributed training such as generative AI jobs introduces multi-dimension and stringent performance requirements to network infrastructure that are different from general cloud computing [43]. Table 1 illustrates the communication patterns associated with various parallelization methods in LLM training. The parameter configurations for flows to achieve fast convergence differ significantly across different collectives in practice. For example, mice flows such as AllReduce involve small-scale incast, which demand a more aggressive increase in per-flow rate to reach full bandwidth with minimal convergence time. In contrast, elephant flows such as AlltoAll generate high-concurrency, uniformly distributed traffic, require more conservative rate adaptation to prevent network oscillations and transient congestion caused by synchronized rate increases across multiple flows. Diverse traffic patterns and performance requirements pose challenges to the design of CC algorithms with comprehensive coverage. Current approaches struggle to establish universally optimal parameter settings, and transitions between traffic patterns frequently trigger severe congestion events due to suboptimal adaptation dynamics.

In large-scale HPC/AI network, e.g. Meta's statistical analysis of collective communication data from approximately 30K training tasks, AlltoAll communication has dominated total collectives, comprising approximately 60% of total collective traffic volume [14]. Its full-mesh traffic mode can cause ephemeral buffer buildup and severe congestion even with fine-grained flow management. The growing adoption of Mixture-of-Experts (MoE) architectures in large language model development has brought renewed attention to the network congestion challenges posed by AlltoAll collective operations [32]. These communication patterns create intense all-to-all traffic that stresses modern data center networks. In contrast, optimized AllReduce implementations employing multi-rail topologies and Ring algorithms can mitigate the risk of congestion [27], highlighting the importance of communication pattern-aware network optimization.

Consequently, an effective congestion control design must fully leverage the capabilities of modern commodity NICs and switches while satisfying stringent performance requirements in high-speed networks. The architecture should emphasize three key principles: (1) algorithmic simplicity to ensure high performance, (2) deployment flexibility for seamless integration, and (3) implementation practicality to maintain compatibility with existing hardware infrastructure. This balanced approach enables optimal performance without requiring costly hardware upgrades or complex configuration.

3 Motivation

3.1 Design Rationale of Congestion Control in High-speed HPC/AI Cluster

The design of algorithms must be compatible with available hardware, necessitating careful consideration of the costs and efficiency associated with exploring various combinations of hardware, software, and algorithmic approaches.

– Sara Hooker in the hardware lottery [21]

Modern HPC/AI data centers employ cutting-edge network infrastructures featuring 400 Gbps NIC ports and approach to emerging 800 Gbps transmission rate. In large-scale LLM training clusters, the network infrastructure alone constitutes over 20% of total hardware costs (excluding NVSwitch) [7]. On the other hand, the rapid evolution of LLM training has posed significant challenges for the deployment of AI infrastructure, particularly in scaling to clusters comprising tens of thousands of GPUs. With both high-speed network infrastructure and large-scale inter-connection GPUs, the configuration of parameters for CC algorithms has become increasingly challenging. Relaxed thresholds are more prone to trigger Priority-based Flow Control (PFC), whereas more aggressive thresholds can substantially reduce overall throughput, leading to performance degradation [14].

In summary, there is a need for practical CC algorithms

designed for advanced high-speed AI cluster networks, that not only fulfill emerging performance requirements but also align with and leverage the capabilities of the hardware infrastructure. We observe that the new CC algorithms must satisfy the following fundamental requirements:

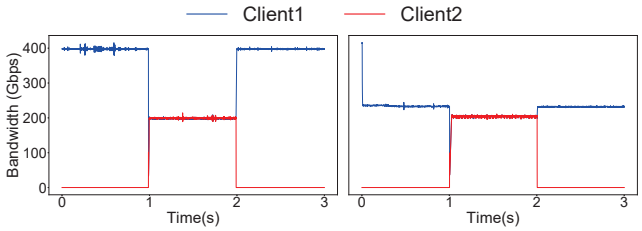
- **Broad Applicability.** CC algorithms should be able to adapt rapidly to traffic bursts while keeping both high bandwidth utilization and low switch buffer occupancy across varying traffic patterns.
- **Simplified Logic.** The algorithm should feature a minimal set of parameters and straightforward threshold selection, avoiding difficult trade-off decisions between strict congestion control and high resource utilization.
- **Implementation Practicality.** The components of the algorithm, e.g. congestion signal, should be widely supported by commercial hardware to avoid customized modifications to NICs and switches. The algorithm should also impose low computational and storage demands for maintaining intermediate congestion handling contexts to achieve fast response.

3.2 Re-examine the CC Design

The evolution of CC algorithms has yielded diverse design paradigms. Nevertheless, our studies have identified incompatibilities between certain approaches and the operational demands of high-speed RoCE networks, presenting significant deployment obstacles. With contemporary NICs delivering 400Gbps inter-GPU server connectivity, CC algorithm design must holistically incorporate both specialized hardware attributes and associated cost constraints. Therefore, this section presents a comprehensive analysis of AI-optimized network hardware characteristics. We re-examine: (1) hardware-informed congestion signal selection criteria, (2) rate adaptation mechanisms optimized for high-bandwidth environments, and (3) co-design methodologies that exploit hardware capabilities while mitigating intrinsic limitations.

Congestion Signals. CC algorithms are based on different congestion signals to detect and feedback congestion information in the network. The widely used signals involves:

• **ECN** (Explicit Congestion Notification) [3] is embedded in the IP header of packets widely supported by unmodified switches to inform endpoints about congestion without dropping packets [3, 16, 60]. ECN provides early feedback by monitoring buffer levels or queue length without awaiting packet loss. However, in the case of multiple queues, a single bit ECN signal may not be sufficient to accurately convey the congestion level or effectively express the status of multiple priority queues. In practice, ECN is coupled with CNP (Congestion Notification Packet). CNP, associated with RoCE connections, offers per-connection congestion notifications with fine-grained information [36]. The combination introduces a simple and efficient congestion notification mechanism that has been extensively adopted in modern data center networks.



(a) Short path (Hop=1) **(b) Long path (Hop=5)**

Figure 1: Improper RTT baseline setting causes inefficient utilization

• **INT** (In-band Network Telemetry) provides near real-time information about the network status, including latency, queue length and packet loss of each hop, which helps CC algorithms to respond correctly to congestion throughout the paths [31, 52]. However, INT is heavily dependent on specialized switch hardware support, and the parsing of INT metadata is computationally demanding, making it difficult to deploy in existing network infrastructure [56]. Further more, the INT header is not yet able to be combined with RoCEv2 packets, and requires the sender to send probe packets with the same five-tuple as the QP traffic through middleware, which increases the end-side latency and CPU overhead. Nonetheless, when congestion arises at a hop that lacks support of INT, particularly in cases of congestion on the reverse path or across multiple hops, the system may fail to accurately identify and react to congestion. As a result, congested flows might persist in transmitting packets at elevated rates, thereby causing more severe congestion or unfair bandwidth allocation [47, 52].

• **RTT** (Round-Trip Time) provides a direct measure of end-to-end delay and an indication of overall network congestion. Modern NICs are capable of measuring RTT within a few microsecond [5, 9, 28], which causes RTT-based algorithms extremely sensitive to slight variation. However, RTT-based measurements exhibit significant dependency on path stability, where dynamic routing changes, fluctuating network device loads, and variable link quality can all introduce substantial measurement jitter. This instability frequently leads to erroneous congestion state estimation and subsequent misapplication of congestion control strategies. The temporal fluctuations in RTT measurements would be disproportionately amplified, triggering excessive rate reductions that significantly degrade overall network utilization. Specifically in multi-tier leaf-spine topologies, conventional congestion control algorithms cannot effectively distinguish between latency variations caused by variation of path length or network congestion. To clearly demonstrate this phenomenon, we implemented a basic RTT-based congestion control algorithm in a testbed comprising three 400Gbps NICs configured with two client nodes and one server node.

Figure 1 presents comparative throughput measurements

for two NICs of the server with different network topologies: (a) short distance communication in a single switch (Hop = 1, baseline RTT of 5μs); (b) an extended path traversing five switches in CLOS topology (Hop=5, measured RTT of 15μs). If the algorithm's baseline RTT is set inappropriately to 10μs for the long-distance communication, compared to short path (Figure 1 a), long path (Figure 1 b) transmission experienced inadequate bandwidth utilization even with no congestion. This experimental result demonstrates the practical challenge of defining a single RTT baseline suitable for all possible network paths and operating conditions in production deployments. While existing solutions attempt to address this through auxiliary mechanisms - such as queue delay extraction for INT header updates [52] or TTL-based hop count estimation [28] - these approaches incur non-trivial computational overhead that may offset their benefits in high-speed networks.

Rate Adjustment. Different types of rate adjustment mechanisms are employed in CC algorithms, such as window-based, rate-based, and credit-based approaches, each of which has raised several challenges in large-scale deployments.

• **Credit-based control** regulates the transmission rate by exchanging credits between the sender and the receiver. By detecting congestion in advance and estimating the link resources available to each node, it allows data transmission to be guaranteed with bounded delay and fast convergence. InfiniBand (IB) leverages the credit mechanism to ensure reliable data transmission at the hardware level and provides exceptionally low end-to-end latency [49]. However, deploying a private network with IB necessitates specialized NICs and switches, which typically incur hardware costs that are 5 to 10 times higher than those of conventional network equipment [8]. Moreover, IB faces scalability limitations due to its port count constraints and fabric complexity, making large-scale deployments challenging [12]. Due to its high performance, IB remains the gold-standard baseline in comparative evaluations. The primary design goal of our CC scheme is to achieve comparable performance of IB while overcoming its scalability barriers.

• **Window-based control** is a conventional technique extensively employed in traditional congestion control mechanisms (e.g. TCP). The transmission rate is regulated through the maintenance of congestion windows at both the sender and receiver, and the window size is adjusted based on the number of ACK packets returned by the receiver, ensuring a fine-grained response to congestion management. Window-based control depends on high-frequency ACK response (e.g. per-packet ACKs [28, 31]) and sequence headers that convey packet lengths to achieve precise control and accurate inflight byte statistics. Inflight byte control helps to monitor the amount of unacknowledged data and prevent sender from injecting excess packets to avoid network overload or receiver buffer overflow. Implementing this mechanism requires complex logic to maintain window sizes and acknowledgment information [53], along with supplementary algorithm to achieve accurate control in extremely congestion (e.g. window<1) under high-speed network. RDMA leverages ACK coalescing and Packet Sequence Number (PSN) [24,26], which precludes the direct implementation of a byte-oriented window. RNICs support only per-flow control, lacking the per-packet granularity necessary for precise window-based CC. Moreover, window-based congestion control mechanisms inherently depend on transport-layer state information to ensure reliable data delivery. This architectural design creates tight coupling between the congestion control implementation and the underlying transport protocol stack. Consequently, any algorithmic modification or enhancement necessitates corresponding hardware updates to the NIC's transport offload engines, significantly constraining the flexibility and evolvability of congestion control schemes in production environments [46].

• **Rate-based control** calculates the transmission rate directly based on network conditions and implements through a rate limiter or pacing module. The sender autonomously sends packets based on CC algorithm, and avoids complex calculation of ACK association, which leads to cumulative ACK in implementation [24, 26]. On the other hand, rate-based control decouples CC and reliable delivery, and makes the implementation of the transport layer less complicated. Therefore current RNICs predominantly support congestion control only at a rate-based per-flow granularity [18,55]. Rate-based congestion control exhibits inherent limitations compared with window-based approaches, e.g. its inability to explicitly bound the amount of inflight bytes. When congestion occurs, before receiving congestion signals, the transmitter continues injecting packets at pre-determined rates. In addition, rate-based control typically adopts the Additive Increase Multiplicative Decrease (AIMD) algorithm (such as DCQCN). The coexistence of dynamically changed traffic patterns and ultra-high-bandwidth NICs introduces fundamental challenges in parameter optimization. For instance, in the high-speed network of 400Gbps bandwidth, CC algorithms should adopt a large rate increase step to ensure rapid convergence. But if large-scale burst congestion occurs, an aggressive aggregate increment can lead to severe network oscillation and instantaneous congestion. These intrinsic limitations demand new congestion control designs.

3.3 Customized CC Design over RNICs

The rise of proprietary LLMs has driven demand for customizable networking, prompting NIC vendors to expose programmable interfaces that enable tailored solutions in data centers. Programmable NICs allow users to define customized congestion control algorithms to address variable network conditions and application demands.

Now SmartNICs have been widely deployed in current data centers (e.g. Nvidia ConnectX-6 DX [38] and BlueField-3 DPU [40]). These equipment provides programmable congestion control (PCC) platforms [37] and support multiple RoCE CC algorithms compatible with existing network protocol stacks. PCC provides hardware-based proactive network probes that are independent with transport layer [50], and automatically generates CC events when sending data or receiving ACK/ NACK/ CNP/ RTT packets. The embedded event APIs streamline the design of CC algorithms and reduce implementation overhead. For instance, PCC leverages dedicated RTT probe packet instead of add additional header to the data packets, mitigating the processing overhead associated with frequent RTT measurements. Upon sending a fixed amount of packets within a session, a TX event is generated which facilitates the measurement of accumulated sending byte within the current RTT and leverages the carried timestamp for timer functionality.

The design of CC algorithms leveraging the PCC capabilities provided by SmartNICs represents a cost-effective and efficient approach. This method ensures compatibility with mainstream commercial switches and NICs, thereby facilitating easier deployment in various network environments. With the iterative enhancement of NIC capabilities, e.g. the BlueField-3 SuperNIC now achieving CNP packets responses within 1 μs interval, CC algorithms can implement fine-grained rate control. Furthermore, implementing congestion control algorithms on RISC-V-based NIC architectures significantly reduces development cycles compared to conventional approaches, enabling tighter co-evolution with rapidly advancing hardware capabilities and application requirements.

Nevertheless, PCC has certain limitations that require compensation through CC algorithms. For instance, PCC supports only a per-flow granularity for rate adjustment, meaning that PCC-based CC algorithm also face the challenges of the poor responsiveness during severe congestion and the lack of inflight byte control. Specifically, the CC controller's trigger interval exhibits higher latency compared to FPGA-based SmartNICs, which achieve per-packet processing at granularity of several hundred nanoseconds, potentially resulting in delayed congestion responses and subsequent over-transmission. Additionally, PCC implementations typically operate on resource-constrained embedded CPUs, necessitating low-complexity CC algorithms to maintain both rapid response times and support for large quantities of queue pairs.

4 Barre Design

Based on the analysis of the characteristics and inherent limitations of RNICs, we propose Barre, a practical and scalable CC solution tailored for high-speed AI cluster networks. Barre ingeniously leverages the PCC capabilities and flexible event interfaces provided by RNICs to address the challenges and ensure compatibility with existing network infrastructure. Owing to its simple design, this solution can be easily deployed and utilized in any AI/HPC data center equipped with modern commercial RNICs.

4.1 Adaptive Adjustment Interval

Barre implements a rate-based congestion control architecture grounded in the widely-adopted AIMD framework. To address the issue of late responses introduced by timer-triggered rate adjustments under dynamically varying loads, Barre implements adaptive rate adjustment intervals based on CNP and RTT signals. We adopt CNP rather than RTT as the congestion signal because selecting an appropriate RTT baseline is challenging in large-scale deployments, which can lead to inadequate bandwidth utilization and unfair network allocation (as shown in Figure 1).

Delay-based increase: if the sender does not receive any CNP packets within a real-time RTT, the transmission rate (R) is updated with an additive increase according to the formula $R = R + \alpha$ (α refers to increment step). As RTT provides a straightforward indication of end-to-end latency across the network link, our approach employs real-time RTT monitoring to dynamically adapt the rate increase interval, enabling microsecond-scale adjustments that precisely track network conditions. When the link is idle, the frequency of rate increases is determined by the RTT baseline. During congestion, the frequency drops off as the end-to-end delay increases.

Per-CNP-based decrease: whenever the sender receives a CNP signal, the transmission rate is reduced according to the formula $R = R \times \beta$ immediately ($0.95<\beta<0.99$). The congestion indication in the IP header of CNP is widely employed by congestion control of RoCEv2 [39], where switches and routers will embed an indication for congestion once congestion is detected. Upon congestion detection, a CNP Packet is generated and routed back to the source, triggering precise rate limitation for the specified Queue Pair (QP). This closed-loop control mechanism enables microsecond-scale reaction to network congestion events. By monitoring switch buffer levels, ECN and CNP provide early congestion awareness. With modern NICs capable of generating CNP packets at the time granularity of 1 μs, Barre combines high-frequency rate reductions triggered per CNP with smaller reduction magnitudes. By adopting a slight but high frequency adjustment approach, the system can respond more precisely to congestion signals, while maintaining smooth network performance and avoiding significant fluctuations in transmission rates.

Dynamic self-constraint fairness: we find that DCQCN struggles with ensuring fairness between flows and adaptively allocating bandwidth in high-speed networks (as shown in Figure 12 a). In Barre design, per-CNP-based rate decrease auto-

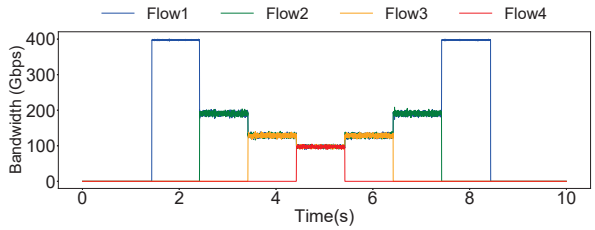


Figure 2: Throughput of 4 flows shows Barre fairness

matically achieves dynamic fairness between high-speed and low-speed flows within the same link. As link bandwidth approaches its bottleneck, high-speed flows, due to transmitting more data packets, receive a proportionally higher number of CNPs, thereby triggering a rate decrease more frequently. Consequently, this mechanism enables high-speed and low-speed flows to rapidly converge toward fair bandwidth allocation. Figure 2 illustrates the fairness achieved by Barre among four flows. The results indicate equitable allocation of bandwidth, maintaining full utilization without significant network oscillation or underutilization. Furthermore, we utilize a fluid model to analyze the theoretical stable point and stability of Barre. The analytical conclusions are verified by simulation experiments (refer to Appendix B).

4.2 Barre Functional Components

To address the limitations of RNIC and rate-based control, we propose three functional components: **Fast Increase**, **Dual-lock** and **Inflight Monitor**. These components are designed to be lightweight and decoupled from one another, enabling seamless integration into existing rate-based algorithms. The design philosophy underlying these features reflects the essence implied by the name "Barre", as a portable handle to make the trunk easier to use.

4.2.1 Fast Increase

The traditional AIMD algorithm applies fixed increase parameter α, which faces the dilemma between inadequate bandwidth utilization and burst transient congestion. When there is ample bandwidth available and the number of flows is relatively small (e.g. SendRecv collective), the rate increase should be more aggressive to improve network utilization. On the contrary, when the number of flows is very large (e.g. AlltoAll collective) or congestion has already occurred, rate increases must be more conservative to prevent excessive queue buildup and buffer overflow at switches. While dynamic increasing factors have been widely adopted as a solution, we attempt to explore a simple and easily implemented approach.

We propose Fast Increase as a solution to address this issue. To distinguish between aggressive and conservative rate increases, we use experienced rate increases over multiple RTTs as an accurate indicator of inadequate link bandwidth. When a flow experiences K times consecutive rate increases without receiving any CNP messages, it indicates substantial available bandwidth on the link. In this scenario, the increasing factor is set to a significantly large value, denoted as A, (e.g. 1/1000 of the bandwidth of NIC port), allowing for fast convergence. During the Fast Increase phase, if the sender receives a CNP signal, the increasing factor resets to its initial value α, and a rate decrease is triggered to mitigate potential congestion.

Integrating with Barre's technique of using real-time RTT for dynamic rate increase intervals, Fast Increase can achieve rapid algorithmic convergence within a very short period by large value A for mice flows, while keeping stability by small value α for elephant flows.

4.2.2 Dual-lock

Barre detects link congestion status via CNP signals received at the sender. Under severe congestion scenarios where BDP is less than MTU, the transmission rate becomes very low, requiring multiple RTTs to send a single data packet. Consequently, it also takes multiple RTTs to generate a CNP packet. This delay can result in several rounds of rate increase operation before the CNP reaches the sender, leading to an influx of additional packets into an already congested path. Currently, there are no effective solutions to address this issue. So we introduce a Dual-lock mechanism.

In QCN and DCQCN, the reaction points use both "ByteCounter" and "Timer" to control rate increase. For example, if the ByteCounter of an RP (reaction point, sender NIC) shows that the RP has transmitted 150 KB data or the Timer has passed 15 ms, RP will increase the sending rate. Generally, the RPs with larger sending rates will increase their rates more quickly, which are defined as the *QCN unfariness*. To solve this problem, the ByteCounter employed in QCN is modified to an adaptive ByteCounter [20]. DCQCN employs the relationship of ByteCounter "or" Timer as the condition of rate increase, which will cause high-speed flows (i.e. flow rate > 1.45 Gbps in DCQCN) to increase more frequently than low-speed flow (i.e. flow rate < 1.45 Gbps), leading to unfair bandwidth allocation.

Dual-lock optimizes rate increase conditions under severe congestion by modifying the relationship between the ByteCounter and Timer from "OR" to "AND", i.e. both conditions must be satisfied simultaneously to trigger a rate increase. As a result, the rate increase condition of high-speed flow is determined by Timer instead of Bytecounter, mitigating the unfairness issue. In practice, the Timer is generally set to one RTT. Therefore, for high-speed flows, the frequency of rate increase remains controlled by RTT, ensuring dynamic response to network conditions. Meanwhile, for low-speed flows, the rate increase is decided by the ByteCounter, preventing aggressive rate increases that could lead to congestion. This dual-lock approach effectively mitigates rapid rate increases during congestion while addressing fairness issues between high-speed and low-speed flows.

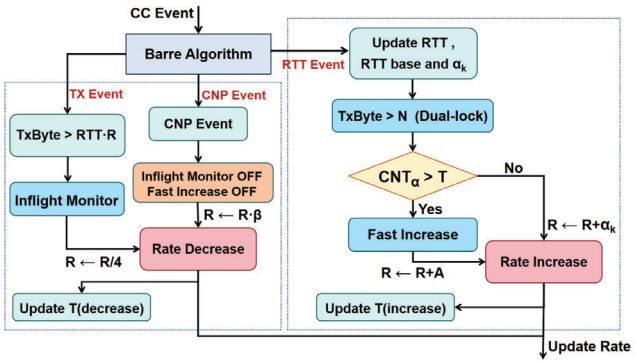


Figure 3: Process of CC module

4.2.3 Inflight Monitor

Due to the lack of fine-grained ACK responses and Seq packet headers that carry payload sizes, RNICs cannot effectively track and limit the amount of inflight bytes that have been sent but not yet acknowledged. During severe congestion, delayed CNP feedback can prevent the CC algorithm from fast respond, causing the sender to continue injecting packets and exacerbating congestion.

PCC periodically generates TX events that include information on the volume of data transmitted each time. Based on this new functionality, we propose an Inflight Monitor mechanism as a safeguard to manage in-flight data during congestion. Inflight Monitor tracks the accumulated sending bytes within the current RTT, denoted as S_{now}. Based on the current transmission rate R and the real-time RTT value, we can estimate the maximum allowable data volume for the current RTT, denoted as $\gamma = R \times RTT$. If S_{now} exceeds the estimated threshold γ, it indicates potential network congestion, even though no CNP signal or RTT probe packet is returned to the sender. Then inflight byte control is triggered, and the current transmission rate is immediately reduced to one-quarter of its original value. The rate remains at this reduced level until a CNP signal is received, at which point the transmission rate is restored to its initial value.

4.3 Effect Improvement of Barre System

The control flow in Barre, as illustrated in Figure 3, is driven by TX/RTT/CNP event provided by PCC. These events trigger corresponding actions such as rate increase, rate decrease, or optimized features operations, ensuring efficient and adaptive management of network traffic under varying conditions. The main function of Barre and the conditions for the functional component are presented in Algorithm 1.

The three components of Barre can be utilized independently, while the coupling yields significantly enhanced performance benefits. To evaluate the effectiveness, we start from Fast Increase and enable them one by one gradually, and compare with the default set as basic Barre algorithm.

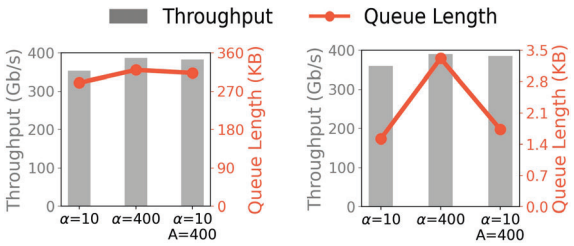
Algorithm 1: Barre Main Function

```
function OnTransmitPacket(Tx_Pkt):
    R_prev ← R_cur // update sending rate
    Update TxBytes;
    if TxBytes_incre > N and Timer > RTT then
        // Exceed Dual-lock threshold, rate increase
        if CNP_RECV = 0 then
            if CNT_α > T then
                R_cur ← R_prev + A // Fast increase
            else
                R_cur ← R_prev + α
            CNT_α ++ // Counter of Fast increase
        CNP_RECV ← 0 , TxBytes ← 0
    if TxBytes_Inflight > R_cur × RTT and FN_Inflight == 0
     then
        FN_Inflight ← 1; R_cur ← R_prev × 1/4
        // Enable Inflight Monitor
function OnRecvRTTResponse(RTT_Pkt):
    Update RTT
    if RTT < RTT_base then
        RTT_base ← RTT
        // Update RTT base for each flow
function OnRecvCNP(CNP_Pkt):
    R_cur ← R_prev × β // β: ratio
    CNP_RECV ← 1, CNT_α ← 0
    TxBytes_inflight = 0
    TxBytes_incre = 0
    if FN_Inflight = 1 then
        R_cur ← R_prev × 4, FN_Inflight ← 0
        // Disable Inflight Monitor, recover original rate
```

First, we evaluate Fast Increase with experiments of two distinct scenarios: mice flow (less connections as QP=4) and elephant flow (large-scale incast as QP=1000). Compared with fixed rate increase magnitude as α=10 or α=400, we employ both the base increase factor (α=10) and the aggressive increase factor (A=400 MB) in Fast Increase. The results are illustrated in Figure 4.

Communication of mice flow (QP=4) has almost no congestion, resulting in negligible differences in switch queue lengths among the three methods, as indicated by the yellow line in Figure 4 (a). Fast Increase achieves higher throughput than the set of small α=10 (an increase of 8.5%) for mice flow in QP=4, and keeps lower queue lengths compared with the aggressive α=400 (an improvement of 48%) for elephant flow in QP=1000. The experimental results demonstrate that Fast Increase effectively balances rapid convergence and low latency. It adapts dynamically to varying traffic conditions, ensuring optimal bandwidth utilization without compromising stability.

Then we enable Dual-lock function and evaluate the average queue occupancy at the switch under varying levels



(a) QP=4 (b) QP=1000

Figure 4: Performance comparison of different increasing factors in mice flow and elephant flow. Fast Increase achieves optimal bandwidth utilization while keeping low queue length in different traffic patterns.

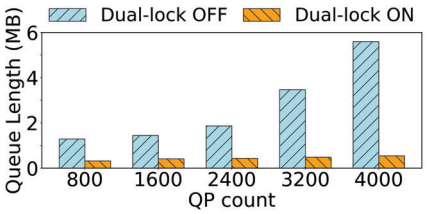


Figure 5: Dual-lock maintains extremely low switch queue length in large-scale Incast.

of congestion. In this experiment, four clients are connected to a single server through a ToR switch, and we continuously increase the number of flows involved in the traffic to simulate an Incast scenario. The ByteCounter threshold for Dual-lock is set to 8KB. As illustrated in Figure 5, across five experimental setups, enabling Dual-lock consistently reduces the average switch queue length while maintaining the same throughput, achieving an overall reduction of 79.9% (with a maximum reduction of 90.25% when QP=4000). This significant decrease in queue length indicates a substantial alleviation of congestion. Moreover, Fast Increase maintains a relatively small increment factor α during congestion, which can lead to high switch buffer occupancy. The combination with Dual-lock achieves higher throughput while keeping low buffer occupying.

We enable the Inflight Monitor function in the last step. Figure 6 illustrates the performance improvements achieved by coupling of the three components. To emulate real training workloads, we conducted NCCL AlltoAll test on a cluster composed of 256 Nvidia BlueField-3 400 Gbps NICs, recording the total bus bandwidth for varying message sizes. The results indicate that when Inflight Monitor is enabled, the throughput is improved on average by 16.45%, with a maximum increase of 21.79%. These findings demonstrate that Inflight Monitor can significantly enhance network performance of large-scale incast congestion scenarios.

Barre is an event-driven algorithm that leverages TX Event, CNP event and RTT event provided by the PCC functionality for its design. These events incur lower overhead compared to ACK events and Timer events, making them more suitable for

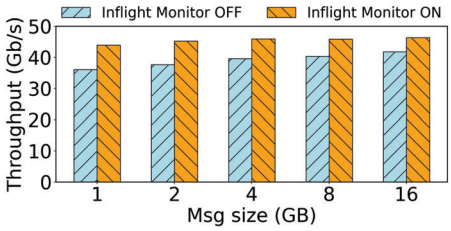


Figure 6: Average throughput in NCCL AlltoAll test. The inflight Monitor function significantly improves the throughput in large-scale congestion.

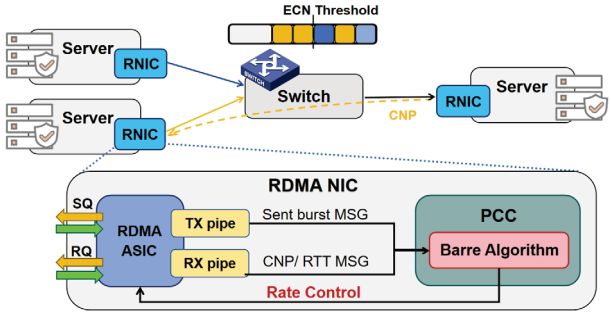


Figure 7: System architecture

efficient processing by both software and hardware. TX Event, serving as the most critical event for Barre, offers accumulated sending byte and timestamp information. Coupled with the application of Barre's three components, this significantly reduces the number of events processed by CC algorithm.

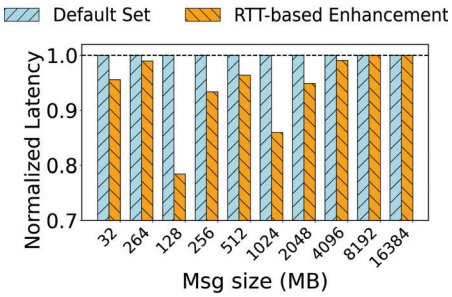
5 Implementation

Barre is implemented on BlueField-3 SuperNICs, and is designed to leverage the maximal potential of the event interface provided by PCC, achieving integration of software, hardware and algorithm. As illustrated in Figure 7, PCC-based software-programmable CC operates directly on the network layer of a dedicated RISC processor embedded within the NIC. This setup facilitates the utilization of transmission (TX) event and reception (RX) events for synchronizing communication operations between applications and the NIC, ensuring reliable and efficient data transmission and reception.

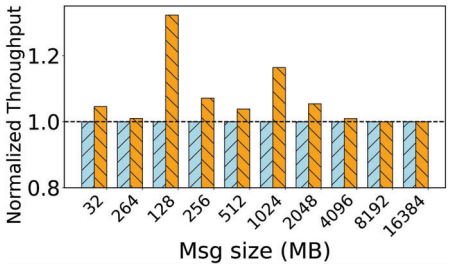
5.1 RTT-based Enhancement

Barre dynamically adjusts rate increase interval based on real-time RTT to adaptively respond to network state. We find several challenges to leveraging RTT as an indicator in deployment and propose two mechanisms to address the problems.

RTT Probe Modification. Barre dynamically adjusts rate increase interval based on real-time RTT to adaptively respond to the network state. PCC provides a dedicated RTT probe packet, while SP (Sender NIC) generates an RTT_Req packet



(a) Normalized Latency



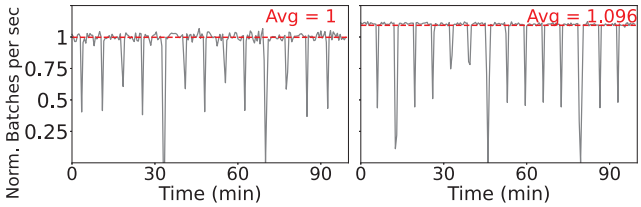
(b) Normalized Throughput

Figure 8: The normalized latency and throughput of RTT-based enhancement in 128-GPU NCCL AlltoAll test.

at the beginning of an RTT, and RP (Receiver NIC) generates an RTT_Rsp packet. Thus Barre executes a rate increase at the time RP receives RTT_Rsp packet. However, due to limitations of hardware NIC (e.g. packet loss or delay), RTT probe packets may be lost, leading to missed rate increases for Barre. In a test using NCCL AlltoAll across a three-layer Clos network cluster of 128 GPUs, we find that the loss rate of RTT probe packets can achieve as high as **8.9%**.

To address this issue, we implement an RTT probe modification by leveraging the head carried in RTT probe packets. We add a sequence header into the RTT probe packet to avoid mismatching. As an RTT packet involves timestamp information, we make each RTT_Rsp packet carry both the sending time of RTT_Req and the receiving time of RTT_Rsp. Thus the sender could accurately calculate the real-time RTT by simply computing the difference between the timestamps. This method prevents mismatching of RTT probe packets from different batches, thereby avoiding errors in the calculation of real-time RTT. Thus, we modify Barre's rate increase logic to update the last $real-timeRTT$ whenever an RTT probe packet returns. A rate increase is triggered if no CNP signal is received by RP within a period equivalent to the $real-timeRTT$.

RTT-based per-flow Increase Factor. Additionally, we find that RTT-based rate increases may lead to unfairness among flows with different path lengths, causing lower frequency of rate increases for longer-distance transmissions. Therefore, we adjust the rate increase magnitude for different flows, correlating the value of the increase factor α with the baseline RTT of each flow, as $\alpha_k = RTT_k \cdot \alpha / C$. While C denotes a



(a) Default set **(b) RTT-based enhancement**

Figure 9: The throughput log from real training tasks (normalized to average throughput of default set). The red dashed line represents the average throughput under the default settings.

constant set to 1us or 2us depending on the network infrastructure. For flow k, Barre continuously measures and updates the minimum real-time RTT value observed during actual communication, and denoted as RTT_k. Therefore, the increasing factor α_k of long-distance transmission is larger due to longer RTT, which makes up for the unfairness leveraged by a lower increase frequency.

We evaluated RTT-based enhancement on a 128-GPU cluster, contrasting it with the default set with an increase factor uniformly applied to all flows and without RTT probe timestamp. The latency and bandwidth of the NCCL AlltoAll test were recorded for various message sizes, with performance normalized to the uniform increase factor, as illustrated in Figure 8. Utilizing the RTT-based enhancement resulted in an average reduction of 5.71% in latency and a corresponding increase of 7.13% in throughput.

As AlltoAll messages in AI training workloads typically do not exceed 1GB, actual tasks are more sensitive to loss of rate increase operation and RTT probe packets compared to NCCL tests. As shown in Figure 9, in end-to-end testing, we observed that employing RTT-based enhancement effectively reduced the latency of AlltoAll communication by up to 50%, and improved the overall throughput of AI training task at an average of 9.6%.

5.2 In-production Deployment

Barre can be fully deployed on hardware NIC, maintaining complete transparency to the application layer without modifications to RoCEv2 protocol. For instance, with Nvidia BlueField-3 SuperNIC, CC algorithm can be compiled into a binary file, flashed onto the NIC firmware, and executed on the RISC-V cores embedded within the NIC. This design allows the entire CC process to bypass the PCIe channel, eliminating the need for host CPU involvement, which significantly improves network efficiency and congestion response speed. Three functional components of Barre address the limitations proposed by RNIC and rate-based control and achieve the fundamental requirements as discussed in Section 3.2. Practical tests have demonstrated that Barre can leverage the four RISC-V cores on the NIC to manage congestion events

at a rate of 10 million events per second, showcasing excellent responsiveness to network congestion.

Our production cluster workloads consist of two types: LLM training and inference. In LLM training tasks spanning thousands of GPUs, collective communications consist of ReduceScatter, AllGather, AlltoAll, and SendRecv operations. In most scenarios, AllGather and ReduceScatter operations dominate, accounting for over 50% of the total communication volume, with message sizes typically at the gigabyte level. MoE models exhibit particularly heavy AlltoAll traffic, where message sizes can expand to 10^8 bytes with increasing batch sizes. For inference tasks operating at smaller scales, typically hundreds of GPUs, AlltoAll communications account for more than 50% of traffic with generally smaller messages (<1 GB). AllReduce operations form the second most significant portion of communications. Notably, execution alternates between AlltoAll full-mesh traffic and ReduceScatter ring-based traffic, causing the network to frequently switch between large-scale incast and uncongested states. This variation requires the CC algorithm rapidly adapting to dynamic traffic pattern, while mitigating congestion caused by elephant flows and also ensuring fast convergence of mice flows.

6 Evaluation

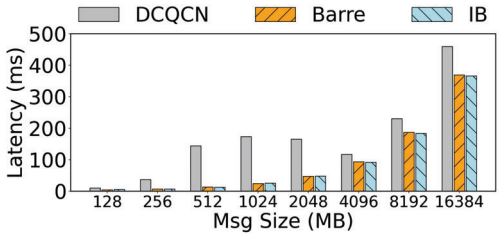
In this section, we evaluate the effectiveness of Barre on large-scale NCCL AlltoAll test (Section 6.2) in comparison with InfiniBand and DCQCN. In addition, we modify a few parameter settings of DCQCN based on the idea of adaptive adjustment interval achieved in Barre (Section 6.3) to demonstrate flexibility and versatility of Barre.

6.1 Network Topology

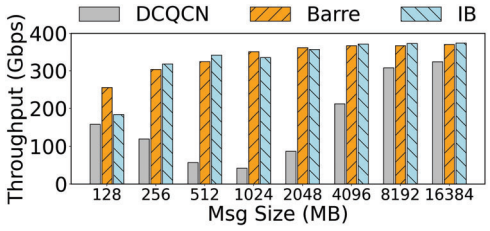
Barre has been deployed in a AI production cluster for nearly one year. The network architecture employs a three-layer switch topology similar to CLOS to interconnect over 10K GPUs, each equipped with a BlueField-3 SuperNIC supporting 400Gbps port bandwidth. For switches at each layer, the bandwidth ratio between downlink and uplink is 1:1. For the ToR switches, a specific Active Optical Cable (AOC) is used to split an 800 Gbps downlink port into two 400 Gbps downlink ports. Each server involves eight 400 Gbps NICs, and is connected to eight different ToR switches with multi-rail topology. A total of 64 GPU servers are interconnected through this set of ToR switches.

6.2 Large-scale Collective Communication

We conducted the congestion control test on a 256-GPU cluster to test Barre's performance. The experimental setting is the same as that in section 6.1. Each GPU was equipped with an Nvidia BlueField-3 400Gbps NIC. As discussed in Section 2.2, the distributed parallel approach for LLM training



(a) Latency



(b) Bus Bandwidth

Figure 10: NCCL AlltoAll test in 256-GPU cluster

involves a significant amount of collective communication traffic. Among 3D parallelisms, AlltoAll dominates and is more likely to cause high ephemeral congestion due to its full-mesh communication mode.

To validate the superior performance of Barre, we conducted comparative experiments using DCQCN [60] and InfiniBand (IB), two commonly employed CC mechanisms in AI networks. In NCCL AlltoAll test, the overall communication volume (message size) was incrementally expanded from 128 MB to 16 GB (2^{14} MB), with corresponding network latency and average bandwidth (message size divided by flow completion time) recorded. The experimental results are illustrated in Figure 10. Observations indicate that Barre achieves superior latency and throughput comparable to IB while significantly outperforming DCQCN. Compared to IB, Barre exhibits substantially lower hardware deployment costs and implementation complexity, along with enhanced scalability in large-scale clusters. Therefore Barre offers a simple, high-performance, and cost-effective CC solution.

In contrast, DCQCN displayed a pattern where latency initially increased and then decreased as message sizes grew. The complexity of DCQCN's parameter takes a long time to complete multiple iterations, which leads to a fluctuation of latency and throughput during smaller message sizes (128 MB to 1024 MB). As message sizes further increase, DCQCN gradually converges and improves bandwidth utilization. Throughout the experiment, both Barre and IB demonstrated excellent convergence properties, maintaining high throughput and low latency even under full mesh traffic conditions in large-scale environments.

Figure 11 illustrates switch traffic logs collected over a period from the production environment. Barre has enhanced communication across a four-layer switch architecture (S3). It achieves near-optimal utilization without triggering PFC

Table 2: DCQCN Parameter Tuning Settings

Parameter	Default Value	Tuned Value	Explanation
min_time_between_cnps	4	0~2	Interval for Receiver Responding with CNP (us)
rpg_min_dec_fac	50	80~98	The minimum reduction at each rate decrease
rate_reduce_monitor_period	4	1~4	Minimum deceleration interval (us)
rpg_alpha_rate	5	8~20	Rate increase magnitude
rpg_time_reset	300	20~100	Rate increase interval (us)

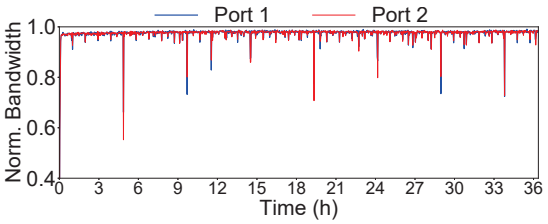


Figure 11: Switch traffic in production

event. Moreover, Barre exhibits robust convergence and fairness under varying traffic loads. Its capability ensures that the network remains stable and efficient, effectively supporting the demanding requirements of large-scale AI workloads.

Given that most CC algorithm are not well-aligned with the characteristics of current mainstream hardware, implementing them in AI/HPC clusters poses significant challenges. We provide a qualitative analysis to compare against other CC protocols (e.g. Swift [28], PowerTCP [1] and HPCC [31]) in Appendix A.1.

Since ring-based collective communications (e.g. AllReduce, AllGather, and ReduceScatter) rarely cause congestion compared to AlltoAll, we present Barre's performance of these operations in Appendix A.2 (Figure 14). Experimental results show that Barre achieves near-full bandwidth for both ReduceScatter and AllGather communications with NVLink disabled to enforce RDMA routing. We also conduct experiments on extreme incast scenarios in Appendix A.3 (with results shown in Figure 15).

6.3 Barre-inspired Optimization on DCQCN

To demonstrate the flexibility and versatility of Barre, we apply its design ideas to optimize DCQCN by adjusting certain parameters. DCQCN, originally developed in 2014, was tailored to leverage the hardware characteristics of Mellanox NICs. Given the technological limitations a decade ago, NIC capabilities were relatively constrained, leading to the implementation of a CNP response interval set at 55 us. The algorithm adjusts the transmission rate based on the number of consecutive CNPs received by the sender over fixed intervals, i.e. as the count of CNPs increased across multiple periods, parameter α would increase while parameter β would decrease accordingly.

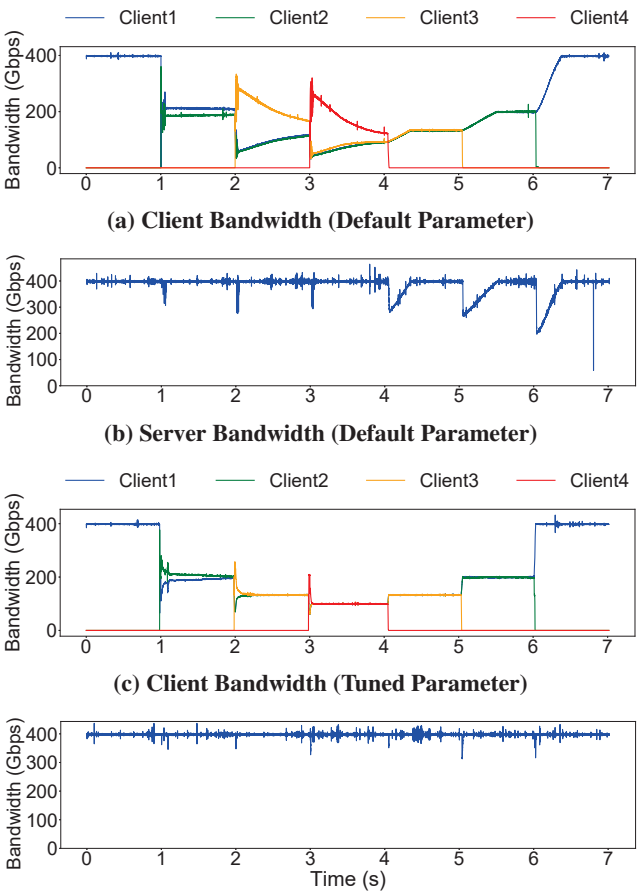
As modern NIC technology has significantly advanced, currently the BF-3 SuperNIC is capable of responding with CNP messages for every 0 to 2 us. We applied the principles of Barre to optimize the DCQCN parameter. The modified parameter settings are shown in Table 2. The CNP packet generation interval is set a smaller value to fit modern NICs, and each CNP will trigger a rate decrease operation with a small decrease step. We recommend setting the increase interval to a range of 20~100 ms, depending on the maximum number of incast flows and the base RTT in the network. A larger number of concurrent incast flows necessitates a longer interval to avoid congestion. For α, we suggest a higher value between 8 and 20 in 400G networks to enable faster rate increases and convergence.

We selected four nodes as clients (Client 1 to Client 4), and the other one as the server, all connected to a single ToR switch. During the experiment, each client established connections with the server according to a predefined schedule, and then gradually terminated the connection one by one.

Figures 12(a) and (b) illustrate bandwidth utilization of the four clients and the server under unmodified DCQCN parameter (Default value in Table 2). As clients progressively joined the network and established connections, the link bandwidth was dynamically reallocated, leading to oscillations in network bandwidth, and low utilization (e.g. 197.7 Gbps at 6s). Despite gradual convergence towards a more balanced bandwidth distribution, the allocations remained unfair throughout the experiment.

Figures 12(c) and (d) present the results following parameter modification to DCQCN based on Barre (Tuned value in Table 2). The observation indicates that the system rapidly becomes stable and achieves fair bandwidth sharing across different flows. Moreover, as connections are progressively terminated, the network quickly converges to a stable bandwidth allocation condition. Specifically, Figure 12(d) shows that during the experimenting period, the server NIC bandwidth remains close to full utilization and exhibits stability. The frequency and magnitude of network oscillations are significantly reduced compared to Figure 12(b). These results demonstrate the effectiveness of the Barre-inspired parameter adjustments in enhancing the responsiveness and fairness of DCQCN, thereby ensuring efficient and stable network operation for dynamically changed traffic loads.

Additionally, we conducted NCCL AllReduce test on a 1024-GPU cluster to evaluate the effect of Barre's inspired op-



(a) Client Bandwidth (Default Parameter)

(b) Server Bandwidth (Default Parameter)

(c) Client Bandwidth (Tuned Parameter)

(d) Sever Bandwidth (Tuned Parameter)

Figure 12: Barre-inspired optimization on DCQCN

timization. Employing multi-rail optimized network topology with a ring algorithm significantly mitigates ECMP hashing conflicts, which could prevent most congestion during AllReduce communication [27]. However, we notice that numerous data centers have not upgraded to multi-rail topologies, which means AllReduce remains a performance problem. Figure 13 illustrates the normalized bus bandwidth across various message sizes of both default and tuned parameters of DCQCN. Following optimization with Barre, an average improvement of 19.54% of throughput was achieved. In addition, the resource allocation among different flows has become fairer. As AlltoAll and AllReduce communication account for the dominant proportion in large-scale AI training tasks, Barre-inspired optimization on DCQCN could achieve significant end-to-end revenue, with only a few modifications of parameter.

7 Related Work

Congestion control is critical for optimizing network performance and ensuring reliable communication.

Host-driven Congestion Control: Host-driven CC can be further divided into sender-driven CC and receiver-driven CC. Currently, sender-driven CC is widely deployed in data center

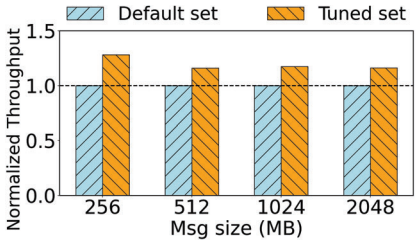


Figure 13: NCCL AllReduce test

networks. DCQCN [60] uses ECN as the congestion signal [3, 41]. Another widely used congestion control signal is Delay or RTT [5,9,28–30,33]. TIMELY [33] and Swift [28] utilize RTT and RTT gradient to ensure network transmission efficiency without the involvement of switches [61]. INT signal has been proposed to provide more detailed congestion information [1, 31]. PowerTCP [1] calculates the congestion status based on the bandwidth-window product. Receiver-driven CC aims at maintaining near-zero queues in switches [11, 19, 22, 23, 25, 34, 42]. ExpressPass [54] uses credits to probe bandwidth, and the receiver regulates the transmission rate based on the credit loss rate. HOMA [34] prioritizes small flows to ensure low latency for short messages.

Switch-driven Congestion Control: The switches make appropriate rate adjustment decisions instead of announcing congestion status. RoCC [48] utilizes queue length as a congestion signal and follows the Proportional Integral Control method to compute the fair rate. It relies on the switches to generate CNPs containing the fair rate to directly control the rate decrease of the senders. HierCC [58] employs virtual queue length as a congestion signal and divides congestion into two types: the congestion between ToR and NICs, and the congestion between ToR. It implements direct rate allocation to adjust sending rates and limit queue lengths. ACCurate [17] makes use of Flow Rate Packets (FRPs) injected by the senders as congestion information. When the switch detects an FRP, it allocates the minimum flow rate based on the number of flows on the egress port.

8 Conclusion

Barre exploits recent trends including programmable congestion control and proposes a decoupled design of congestion control atop the transport layer stack. By careful design to efficiently use the advanced features of TX event based byte-counter and timestamp, Barre introduces an adaptive rate decrease per CNP and dynamic rate increase based on RTT probing, which meets the demand for fast response to traffic variance and low software and hardware overhead. Barre has been implemented in 400Gbps Bluefield-3 NICs as part of our production data center. Barre exhibits not only superior performance in terms of fairness and congestion response but is able to enhance DCQCN, which converges faster and maintains better fairness and stability.

References

[1] Vamsi Addanki, Oliver Michel, and Stefan Schmid. PowerTCP: Pushing the performance limits of datacenter networks. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 51–70, Renton, WA, 2022. USENIX Association.

[2] Saksham Agarwal, Arvind Krishnamurthy, and Rachit Agarwal. Host congestion control. In *Proceedings of the ACM SIGCOMM 2023 Conference*, ACM SIGCOMM '23, pages 275–287, New York, NY, USA, 2023. Association for Computing Machinery.

[3] Mohammad Alizadeh, Albert Greenberg, David A Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data center tcp (dctcp). In *Proceedings of the ACM SIGCOMM 2010 Conference*, SIGCOMM '10, pages 63–74, New York, NY, USA, 2010. Association for Computing Machinery.

[4] Serhat Arslan, Yuliang Li, Gautam Kumar, and Nandita Dukkipati. Bolt: Sub-RTT congestion control for Ultra-Low latency. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 219–236, Boston, MA, 2023. USENIX Association.

[5] Venkat Arun and Hari Balakrishnan. Copa: Practical Delay-Based congestion control for the internet. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 329–342, Renton, WA, 2018. USENIX Association.

[6] InfiniBand Trade Association et al. Supplement to infiniband architecture specification. *Release*, 1(2):1, 2010.

[7] Pytorch To Atoms. Meta's 24k h100 cluster capex/tco and bom analysis. https://open.substack.com/pub/pytorchtoatoms/p/metas-24k-h100-cluster-capextco-and?utm_campaign=post&utm_medium=web.

[8] Candy798. A comparative analysis of infiniband and roce for ai data centers. https://medium.com/@lixian_58397/a-comparative-analysis-of-infiniband-and\-roce-for-ai-data-centers-78a99c39881c.

[9] Neal Cardwell, Yuchung Cheng, C Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. Bbr: congestion-based congestion control. *Commun. ACM*, 60(2):58–66, 2017.

[10] Biyao Che, Yuxiang Wang, Zirui Wan, Ying Chen, Zixiao Wang, Yuan Tian, Jizhuang Zhao, Shuo Wang, and Jiao Zhang. Fcc: A fast-converging low-latency congestion control algorithm for datacenter rdma network. In *Proceedings of the 8th Asia-Pacific Workshop on Networking*, APNet '24, pages 200–201, New York, NY, USA, 2024. Association for Computing Machinery.

[11] Inho Cho, Keon Jang, and Dongsu Han. Credit-scheduled delay-bounded congestion control for datacenters. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, pages 239–252, New York, NY, USA, 2017. Association for Computing Machinery.

[12] Massed Compute. Scalability limitations of nvidia's infiniband solutions compared to ethernet. https://massedcompute.com/faq-answers/?question=What%20are%20the%20scalability%20limitations%20of%20NVIDIA%27s%20InfiniBand%20solutions%20compared%20to%20Ethernet?

[13] Mo Dong, Tong Meng, Doron Zarchy, Engin Arslan, Yossi Gilad, Brighten Godfrey, and Michael Schapira. PCC vivace: Online-Learning congestion control. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 343–356, Renton, WA, 2018. USENIX Association.

[14] Adithya Gangidi, Rui Miao, Shengbao Zheng, Sai Jayesh Bondu, and Goes. Rdma over ethernet for distributed training at meta scale. In *Proceedings of the ACM SIGCOMM 2024 Conference*, ACM SIGCOMM '24, page 57–70, New York, NY, USA, 2024. Association for Computing Machinery.

[15] Adithya Gangidi, Rui Miao, Shengbao Zheng, Sai Jayesh Bondu, Guilherme Goes, Hany Morsy, Rohit Puri, Mohammad Riftadi, Ashmitha Jeevaraj Shetty, Jingyi Yang, Shuqiang Zhang, Mikel Jimenez Fernandez, Shashidhar Gandham, and Hongyi Zeng. Rdma over ethernet for distributed training at meta scale. In *Proceedings of the ACM SIGCOMM Conference*, 2024.

[16] Yixiao Gao, Yuchen Yang, Tian Chen, Jiaqi Zheng, Bing Mao, and Guihai Chen. Dcqcn+: Taming large-scale incast congestion in rdma over ethernet networks. In *2018 IEEE 26th International Conference on Network Protocols (ICNP)*, pages 110–120. IEEE, 2018.

[17] Dimitris Giannopoulos, Nikos Chrysos, Evangelos Mageiropoulos, Giannis Vardas, Leandros Tzanakis, and Manolis Katevenis. Accurate congestion control for rdma transfers. In *2018 Twelfth IEEE/ACM International Symposium on Networks-on-Chip (NOCS)*, pages 1–8. IEEE, 2018.

[18] Ernst Gunnar Gran, Magne Eimot, Sven-Arne Reinemo, Tor Skeie, Olav Lysne, Lars Paul Huse, and Gilad

Shainer. First experiences with congestion control in infiniband hardware. In *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, pages 1–12. IEEE, 2010.

[19] Mark Handley, Costin Raiciu, Alexandru Agache, Andrei Voinescu, Andrew W Moore, Gianni Antichi, and Marcin Wójcik. Re-architecting datacenter networks and stacks for low latency and high performance. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 29–42, New York, NY, USA, 2017. Association for Computing Machinery.

[20] Yuki Hayashi, Hayato Itsumi, and Miki Yamamoto. Improving fairness of quantized congestion notification for data center ethernet networks. In *International Conference on Distributed Computing Systems Workshops*, 2011.

[21] Sara Hooker. The hardware lottery. *Communications of the ACM*, 64(12):58–65, 2021.

[22] Jinbin Hu, Jiawei Huang, Zhaoyi Li, Yijun Li, Wenchao Jiang, Kai Chen, Jianxin Wang, and Tian He. Rpo: Receiver-driven transport protocol using opportunistic transmission in data center. In *2021 IEEE 29th International Conference on Network Protocols (ICNP)*, pages 1–11. IEEE, 2021.

[23] Shuihai Hu, Wei Bai, Gaoxiong Zeng, Zilong Wang, Baochen Qiao, Kai Chen, Kun Tan, and Yi Wang. Aeolus: A building block for proactive transport in datacenters. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, SIGCOMM '20, pages 422–434, New York, NY, USA, 2020. Association for Computing Machinery.

[24] Chengyuan Huang, Yixiao Gao, Wei Chen, Duoxing Li, Yibo Xiao, Ruyi Zhang, Chen Tian, Xiaoliang Wang, Wanchun Dou, Guihai Chen, et al. Mc-rdma: Improving replication performance of rdma-based distributed systems with reliable multicast support. In *2023 IEEE 31st International Conference on Network Protocols (ICNP)*, pages 1–11. IEEE, 2023.

[25] Jiawei Huang, Shuping Li, Rui Han, and Jianxin Wang. Receiver-driven fair congestion control for tcp outcast in data center networks. *Journal of Network and Computer Applications*, 131(C):75–88, 2019.

[26] InfiniBand. Infiniband architecture specification volume 1, 2021.

[27] Ziheng Jiang, Haibin Lin, Yinmin Zhong, Qi Huang, Yangrui Chen, Zhi Zhang, Yanghua Peng, Xiang Li, Cong Xie, Shibiao Nong, Yulu Jia, Sun He, Hongmin Chen, Zhihao Bai, Qi Hou, Shipeng Yan, Ding Zhou, Yiyao Sheng, Zhuo Jiang, Haohan Xu, Haoran Wei, Zhang Zhang, Pengfei Nie, Leqi Zou, Sida Zhao, Liang Xiang, Zherui Liu, Zhe Li, Xiaoying Jia, Jianxi Ye, Xin Jin, and Xin Liu. MegaScale: Scaling large language model training to more than 10,000 GPUs. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 745–760, Santa Clara, CA, 2024. USENIX Association.

[28] Gautam Kumar, Nandita Dukkipati, Keon Jang, Hassan MG Wassel, Xian Wu, Behnam Montazeri, Yaogong Wang, Kevin Springborn, Christopher Alfeld, Michael Ryan, et al. Swift: Delay is simple and effective for congestion control in the datacenter. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, SIGCOMM '20, pages 514–528, New York, NY, USA, 2020. Association for Computing Machinery.

[29] Changhyun Lee, Chunjong Park, Keon Jang, Sue Moon, and Dongsu Han. Accurate latency-based congestion feedback for datacenters. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, USENIX ATC '15, pages 403–415, Santa Clara, CA, 2015. USENIX Association.

[30] Changhyun Lee, Chunjong Park, Keon Jang, Sue Moon, and Dongsu Han. Dx: Latency-based congestion control for datacenters. *IEEE/ACM Transactions on Networking*, 25(1):335–348, 2017.

[31] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, et al. Hpcc: High precision congestion control. In *Proceedings of the ACM special interest group on data communication*, SIGCOMM '19, pages 44–58. Association for Computing Machinery, New York, NY, USA, 2019.

[32] Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437*, 2024.

[33] Radhika Mittal, Vinh The Lam, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. Timely: Rtt-based congestion control for the datacenter. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, pages 537–550, New York, NY, USA, 2015. Association for Computing Machinery.

[34] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John Ousterhout. Homa: A receiver-driven low-latency transport protocol using network priorities. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, pages 221–235, New York, NY, USA, 2018. Association for Computing Machinery.

[35] Broadcom Inc Moshe Voloshin, System Architect. Introduction to congestion control for roce. *Broadcom NCC-WP103*, 2023.

[36] Broadcom Inc. Moshe Voloshin, System Architect. Introduction to congestion control for roce, 2023.

[37] NVIDIA. Doca pcc. https://docs.nvidia.com/doca/sdk/doca+pcc/index.html.

[38] NVIDIA. Nvidia connectx-6 dx cisco ethernet smartnic, 2021.

[39] Nvidia. Understanding rocev2 congestion management, 2022.

[40] NVIDIA. Nvidia bluefield-3 networking platform, 2023.

[41] Rong Pan, Balaji Prabhakar, and Ashvin Laxmikantha. Qcn: Quantized congestion notification. *IEEE802*, 1:52–83, 2007.

[42] Jonathan Perry, Amy Ousterhout, Hari Balakrishnan, Devavrat Shah, and Hans Fugal. Fastpass: A centralized" zero-queue" datacenter network. *SIGCOMM Comput. Commun. Rev.*, 44(4):307–318, 2014.

[43] Kun Qian, Yongqing Xi, Jiamin Cao, Jiaqi Gao, Yichi Xu, Yu Guan, Binzhang Fu, Xuemei Shi, Fangbo Zhu, Rui Miao, et al. Alibaba hpn: A data center network for large language model training. In *Proceedings of the ACM SIGCOMM 2024 Conference*, pages 691–706, 2024.

[44] Kun Qian, Yongqing Xi, Jiamin Cao, Jiaqi Gao, Yichi Xu, Yu Guan, Binzhang Fu, Xuemei Shi, Fangbo Zhu, Rui Miao, Chao Wang, Peng Wang, Pengcheng Zhang, Xianlong Zeng, Eddie Ruan, Zhiping Yao, Ennan Zhai, and Dennis Cai. Alibaba hpn: A data center network for large language model training. In *Proceedings of the ACM SIGCOMM Conference*, 2024.

[45] Yuval Shpigelman, Idan Burstein, Noam Bloch, Reut Zuck, and Roee Moyal. Programmable congestion control communication scheme, April 5 2022. US Patent 11,296,988.

[46] Athinagoras Skiadopoulos, Zhiqiang Xie, Mark Zhao, Qizhe Cai, Saksham Agarwal, Jacob Adelmann, David Ahern, Carlo Contavalli, Michael Goldflam, Vitaly Mayatskikh, et al. High-throughput and flexible host networking for accelerated computing. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 405–423, 2024.

[47] John Snyder and Alvin R Lebeck. Fast convergence to fairness for reduced long flow tail latency in datacenter networks. In *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1007–1017. IEEE, 2022.

[48] Parvin Taheri, Danushka Menikkumbura, Erico Vanini, Sonia Fahmy, Patrick Eugster, and Tom Edsall. Rocc: robust congestion control for rdma. In *Proceedings of the 16th International conference on emerging networking experiments and technologies*, CoNEXT '20, pages 17–30, New York, NY, USA, 2020. Association for Computing Machinery.

[49] FS.com UK. Fs hpc data centrenetwork solution: Infiniband vs. roce solution. https://www.linkedin.com/pulse/h100-ai-data-centre-network-selection-\guide-infiniband-vs-roce-5yjnc.

[50] Zirui Wan, Jiao Zhang, Haoran Wei, Zhuo Jiang, Xiaolong Zhong, Wenfei Wu, Huaping Zhou, Tian Pan, and Tao Huang. Recc: Joint congestion control based on rtt and ecn for high-speed rdma networks. *Proceedings of the ACM on Networking*, 2(CoNEXT4):1–18, 2024.

[51] Zirui Wan, Jiao Zhang, Mingxuan Yu, Junwei Liu, Jun Yao, Xinghua Zhao, and Tao Huang. Bicc: Bilateral congestion control in cross-datacenter rdma networks. In *IEEE INFOCOM 2024-IEEE Conference on Computer Communications*, pages 1381–1390. IEEE, 2024.

[52] Weitao Wang, Masoud Moshref, Yuliang Li, Gautam Kumar, TS Eugene Ng, Neal Cardwell, and Nandita Dukkipati. Poseidon: Efficient, robust, and practical datacenter CC via deployable INT. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 255–274, Boston, MA, 2023. USENIX Association.

[53] Zilong Wang, Layong Luo, Qingsong Ning, Chaoliang Zeng, Wenxue Li, Xinchen Wan, Peng Xie, Tao Feng, Ke Cheng, Xiongfei Geng, et al. Srnic: A scalable architecture for rdma nics. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 1–14, 2023.

[54] Zihao Wei, Dezun Dong, Shan Huang, and Liquan Xiao. Expresspass+: Ecn-friendly credit reservation congestion control for datacenters. In *Proceedings of the ACM*

SIGCOMM 2019 Conference Posters and Demos, SIGCOMM Posters and Demos '19, page 169–171, New York, NY, USA, 2019. Association for Computing Machinery.

[55] Jaichen Xue, Muhammad Usama Chaudhry, Balajee Vamanan, TN Vijaykumar, and Mithuna Thottethodi. Fast congestion control in rdma-based datacenter networks. In *Proceedings of the ACM SIGCOMM 2018 Conference on Posters and Demos*, pages 24–26, 2018.

[56] Dingyu Yan, Yaping Liu, Shuo Zhang, Binxing Fang, Feng Zhao, and Zhikai Yang. Pcnp: A rocev2 congestion control using precise cnp. *Computer Networks*, 247:110453, 2024.

[57] Siyu Yan, Xiaoliang Wang, Xiaolong Zheng, Yinben Xia, Derui Liu, and Weishan Deng. Acc: Automatic ecn tuning for high-speed datacenter networks. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM '21, pages 384–397, New York, NY, USA, 2021. Association for Computing Machinery.

[58] Jiao Zhang, Yali Zhang, Zixuan Guan, Zirui Wan, Yinben Xia, Tian Pan, Tao Huang, Dezhi Tang, and Yun Lin. Hiercc: Hierarchical rdma congestion control. In *Proceedings of the 5th Asia-Pacific Workshop on Networking*, APNet '21, pages 29–36, New York, NY, USA, 2022. Association for Computing Machinery.

[59] Yiran Zhang, Qingkai Meng, Chaolei Hu, and Fengyuan Ren. Revisiting congestion control for lossless ethernet. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 131–148, 2024.

[60] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. Congestion control for large-scale rdma deployments. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, pages 523–536, New York, NY, USA, 2015. Association for Computing Machinery.

[61] Yibo Zhu, Monia Ghobadi, Vishal Misra, and Jitendra Padhye. Ecn or delay: Lessons learnt from analysis of dcqcn and timely. In *Proceedings of the 12th International on Conference on emerging Networking EXperiments and Technologies*, CoNEXT '16, pages 313–327, New York, NY, USA, 2016. Association for Computing Machinery.

Appendix

A Extended Evaluation

A.1 Qualitative Comparison Against other CC Protocols

We provide a qualitative analysis to compare against other CC protocols. Swift [28] relies on per-packet RTT measurements for congestion control. However, standard RDMA-packet does not carry RTT information, necessitating individual RTT probe packets. In high-speed AI networks, this requirement can result in significant overhead, with probe packet rates reaching up to 12.5 MPPS. Additionally, because AI training networks adopt a multi-tier ToR network topology, it becomes challenging to establish an appropriate RTT baseline for flows with different path lengths, which can lead to inefficient network utilization (as shown in Figure 1). Protocols like PowerTCP [1] and HPCC [31] rely on INT-metadata carried in per-packet ACKs, which are incompatible with the current ROCEv2 UDP packet format. Moreover, RNICs do not support per-packet ACKs and lack the capacity to handle the enormous volume of ACK packets and INT-metadata.

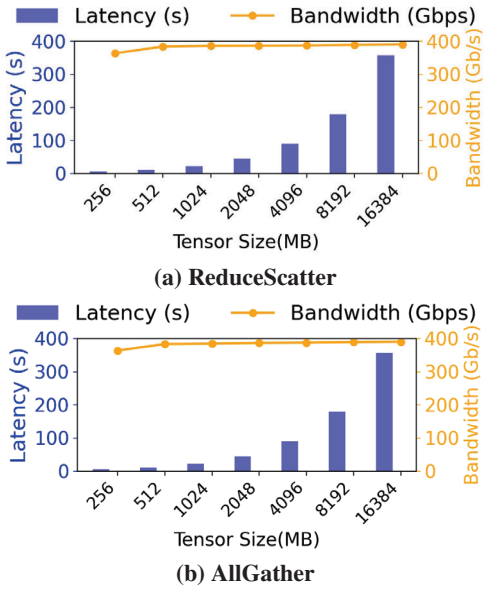
Therefore, we compare with the most widely used DCQCN and IB protocols. It is notable that our experiments are conducted on the real-world system, as simulation results lack practical significance due to their omission of hardware capabilities.

A.2 The Performance of Other Collectives

In practice, ring-based collective communications, e.g. AllReduce, AllGather, and ReduceScatter, rarely cause congestion compared to AlltoAll operations. We evaluated ReduceScatter and AllGather operations in a 64-GPU, 400 Gbps CLOS cluster with NVLink disabled to enforce RDMA routing, the results are shown in Figure 14. The x-axis indicates the message size of the tensor involved in the collective communication. The blue bar represents the total communication delay, and the orange line represents the average NIC throughput. By testing tensor sizes from 256 MB to 16,384 MB, we observed that the average bandwidth rapidly increased to 390.35 Gb/s, achieving near-full bandwidth utilization. Although congestion hardly triggers in these collectives, it can occur during traffic transitions, such as switching from AllReduce to AlltoAll, where Barre demonstrates good performance.

A.3 The Performance of Extreme Incast Scenarios

We evaluated Barre's performance under extreme incast scenarios, with the results presented in Figure 15. We gradually increase the number of incast flows from 16 to 4096, and



(a) ReduceScatter

(b) AllGather

Figure 14: Performance of NCCL ReduceScatter and AllGather in 64-GPU cluster

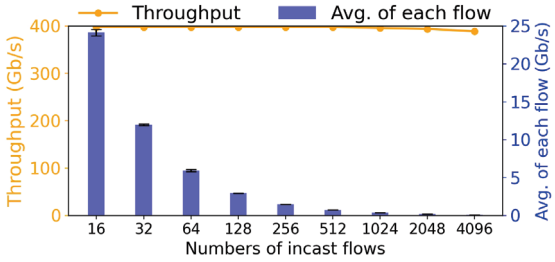


Figure 15: Performance of extreme incast scenarios

record throughput of each flows. The x-axis denotes the number of flows participating in the incast. The orange line represents the total network throughput, the blue bars indicate the average throughput per flow, and the black error bars depict the difference in throughput across each flows. The results demonstrate that as the number of incast flows grows, the total throughput remains consistently within the range of 390–400 Gbps. Furthermore, the stdev of the average throughput across all flows rapidly decreases from 0.35 to 0.002. The results indicate that even under large-scale extreme incast, Barre is capable of maintaining relative fairness and stability while achieving high throughput.

B Model Analysis and Derivation

We utilize the classical fluid model to analyze the stability and steady state of Barre. The parameters and variables used in the fluid model are detailed in Table 3. In this model, for every $D(t)$R increases by α such that $R = R + \alpha$. Conversely, for every Congestion Notification Packet (CNP) event, the sending rate R is adjusted by a multiplicative factor β, resulting in $R = R \cdot \beta$. The fluid model is represented by Equations (1)-(5).

Table 3: Symbols in Barre fluid model

Symbol	Explanation
α	Increase parameter
β	Decrease parameter
C	Link capacity
U	Maximum Transmission Unit (MTU)
d	Baremetal round-trip delay
K_{min}, K_{max}	ECN parameter
P_{max}	ECN parameter
$D(t)$	Real-time Round Trip Delay
$R(t)$	Sending Rate of current CC session (per-QP, per-DestIP)
N	Number of flows over the bottleneck link
n	Number of CNPs in one RTT
$q(t)$	Real-time queue size
$p(t)$	ECN marking probability

$$\frac{\mathrm{d}R}{\mathrm{d}t} = \frac{\alpha}{D(t)} - \frac{\left(1-\beta^{p(t)R(t)D(t)/U}\right)R(t)}{D(t)} \tag{1}$$

$$n(t) = p(t)\frac{CD(t)}{U}) \tag{2}$$

$$p(t) = \frac{q(t)-K_{\min}}{K_{\max}-K_{\min}}P_{\max} \tag{3}$$

$$\frac{\mathrm{d}q}{\mathrm{d}t} = NR(t) - C \tag{4}$$

$$D(t) = d + \frac{q(t)}{C} \tag{5}$$

B.1 Stable Point Analyses

The stable point of the system is defined as the solution of the equations $\frac{\mathrm{d}R}{\mathrm{d}t}=0$ and $\frac{\mathrm{d}q}{\mathrm{d}t}=0$, denoted by ($R^*$, q^*, p^*, n^*). From this condition, we can get:

$$n^* = P^*\frac{CD^*}{U}$$

$$R^* = \frac{C}{N}$$

Combined with Equation (2) (3) and (5), n^* could be translated in to Equation (6):

$$\begin{aligned} n^* &= \frac{N\ln\left(1-\frac{\alpha N}{C}\right)}{\ln\beta} \\ &= (q-K_{\min})(q+Cd)\frac{P_{\max}}{(K_{\max}-K_{\min})U} \end{aligned} \tag{6}$$

Thus, we obtain a quadratic equation in terms of q:

$$q^2 + (Cd-K_{\min})q - K_{\min}Cd - \frac{N\ln\left(1-\frac{\alpha N}{C}\right)(K_{\max}-K_{\min})U}{P_{\max}\ln\beta} = 0$$

And the discriminant of this equation in terms of q is:

$$\Delta = (Cd-K_{\min})^2 + 4\left(K_{\min}Cd + \frac{N\ln\left(1-\frac{\alpha N}{C}\right)(K_{\max}-K_{\min})U}{P_{\max}\ln\beta}\right)$$

Since the rate increment of all flows within one RTT will definitely be less than C, $\ln\left(1-\frac{\alpha N}{C}\right)<0$. Because of $0<\beta<1$, we can get $\delta>0$ and $\sqrt{\Delta} > Cd - K_{\min}$, so the solution $q_1>0$, as shown in Equation (7).

$$q_1 = \frac{-(Cd-K_{\min})+\sqrt{\Delta}}{2} \tag{7}$$

Therefore, the stable point for the queue length $q^* = q$, and the stable point p^* for the ECN marking probability is:

$$p^* = \frac{q^*-K_{\min}}{K_{\max}-K_{\min}}$$

In conclusion, we can derive that a stable point must exist, and it is given by (R^*, q^*, p^*, n^*).

B.2 Stability Analyses

Near the stable point, the feedback delay of the control system is assumed to be constant D^*. The right halves of Equations (1) and (4) are represented by functions $f()$ and $g()$ respectively as follows:

$$f(R,p,q) = \frac{\alpha}{D(t)} - \frac{\left(1-\beta^{p(t)R(t)D(t)/U}\right)R(t)}{D(t)} \tag{8}$$

$$g(R) = NR(t) - C \tag{9}$$

The partial derivatives near the equilibrium point (R^*, q^*, p^*) are:

$$\begin{aligned} \left.\frac{\partial f}{\partial R}\right|_{\substack{R=R^*\\q=q^*\\p=p^*}} &= \left.-\frac{1-\beta^{p(t)R(t)D(t)/U}\left(\frac{p(t)R(t)D(t)}{U}\ln\beta+1\right)}{D(t)}\right|_{\substack{R=R^*\\q=q^*\\p=p^*}} \\ &= -\frac{1-\beta^{n^*/N}(n^*\ln\beta/N+1)}{D^*} \end{aligned}$$

$$\begin{aligned} \left.\frac{\partial f}{\partial p}\right|_{\substack{R=R^*\\q=q^*\\p=p^*}} &= \left.\frac{R(t)}{D(t)}\beta^{p(t)R(t)D(t)/U}\ln\beta\cdot\frac{R(t)D(t)}{U}\right|_{\substack{R=R^*\\q=q^*\\p=p^*}} \\ &= \frac{(R^*)^2\ln\beta}{U}\beta^{n^*/N} \end{aligned}$$

$$\begin{aligned} \left.\frac{\partial f}{\partial q}\right|_{\substack{R=R^*\\q=q^*\\p=p^*}} &= -\frac{\alpha}{CD(t)^2} + \frac{R(t)}{CD(t)^2}\left(1-\beta^{p(t)R(t)D(t)/U}\left(1-\frac{p(t)R(t)D(t)}{U}\ln\beta\right)\right) \\ &= -\frac{\alpha}{CD^{*2}} + \frac{R^*}{CD^{*2}}\left(1-\beta^{n^*/N}\left(1-\frac{n^*}{N}\ln\beta\right)\right) \\ &= -\frac{\alpha}{CD^{*2}} + \frac{R^*}{CD^{*2}}\left(1-\beta^{n^*/N}\left(1-\ln\left(1-\frac{\alpha}{R^*}\right)\right)\right) \end{aligned}$$

$$\left.\frac{\partial g}{\partial R}\right|_{R=R^*} = N$$

According to the analysis at the stable point, it can be known that $-\frac{\alpha}{CD^{*2}} + \frac{R^*}{CD^{*2}}(1-\beta^{n^*/N}) = 0$. Therefore, the partial derivative of f with respect to q can be expressed as

$\frac{\partial f}{\partial q}\Big|_{\substack{R=R^*\\q=q^*\\p=p^*}} = \frac{R^*}{CD^{*2}}\beta^{n^*/N}\ln(1-\frac{\alpha}{R^*})$. So, the condition for the partial derivative of f with respect to q to be zero is restricted by whether $\ln(1-\frac{\alpha}{R^*})$ is 0. Only when $R^* >> \alpha$, can $\frac{\partial f}{\partial q}$ be approximated as zero.

Condition 1: $\frac{\partial f}{\partial q}\Big|_{\substack{R=R^*\\q=q^*\\p=p^*}} \neq 0$**.** At the equilibrium point (R^*, q^*, p^*), find the partial derivatives of $f()$ with respect to r, p, q. Then apply the Taylor series expansion near the equilibrium point to obtain:

$$\begin{aligned}\frac{d\delta R}{dt} \approx &-\frac{1-\beta^{n^*/N}\left(\frac{n^*\ln\beta}{N}+1\right)}{D^*}\delta R(t)\\ &+\frac{(R^*)^2\ln\beta}{U}\beta^{n^*/N}\delta p(t-D^*)\\ &+\left(\frac{R^*}{CD^{*2}}\beta^{n^*/N}\ln\left(1-\frac{\alpha}{R^*}\right)\right)\delta q(t)\end{aligned} \tag{10}$$

$$\frac{d\delta q}{dt} \approx N\delta R(t) \tag{11}$$

In equations (10) and (11), $\delta R(t) = R(t) - R^*$, and $\delta p(t) = p(t) - p^*$. Applying the Laplace transform to both sides of the equation, we could get:

$$\begin{aligned}s\tilde{R}(s) - \delta R(0) = &-\frac{1-\beta^{n^*/N}\left(\frac{n^*\ln\beta}{N}+1\right)}{D^*}\tilde{R}(s)\\ &+\frac{(R^*)^2\ln\beta}{U}\beta^{n^*/N}\tilde{p}(s)e^{-sD^*}\\ &+\left(\frac{R^*}{CD^{*2}}\beta^{n^*/N}\ln\left(1-\frac{\alpha}{R^*}\right)\right)\tilde{q}(s)\end{aligned}$$

$$s\tilde{q}(s) - \delta q(0) = N\tilde{R}(s)$$

Here, $\tilde{R}(s)$ and $\tilde{q}(s)$ represent the Laplace transforms of $\delta R(t)$ and $\delta q(t)$, respectively. The transfer function from $\tilde{p}(s)$ to $\tilde{R}(s)$ is:

$$G_{p\to R}(s) = \frac{\frac{(R^*)^2\ln\beta}{U}\beta^{n^*/N}}{s+\frac{1-\beta^{n^*/N}\left(\frac{n^*\ln\beta}{N}+1\right)}{D^*}-\frac{N}{s}\left(\frac{R^*}{CD^{*2}}\beta^{n^*/N}\ln\left(1-\frac{\alpha}{R^*}\right)\right)}e^{-sD^*}$$

The transforms of $\tilde{R}(s)$ to $\tilde{q}(s)$ is $G_{R\to q}(s) = \frac{N}{s}$. Similarly, we define $\delta p(t) = p(t) - p^*$. Based on Equation (4), $\delta p(t)$ could be translated into:

$$\delta p(t) = p(t) - p^* \quad = \frac{\delta q(t)}{K_{\max} - K_{\min}}P_{\max}$$

The transforms of $\tilde{q}(s)$ to $\tilde{p}(s)$ is :

$$G_{q\to p}(s) = \frac{\tilde{p}(s)}{\tilde{q}(s)} = \frac{P_{\max}}{K_{\max} - K_{\min}}$$

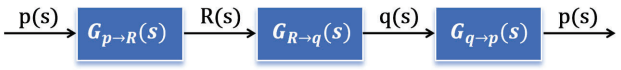


Figure 16: The equivalent open-loop system

Thus, the transfer function for the closed-loop system is:

$$G(s) = G_{p\to R}(s)G_{R\to q}(s)G_{q\to p}(s) = \frac{a}{s(s+b+\frac{c}{s})}e^{-sD^*} \tag{12}$$

In Equation (12), a, b and c can be expressed as:

$$a = \frac{(R^*)^2\ln\beta}{U}\beta^{n^*/N}N\frac{P_{max}}{K_{max}-K_{min}}$$

$$b = \frac{1-\beta^{n^*/N}(n^*\ln\beta/N+1)}{D^*}$$

$$c = -N(\frac{R^*}{CD^{*2}}\beta^{n^*/N}\ln(1-\frac{\alpha}{R^*}))$$

The equivalent open-loop system to the closed-loop system is shown in Figure 16. The frequency response of the system is denoted by $G(j\omega)$, with the magnitude and phase angle denoted by $r(\omega) = |G(j\omega)|$ and $\theta(\omega) = \angle G(j\omega)$, respectively. Thus, we have:

$$r(\omega) = \frac{a}{\sqrt{b^2\omega^2 + (c-w^2)^2}}$$

$$\theta(\omega) = -\omega D^* - \arctan\frac{b\omega}{c-w^2}$$

Let ω_g denote the 0-dB crossover frequency on the Bode plot, where $r(\omega_g) = 1$. In accordance with the Bode stability criterion to find the stability conditions, if within the frequency range $(0, \omega_g)$, the phase angle $\theta(\omega)$ is greater than $-\pi$, then the phase margin is greater than 0, ensuring the system is stable. Since $\theta(\omega)$ is monotonically decreasing and given that $\theta(0) = -\frac{\pi}{2}$, the stability condition is $\theta(\omega_g) > -\pi$, as $-\omega_g D^* - \arctan\frac{b\omega_g}{c-w_g^2} > -\pi$. The stability condition is:

$$-\omega_g D^* - \arctan\frac{b\omega_g}{c-w_g^2} > -\pi \tag{13}$$

Condition 2: $\frac{\partial f}{\partial q}\Big|_{\substack{R=R^*\\q=q^*\\p=p^*}} = 0$**.** Given that parameter β is set close to 1 in the Barre algorithm, $\ln\beta$ can be approximated as 0, thus $\frac{\partial f}{\partial q}\Big|_{\substack{R=R^*\\q=q^*\\p=p^*}} \approx 0$. Performing a Taylor series expansion around the equilibrium point yields the following:

$$\frac{d\delta R}{dt} \approx -\frac{1-\beta^{n^*/N}\left(\frac{n^*\ln\beta}{N}+1\right)}{D^*}\delta R(t) + \frac{(R^*)^2\ln\beta}{U}\beta^{n^*/N}\delta p(t) \tag{14}$$

$$\frac{d\delta q}{dt} \approx N\delta R(t) \tag{15}$$

In Equation (14) and (15), $\delta R(t) = R(t) - R^*$, and $\delta p(t) = p(t) - p^*$. Applying the Laplace transform to both sides of the equation, we could get:

$$s\tilde{R}(s) - \delta R(0) = -\frac{1-\beta^{n^*/N}\left(\frac{n^*\ln\beta}{N}+1\right)}{D^*}\tilde{R}(s) + \frac{(R^*)^2\ln\beta}{U}\beta^{n^*/N}\tilde{p}(s)$$

$$s\tilde{q}(s) - \delta q(0) = N\tilde{R}(s)$$

Here, $\tilde{R}(s)$ and $\tilde{q}(s)$ represent the Laplace transforms of $\delta R(t)$ and $\delta q(t)$, respectively. The transfer function from $\tilde{p}(s)$ to $\tilde{R}(s)$ is:

$$G_{p\to R}(s) = \frac{\frac{(R^*)^2\ln\beta}{U}\beta^{n^*/N}}{s + \frac{1-\beta^{n^*/N}\left(\frac{n^*\ln\beta}{N}+1\right)}{D^*}}$$

The transforms of $\tilde{R}(s)$ to $\tilde{q}(s)$ is $G_{R\to q}(s) = \frac{N}{s}$. Similarly, we define $\delta p(t) = p(t) - p^*$. Based on Equation (4), $\delta p(t)$ could be translated into:

$$\delta p(t) = p(t) - p^* \quad = \frac{\delta q(t)}{K_{\max} - K_{\min}} P_{\max}$$

The transforms of $\tilde{q}(s)$ to $\tilde{p}(s)$ is :

$$G_{q\to p}(s) = \frac{\tilde{p}(s)}{\tilde{q}(s)} = \frac{P_{\max}}{K_{\max} - K_{\min}}$$

Thus, the transfer function for the closed-loop system is:

$$G(s) = G_{p\to R}(s)G_{R\to q}(s)G_{q\to p}(s) = \frac{a}{s(s+b)} \tag{16}$$

In Equation (16), a and b can be expressed as:

$$a = \frac{(R^*)^2\ln\beta}{U}\beta^{n^*/N}N\frac{P_{max}}{K_{max} - K_{min}}$$

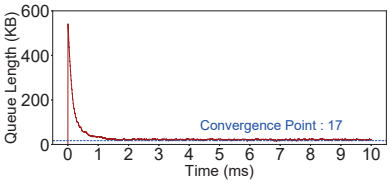
$$b = \frac{1-\beta^{n^*/N}(n^*\ln\beta/N+1)}{D^*}$$

The equivalent open-loop system to the closed-loop system is the same as in Figure 16. The frequency response of the system is denoted by $G(j\omega)$, with the magnitude and phase angle denoted by $r(\omega) = |G(j\omega)|$ and $\theta(\omega) = \angle G(j\omega)$, respectively. Thus, we have:

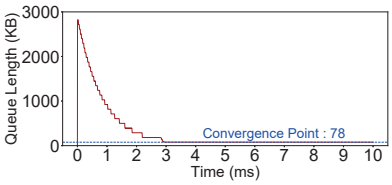
$$r(\omega) = \frac{a}{\sqrt{\omega^2 + b^2}}\ ,\ \theta(\omega) = -\frac{\pi}{2} - \arctan\frac{\omega}{b}$$

Let ω_g denote the 0-dB crossover frequency on the Bode plot, where $r(\omega_g) = 1$. In accordance with the Bode stability criterion to find the stability conditions, if within the frequency range $(0, \omega_g)$, the phase angle $\theta(\omega)$ is greater than $-\pi$, then the phase margin is greater than 0, ensuring the system is stable. Since $\theta(\omega)$ is monotonically decreasing and given that $\theta(0) = -\frac{\pi}{2}$, the stability condition is $\theta(\omega_g) > -\pi$, as $-\frac{\pi}{2} - \arctan\frac{\omega_g}{b} > -\pi$. The stability condition is:

$$\arctan\frac{\omega_g}{b} < \frac{\pi}{2} \tag{17}$$



(a) 20:1 Incast



(b) 100:1 Incast

Figure 17: Convergence of Barre's fluid model

B.3 Simulation Verification

We utilize NS3 simulations to evaluate the convergence of Barre. The network topology consists of 10 servers connected to a single ToR switch, with each server equipped with a 100Gbps NIC linked to the switch. The baseline RTT is set to 2us. The increase factor and decrease factor for CC algorithm are configured at 10Mbps and 0.99, respectively. For ECN threshold, we configure K_{min}=10KB, K_{max}=200KB, and P_{max}=1 KB. We test different scale of AlltoAll traffic for 20:1 Incast (Figure 17 (a)) and 100:1 Incast (Figure 17 (b)).

In Figure 17 (a), 20 flows start at the same time, and keep going until each flow finished sending 100MB message. The simulation result shows that, the switch queue length of the congestion point is bounded by 17 KB when the flows are converged at approximately 1.5 ms. We use can get theoretical steady state q given by Equation (7) in fluid model analysis is 9.6 KB. Similarly, Figure 17 (b) shows the simulation result of 100 Incast flows, as each flow sends 10 MB message. The switch queue length converged to 79 KB at about 3ms. And the computational steady point is 78.27KB.

The simulation results and theoretical analysis demonstrate a consistent convergence point, validating the accuracy of Barre's convergence analysis. More importantly, Barre exhibits robust convergence properties even with large-scale incast traffic. This consistency underscores the reliability of Barre's design principles and its effectiveness in maintaining stable performance under dynamic traffic conditions.

FLB: Fine-grained Load Balancing for Lossless Datacenter Networks

Jinbin Hu[1,2,3*], Wenxue Li[2*], Xiangzhou Liu[2], Junfeng Wang[2], Bowen Liu[2], Ping Yin[4],
Jianxin Wang[1], Jiawei Huang[1], Kai Chen[2]
[1]*Central South University*
[2]*iSING Lab, Hong Kong University of Science and Technology*
[3]*Changsha University of Science and Technology*
[4]*Inspur*

Abstract

Remote Direct Memory Access (RDMA) over Converged Ethernet (RoCE) cooperating with Priority Flow Control (PFC) has been widely deployed in production datacenters to enable low latency, lossless transmission. At the same time, modern datacenters typically offer parallel transmission paths between any pair of end-hosts, underscoring the importance of load balancing. However, the well-studied load balancing mechanisms designed for lossy datacenter networks (DCNs) are ill-suited for such lossless environments.

Through extensive experiments, we are among the first to comprehensively inspect the interactions between PFC and load balancing, and uncover that existing fine-grained rerouting schemes can be *counterproductive* to spread the congested flows among more paths, further aggravating PFC's head-of-line (HoL) blocking. Motivated by this, we present FLB, a *F*ine-grained *L*oad *B*alancing scheme for lossless DCNs. At its core, FLB employs threshold-free rerouting to effectively balance traffic load and improve link utilization during normal conditions and leverages timely congested flow isolation to eliminate HoL blocking on non-congested flows when congestion occurs. We have fully implemented a FLB prototype, and our evaluation results show that FLB reduces PFC PAUSE rate by up to 96% and avoids HoL blocking, translating to up to 45% improvement in goodput over CONGA+DCQCN and 40%, 36%, 29% and 18% reduction in average flow completion time (FCT) over LetFlow+Swift, MP-RDMA, Proteus+DCQCN and LetFlow+PCN, respectively.

1 Introduction

Lossless network switching fabric is crucial and increasingly deployed in Converged Enhanced Ethernet (CEE) datacenters [1–15], e.g., Microsoft Azure [16], Alibaba Cloud [17] and Google cloud [18]. Remote Direct Memory Access (RDMA) over Converged Ethernet (RoCE) [4, 5, 19] is a widely deployed transport protocol in CEE datacenters. To guarantee lossless transmission, CEE employs priority flow control (PFC) [20] to prevent buffer overflow.[1] However, PFC PAUSE/RESUME mechanisms may lead to HoL blocking and congestion spreading [2, 4, 21–25], etc., causing severe performance degradation up to 10× for individual flows [2]. Worse still, a local PFC pausing may propagate backward hop by hop and eventually to the sources, turning into a global network paralysis! At the same time, load balancing (i.e., transmitting packets via multiple paths) is crucial because relying on a single path cannot fully leverage the rich parallel paths available in modern datacenters [7, 26–30].

In this paper, we point out that despite the great effort, the well-studied load balancing schemes [28, 29, 31–39] designed for lossy DCNs are ill-suited for PFC-enabled lossless DCNs for reasons experimentally demonstrated in §2.

On one hand, existing load balancing (LB) solutions cannot effectively split traffic among parallel paths in lossless DCN, even if they work well for the lossy network (§2.1). The reason is that in the current RDMA deployment, the forwarding rate is usually smoothed by a rate-shaper, flowlets rarely appear under the same fixed flowlet timeout as in lossy DCN to ensure no out-of-order packets [7]. Thus, the advanced flowlet-based load balancing schemes such as CONGA [28] and LetFlow [31] cannot flexibly reroute traffic in lossless DCN, resulting in unbalanced load and low link utilization.

On the other hand, fine-grained load balancing schemes (*e.g.*, flowlet-based [28, 31–33], flowcell-based [34], packet-level [7, 29, 35, 36, 40]) make the HoL blocking and congestion spreading more severe under PFC-triggered scenarios compared to single-path approaches (§2.2). Specifically, once PFC is triggered, if the congested flows that really responsible for congestion are sprayed across multiple paths, the number of PFC paused ports increases significantly, resulting in more blocked victim flows. In brief, the existing multipath schemes can be counterproductive in PFC-enabled lossless DCNs, as they spread the congested flows among more paths, aggravat-

[*] Equal contribution.

[1] With PFC, the downstream switch sends a PAUSE frame to its upstream switch once the ingress queue length exceeds the PFC threshold, and sends a RESUME frame when the queue drains below another threshold [4, 7].

ing the HoL blocking.

Furthermore, the above problems cannot be completely addressed even combined with sophisticated end-to-end congestion control protocols such as DCQCN [4], TIMELY [10], Swift [12], HPCC [11], and PCN [2] (§2.3). This is because current congestion control protocols cannot entirely prevent PFC from being triggered, particularly in bursty scenarios, thereby failing to avoid the counterproductive interaction between PFC and load balancing mechanisms.

Given the limitations of existing solutions, we ask: *can we design a load balancing scheme for PFC-enabled lossless DCNs that achieves high link utilization while eliminating PFC side effects?* This translates to three design goals:

- Flexibly rerouting the traffic to effectively balance load and enhance link utilization in normal conditions.
- Eliminating the HoL blocking and congestion spreading during congestion and PFC triggering.
- Reducing dependency on complex congestion control.

To this end, we present FLB, a PFC-aware fine-grained load balancing scheme for lossless DCNs that achieves all the above design goals (§3). First of all, to flexibly balance the load in normal conditions, FLB reroutes flows without a preset threshold, while still effectively adapting to varying packet intervals (§3.1). To avoid packet reordering, FLB reroutes the arrival packets from current slow path to another one only when the time interval between the arrival packet and the previous one in the same flow is larger than the delay difference of the two paths. Meanwhile, to ensure low latency, the path with minimum delay among all feasible paths is selected for rerouting.

Second, to mitigate the side effects of PFC during congestion, FLB leverages a traffic isolation scheme (§3.2) that consolidates, rather than spreads, the congested flows into a minimized set of isolation paths, while rerouting the uncongested flows through the remaining parallel paths. Specifically, when the egress queue exceeds a preset isolation threshold, a congestion notification message (CNM) is generated and sent to the source edge switch. Based on the CNM, the source edge switch identifies the congested flows and consolidates them from original paths into a minimized set of isolation paths. In this way, FLB maintains the efficiency of multipath transmission for uncongested flows while effectively eliminating HoL blocking and preventing congestion from spreading caused by congested flows.

Theoretically, FLB can be integrated with any end-host congestion control mechanisms. Moreover, by effectively balancing traffic and limiting the side effects of PFC, FLB reduces the need for complex congestion control algorithms. To validate this, we come up with a minimal end-host rate control (§3.3) where the sender starts transmission at line rate and pauses only when receiving a CNM. We show that even with this simple rate control, FLB achieves superior performance compared to other combinations of load balancing and congestion control (§4.3).

We have implemented a FLB prototype (§3.4) using P4 programming language [41], and quantified the hardware resource usage of FLB. We evaluate FLB on a testbed including 20 Dell servers equipped with Mellanox ConnectX-5 100GbE Network Interface Cards (NICs) and 100Gbps programmable switches. The experimental results show that FLB reduces the average flow completion time (AFCT) by up to 30% compared to MP-RDMA under realistic datacenter workloads. FLB also improves link utilization by 78%, 144% and 28% in Web Search workload over LetFlow, LetFlow+DCQCN and MP-RDMA, respectively (§4.1). We also analyzed various design components of FLB and demonstrated their contributions to its superior performance (§4.2).

To complement the small-scale testbed experiments, we conducted large-scale NS3 simulations. The results show that FLB with minimal rate control (FLB+RC) outperforms state-of-the-art multipath transmission schemes, even when integrated with congestion controls, reducing the AFCT by up to 70%, 36% and 29% compared to LetFlow+DCQCN, MP-RDMA and Proteus+DCQCN, respectively (§4.3).

Finally, we note that there emerges a thread of research on lossy fabric without PFC [8, 42]. However, as reported in [1], lossless fabric is still the most-deployed infrastructure in production datacenters. FLB targets a practical solution for these lossless FPC-enabled datacenter fabrics.

2 Problem Demonstration

This section experimentally demonstrates the problems of existing fine-grained load balancing schemes (designed for lossy DCNs) in PFC-enabled lossless DCNs, motivating our design of FLB.

2.1 Inflexible rerouting leads to load imbalance and link under-utilization

As many prior studies have noted, flow-based load balancing, such as ECMP [43], is ineffective in distributing the load in modern datacenters [44, 45]. Therefore, spreading traffic across multiple paths with finer granularity is essential to achieving higher network utilization.

Both industry and academia have proposed various fine-grained load balancing mechanisms [7, 28, 29, 31–36]. However, these mechanisms are typically designed for lossy DCNs. State-of-the-art flowlet-based schemes, such as CONGA [28] and LetFlow [31], are also unable to flexibly reroute traffic in lossless DCNs due to the rare occurrence of flowlet gaps. This is because hardware-accelerated data transmission with rate shapers in RDMA networks minimizes packet intervals. Furthermore, RDMA bypasses the operating system, eliminating interruptions that might otherwise create gaps.

Small-scale Testbed Experiments. We investigate how the typical flowlet-based schemes work in Leaf-spine [28] topol-

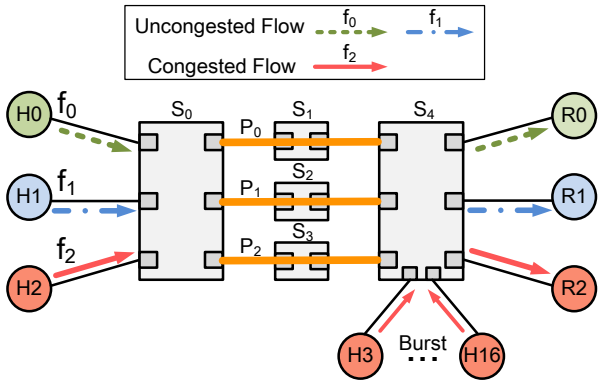


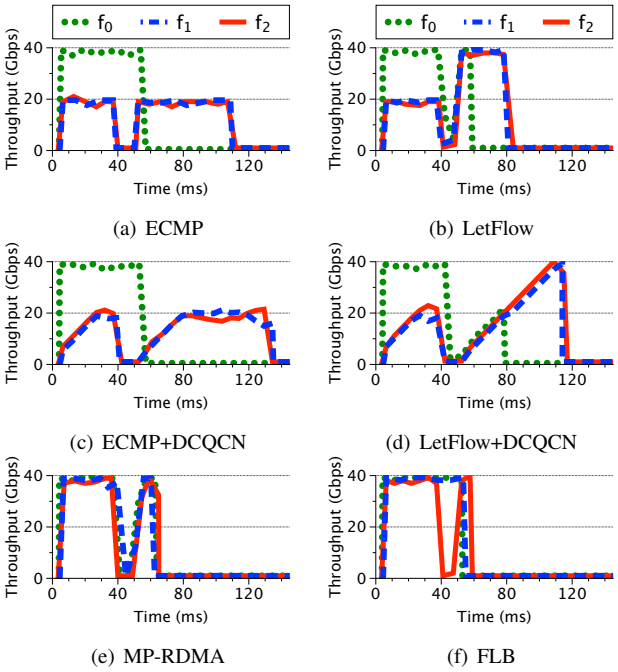
Figure 1: Typical network scenarios. Under flow-based LB such as ECMP, f_0 takes the path P_0 {S0, S1, S4}, f_1 and f_2 hash collide on the path P_2 {S0, S3, S4}. Under flowlet-based LB such as LetFlow, f_0, f_1 and f_2 are spread across three paths P_0 {S0, S1, S4}, P_1 {S0, S2, S4} and P_2 {S0, S3, S4}.

ogy as used in prior work [2,3]. As shown in Fig. 1, 3 senders (H0, H1, H2) and 3 receivers (R0, R1, R2) connect to the corresponding edge switches (S0, S4), respectively. There are 3 equal-cost paths between the two edge switches passing through core switches (S1, S2, S3) and represented by P_0 {S0, S1, S4}, P_1 {S0, S2, S4} and P_2 {S0, S3, S4}, respectively. In addition, 14 senders (H3∼H16) connect to the switch S4.

We conduct experiments to explore the issues of load balancing schemes including ECMP, LetFlow, ECMP+DCQCN, LetFlow+DCQCN, and MP-RDMA, which are implemented by using DPDK [46] and P4 hardware switch [41,47]. The testbed consists of 20 servers connected to two P4 switches with 3 parallel paths of 40Gbps. Each switch has 22MB shared buffer and 32 full duplex ports enabling PFC with dynamic threshold[2]. The flowlet timeout is set to 50μs. We set all parameters to the default values recommended in [4,7,31]. The bursty scenarios follow the prior work [2,3]. At time 0ms, H0, H1 and H2 start a long flow with 250MB to R0, R1 and R2, respectively, named f_0, f_1 and f_2. At time 40ms, each sender of H3∼H16 generates 40 short flows to R2 at line rate, and the size of each short flow is 64KB. These bursty flows last about 8ms.

At the beginning, we configure f_0 takes the path P_0 {S0, S1, S4}, while f_1 and f_2 share the path P_2 {S0, S3, S4} under both ECMP and LetFlow. The results are shown in Fig. 2. Since f_2 and all bursty short flows arrive at the same receiver R2, the egress port from S4 to R2 is congested during the duration of bursty flows. Once PFC is triggered at the ingress ports in S4, the PAUSE frames from these ingress ports pause their upstream ports. As previously configured, both f_1 and f_2 are routed to the same path P_2 under ECMP, LetFlow and MP-RDMA. An ideal load balancing scheme should timely switch f_1 to the idle path P_1. However, as shown in Fig. 2, none of them can flexibly distribute traffic. They fail to reroute

[2]We implement PFC based on the IEEE Standard 802.1Qbb [20].



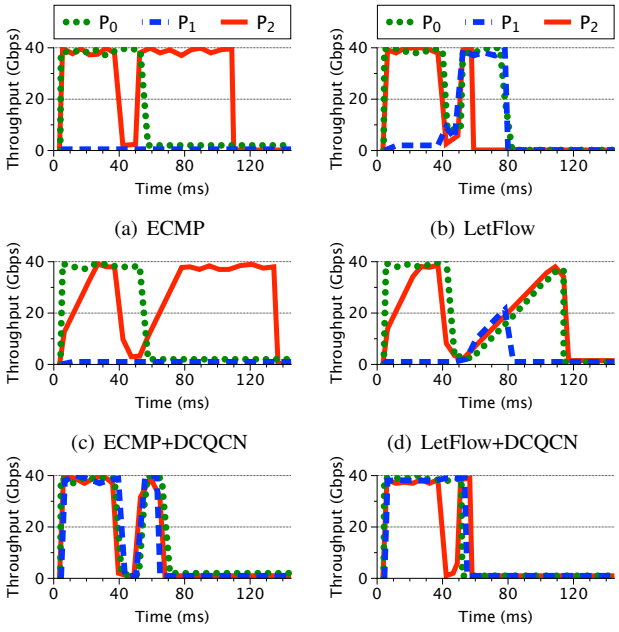
(a) ECMP (b) LetFlow (c) ECMP+DCQCN (d) LetFlow+DCQCN (e) MP-RDMA (f) FLB

Figure 2: Throughput of flows under various schemes. Under ECMP and ECMP+DCQCN, the uncongested flow f_1 is blocked. Under LetFlow, LetFlow+DCQCN and MP-RDMA, both the uncongested flows f_0 and f_1 are blocked. The blocked victim flows cannot be rerouted to other paths. There are no victim flows in FLB.

the uncongested flow f_1 from the congested path $P2$ to an uncongested path. This is because the flowlet-based switching cannot reroute packets unless flowlets emerge, which is prevented by inappropriate fixed timeouts for avoiding packet reordering. As a result, available link bandwidth is wasted.

2.2 Multi-path spreading expands the influence scope of PFC's HoL blocking

We repeat the above experiment in Fig. 1 but *manually* configured the congested flow f_2 to utilize all three paths (P_0, P_1, and P_2) under LetFlow and MP-RDMA. The throughput for these three paths is shown in Fig. 3, and we make the following observations: (1) single-path transmission: under the ECMP and ECMP+DCQCN schemes, since f_2 follows a single path, P_2, *only* P_2 *is paused by PFC*. As a result, only the uncongested flow f_1 on the same path is blocked, while flow f_0 on the other path remains unaffected by PFC's HoL blocking; (2) multi-path spreading: under LetFlow and MP-RDMA schemes, *all three paths* (P_0, P_1, *and* P_2) *are paused by PFC* during the bursty flows. This occurs because the congested flow f_2 is routed across all three paths, causing PFC PAUSE frames to be sent to all upstream ports of S4. This leads to more severe congestion spreading, where all paths (P_0 and P_1) and flows (f_0 and f_1) are blocked.



(a) ECMP (b) LetFlow

(c) ECMP+DCQCN (d) LetFlow+DCQCN

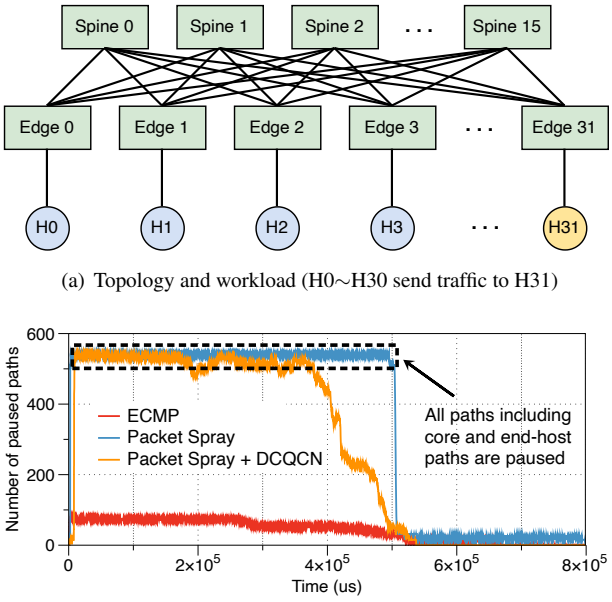
(e) MP-RDMA (f) FLB

Figure 3: Throughput of paths under various schemes. Under ECMP and ECMP+DCQCN, one path P_2 is paused by PFC due to the congested flow f_2 transmitting on P_2. Under LetFlow, LetFlow+DCQCN and MP-RDMA, three paths P_0, P_1 and P_2 are paused since the congested flow f_2 is spread on all three paths.

Large-scale Simulations. To further show the problem, we construct a larger-scale Clos topology with 16 spine and 32 leaf switches using NS3 (Fig. 4(a)). The switch configuration is the same as in the preceding testbed experiments. The end-hosts H0∼H30 simultaneously transmit 200MB of traffic to H31, creating a long-living 31-to-1 incast pattern.

We investigate the extent of congestion spreading during multi-path transmission (using packet spraying as an example) in lossless DCNs. We compare the number of paused paths (*i.e.*, paths receiving PFC PAUSE frames during a given time interval) under packet spraying and ECMP. The results are presented in Fig. 4(b). As the results show, ECMP causes only the passing paths of 31 flows to pause (about 70 paths in total). In contrast, packet spraying results in all paths (about 340 paths) being paused. This is because data is sprayed across all paths, causing the PFC PAUSE frames to back-propagate throughout the network hop by hop.

In summary, with fine-grained rerouting, packets from congested flows are counterproductive spread across multiple paths, causing more paths to be paused by PFC, which aggravates to more victim flows and reduced link utilization. Therefore, an ideal load balancing scheme should prevent congested flows from using multiple paths and minimize the number of paused paths.



(a) Topology and workload (H0∼H30 send traffic to H31)

(b) The number of paused paths of three configurations.

Figure 4: Difference in congestion spreading between packet spraying and ECMP under large-scale topology.

2.3 Congestion control does not help

We envision that the above problems cannot be solved with congestion control as it cannot entirely prevent PFC from being triggered. To illustrate this issue, we observe again the results in Fig. 2 and Fig. 3. The performance is even worse when the existing LB schemes work with congestion control (*e.g.*, LetFlow+DCQCN). Although the congestion control effectively regulates the rates of f_1 and f_2 before 40ms, PFC is still triggered when the bursty flows start, because each bursty flow lasts less than 1 RTT and cannot be controlled by DCQCN. In this case, after the uncontrollable bursty flows finish transmissions, the remaining flows cannot utilize the available bandwidth immediately due to the slow evolution-based rate increasing of end-to-end congestion control. For MP-RDMA, the throughput of flows f_0, f_1 and f_2 are further reduced by congestion control due to the slow convergence.

We also evaluate the impact of end-host congestion controls (using DCQCN as an example) on the congestion spreading caused by packet spraying. As shown in Fig. 4(b), DCQCN only slowly reduces the number of paused paths, it cannot quickly eliminate congestion and stop the PFC pausing.

3 FLB

Building on the preceding observations and analysis, we pose the question: Can we design a fine-grained load balancing scheme for PFC-enabled lossless DCNs that achieves (1) efficient rerouting with high link utilization, (2) elimination of PFC side effects such as congestion spreading and aggravated

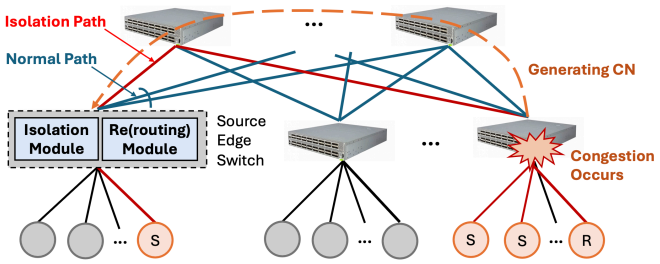


Figure 5: FLB overview. S, R and CN stand for sender, receiver and congestion notification, respectively.

HoL blocking, and (3) reduced reliance on congestion control mechanisms? We address this question with FLB.

The design of FLB mainly consists of two modules, as illustrated in Fig. 5. In normal conditions, FLB employs a flexible rerouting scheme (*i.e.*, the Rerouting Module) to effectively balance the traffic across multiple paths without causing packet reordering. Upon congestion, FLB quickly consolidates congested flows from their original multi-spread paths onto a minimized set of paths (*i.e.*, the Isolation Module), thereby eliminating the impact of PFC's HoL blocking on non-congested flows.

FLB can be integrated with any end-host congestion control schemes, including the existing lossless DCN congestion control algorithms [2, 4, 10, 48]. Moreover, since well-balanced traffic reduces congestion and congested flows are isolated, FLB minimizes the reliance on complex congestion control. To validate this, we design a minimal rate control at end hosts (§3.3), with a simplified algorithm that the senders start transmission at line rate and pause only when notified.

3.1 Flexible Rerouting

FLB employs flexible rerouting without the preset threshold to effectively balance the uncongested traffic and improve link utilization. Instead of switching with coarse granularity or passively waiting for emergence of flowlets, FLB works actively at the packet granularity. When making rerouting decisions, FLB ensures that the subsequent packet with larger sequence number arrives at the receiver later than the previous packets with smaller sequence number in the same flow to avoid reordering.

Algorithm 1 shows the rerouting logic of FLB. When a new active flow arrives, it selects a forwarding path with the minimum delay except the isolated path (lines 3-5). For the packets of existing flows, FLB compares the time interval between the arrival and previous packets in the same flow with the delay difference between the current path and other parallel paths, and selects the fastest path with the delay difference less than the time gap between two packets in order to avoid packet reordering (lines 8-12). Otherwise, the packet remains on its current path (lines 13-14).

Since FLB leverages the path delay differences to make forwarding decisions, the measured one-way delay can be used even if the clock is not synchronized [49]. Specifically, the source edge switch periodically samples packets, embedding their departure timestamps in the packet headers. When these marked packets arrive at the destination edge switch, the system computes precise one-way delay by subtracting the embedded timestamp from the arrival time. This delay measurement is then piggybacked on subsequent packets traversing the same port pair [50], enabling continuous path delay monitoring at the source edge switch without additional overhead. To address the issue of clock asynchrony, FLB calculates the real one-way queueing delay by subtracting the base delay from the measured one-way delay. The base delay is the one-way delay without any queueing delay and is obtained by recording the minimum history delay [50]. By default, FLB measures one-way delay at the edge switches periodically with two base RTT to reduce overhead.

Algorithm 1: Rerouting without Reordering

Input:
t_{OWD}: Measured one-way delay of a path;
t_{prev}: The arrival time of previous packet;
t_{cur}: The arrival time of current packet;
Output: p^*: The new routing path;

1 **for** *every packet* **do**
2 Assume its corresponding flow is f;
3 **if** f *is a new flow* **then**
4 $\{P'\}$ = all paths except the isolated paths occupied by the congested flows;
5 $p^* = Argmin_{p \in \{P'\}}(p.t_{OWD})$;
6 **else**
7 Assume f's path is p;
8 $\Delta t = f.t_{cur} - f.t_{prev}$;
9 $\{P'\}$ = all paths with less delay than p and no threat of reordering;
10 /* $\forall p' \in \{P'\},\ 0 < p.t_{OWD} - p'.t_{OWD} < \Delta t$ */
11 **if** $\{P'\} \neq \varnothing$ **then**
12 $p^* = Argmin_{p' \in \{P'\}}(p'.t_{OWD})$;
13 **else**
14 $p^* = p$; /* Do not reroute */
15 **return** p^*

FLB guarantees a better path switching. A flow requires a better path when its original path becomes less efficient than other parallel paths due to congestion. In such cases, end-host PAUSE (if no congestion control is enabled) or congestion control rate adjustments will react accordingly. PFC PAUSE creates a sufficient time interval between consecutive packets, triggering rerouting to a better path. Similarly, congestion control adjustments reduce the sending rate, increasing the time interval between consecutive packets, which also enables rerouting. As a result, FLB can consistently switch to a better path whenever one is available.

FLB works at the finer granularity than flowlet . Instead of switching with coarse flow-level granularity or passively waiting for emergence of flowlets, FLB reroutes flows flexibly without the fixed threshold at the switch to reduce PFC triggering and improve link utilization.

Periodic removal of flow entries. We use timeouts to periodically remove inactive flow table entries. Specifically, flow entries that have not been hit for a certain period (*e.g.*, 1 ms) are removed by periodically checking the age bit of each flow [28, 51]. This logic also applies to the congested flow entries used in the isolation module (detailed in §3.2).

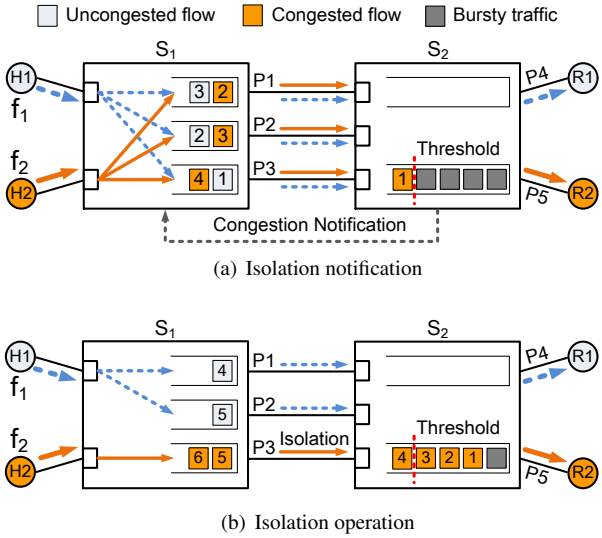
3.2 Flow Isolation

When congestion occurs, FLB aims to shield uncongested flows from being blocked by congested flows (i.e., HoL blocking). Specifically, FLB dynamically isolates congested flows onto a minimal set of paths, rather than all paths, to limit the impact of PFC and reduce buffer and link bandwidth wastage[3]. As a result, uncongested flows are no longer blocked.

Fig. 6 illustrates the flow isolation mechanism of FLB, where one priority class is assumed for each port. In Fig. 6(a), two flows f_1 and f_2 are transferred from source edge switches S1 to S2 via parallel links. With load balancing enabled, the packets of f_1 and f_2 share multiple egress ports of switch S1. Then, traffic bursts from other ingress ports to the egress port P5 at S2. On switch S2, when the queue length of egress port P5 reaches the isolation threshold due to the burst, a congestion notification message (CNM) with congestion flag, containing the flow identifier (ID) and the number of congested flows (n), is immediately generated and sent to S1 before PFC is triggered.

Upon receiving the CNM from downstream switch S2, as shown in Fig. 6(b), FLB stores the flow ID in a flow table (called isolation table) and isolates the congested flow f_2 onto a minimized set of isolated paths (for illustration, Fig. 6(b) shows one path) where the last packet in f_2 is pending, while rerouting subsequent packets of the uncongested flow f_1 to another path. Uncongested flows are not allowed to use the isolated path. When multiple isolation paths are available, the source edge switch randomly assigns the congested flows to one of these paths.

Determining the number of isolation paths. The source edge switch determines this number based on the total converged rate of all congested flows. Assuming the link bandwidth (C) is uniform across the entire cluster, the converged sending rate for each congested flow can be calculated as C/n, where n is the number of congested flows reported in the CNM. The total converged rate of all congested flows, partitioned by the single link bandwidth, determines the number of isolation paths. Maintaining multiple isolation paths with a comparable bandwidth ensures that the source edge switch does not introduce more severe congestion on these congested flows. This further ensures that the congestion root remains constant and does not shift to the source edge switch.



(a) Isolation notification

(b) Isolation operation

Figure 6: FLB isolation mechanism.

Releasing congested flows when congestion subsides. When the queue length of the previously congested port falls below the isolation threshold, a CNM with a non-congestion flag is generated and sent to the corresponding source edge switch. Upon receiving the CNM with the non-congestion flag, the associated congested flow is removed from the isolation table. FLB periodically checks the isolation table to recalculate the required number of isolation paths, releasing one or more paths as necessary. Once the isolation table is empty, all isolation paths are released. In scenarios where the CNM with the non-congestion flag is lost, we rely on timeouts to release inactive congested flow entries.

Determining isolation threshold. One challenge is to determine the isolation threshold for isolating congested flows at switch. If we use a large isolation threshold, PFC may be triggered at a high risk, resulting in more serious congestion spreading. In contrast, a small isolation threshold may prematurely pause the congested flow, resulting in throughput degradation. The detailed analysis of optimizing isolation threshold is leaved in §A and the corresponding experiments are shown in §4.2.

CNM construction. FLB utilizes the direct signal CNM of existing QCN mechanism [48], which is commonly available in commodity switches [52]. However, QCN forwards packets based on L2 addresses and does not preserve the original sender's Ethernet/IP address. To solve it, FLB records flow ID and the source MAC address of the previous hop in the flow

[3] Since the ingress port must reserve buffer space for each priority to absorb inflight packets during the transmission of PFC PAUSE to the upstream switch, the shallow buffer (*e.g.*, 9MB or 12MB) of the commodity switch has no enough space to support PFC on all of the class-of-service queues [5]. As reported in [5], PFC can be enabled on only 2 priorities in practice even though the switches typically support 8 traffic classes.

table at switches. The switch can then propagate the CNM to all the corresponding source edge switches by looking up the flow table hop by hop in IP-routed networks.

3.3 Minimal Rate Control

Enabled by the limited impacts of PFC (achieved by the flow isolation in §3.2), and the well-balanced traffic (achieved by the flexible rerouting in §3.1), FLB reduces the need for and dependency on complex congestion control algorithms. To validate this, we design a minimal rate control, which works as follows:

- All flows start at line rate by sending a bandwidth-delay product (BDP) worth of data packets.
- The source edge switch forwards the CNM to notify the end-hosts of the congested flows. Upon receiving a CNM with a congestion flag, the source host pauses its data transmission for each congested flow and directly sets the future sending rate as $r = \frac{C}{n}$, where C is the line rate and n is the maximum number of congested flows indicated in the CNM. The congested flows resume transmission at the rate r upon a CNM with a non-congestion flag is received. The rate varies with the change of n in the received CNM.

This minimal rate control approach bypasses the iterative convergence process, directly adjusting the sending rate based on CNM information, which significantly reduces the complexity of end-host congestion control.

3.4 Implementation

We implement FLB based on the Wedge 100BF-32X programmable switch [47]. The ingress pipeline with multiple match-action tables for data packets is shown in Fig. 7. Specifically, the metadata $flow.isolation.flag$ is set to 1 to trigger the isolation operation when the queue length exceeds the isolation threshold K. Since the egress queue cannot be read from the ingress intrinsic metadata on P4 hardware platform, we implement the queue-size SRAM with a similar function by counting the packets enqueueing and dequeueing the egress port. In the *Forwardport* tables, for an uncongested flow, FLB selects an egress port according to the rerouting Algorithm 1 in §3.1. For a congested flow, FLB randomly forwards the packet to an isolated port. For a new flow, the packet is routed on the path with minimal delay.

Flow table is maintained at the edge switch. Each flow entry consists of a 16-bit flow ID, a 8-bit selected path ID, a 8-bit aging metric and 48-bit MAC address. Flow ID is calculated based on a CRC16 hash function over the unique 5 tuples of a flow. Aging metric (similar to CONGA [28]) is used to measure packet interval without recording timestamps. Specifically, each incoming packet resets the aging metric of its flow, while a timer periodically increases the metric by one. In this way, the memory consumption is reduced significantly, *e.g.*, only 0.1MB SRAM is used. While in a 3-tier Fat-tree

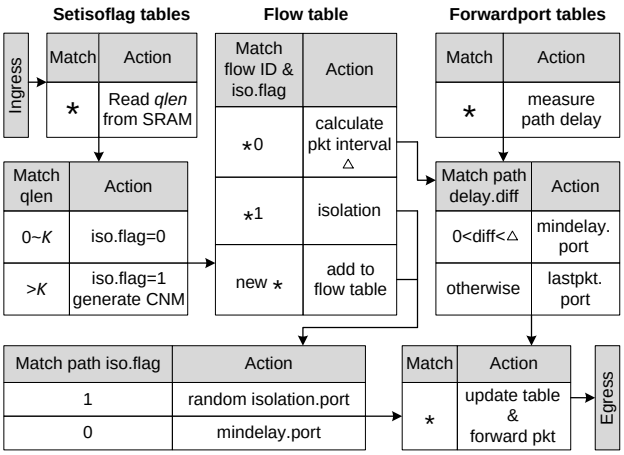


Figure 7: Ingress pipeline for data packets.

topology [53], SRAM consumption increases to 0.12MB due to the increase of path ID to 24bit.

Resource consumption: Table 1 lists the resource consumption of ECMP, LetFlow and FLB under 100K concurrent flows. While the resource consumed by FLB is higher than the others, it occupies no more than 10% of the total switch resource in general. The reason of higher resource consumption is that FLB needs more pipeline stages to isolate congested flows, reroute uncongested flows and flexibly select paths. FLB performs the pipelining of each packet through visiting the match-action tables without violating the access restriction, resulting in more overhead such as Gateway. Meanwhile, FLB needs more SRAM and stateful ALUs to store and update selected path ID and age bits of the last packet. Nonetheless, the overall resources consumption for FLB remains low, meaning that FLB can be deployed in high-speed switch with reasonable resources consumption.

Table 1: Resource consumption of different schemes.

Resource	ECMP	LetFlow	FLB
Match Crossbar	2.41%	4.82%	5.82%
Hash Bits	3.08%	5.67%	5.87%
Gateway	1.39%	2.96%	9.56%
SRAM	1.56%	3.33%	4.12%
VLIW Actions	1.56%	2.34%	3.34%
ALU Instruction	2.6%	5.2%	8.2%

Practical deployment: To reduce deployment overhead in multi-tier topologies, FLB is deployed only on the edge switches while still maintaining comprehensive control over the routing decisions. (1) If FLB is deployed in the two-level topologies (e.g., Leaf-Spine), rerouting is determined solely at the leaf layer. Once the spine switch is selected, the routing to the destination leaf is automatically decided, as there is only a single path from the spine to the destination leaf. (2) If FLB is deployed in the multi-level topologies (e.g., three-layer Fat-Tree), the edge switches maintain the flow table and the delay information of each end-to-end path specified by a

unique path ID [54]. When a packet arrives at an edge switch, it selects the appropriate path based on this information and is forwarded using explicit source routing techniques, such as XPath [54].

4 Evaluation

We evaluate FLB using both testbed experiments and NS3 simulations. Our evaluation centers around the following key questions:

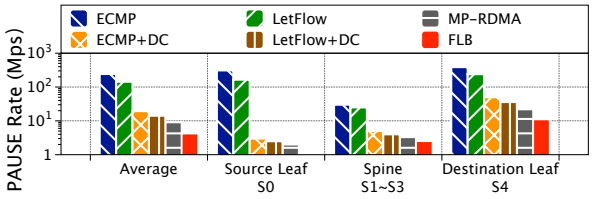
- **How does FLB perform in practice?** Testbed experiments (§4.1) demonstrate the superior performance of FLB. Specifically, FLB reduces the AFCT by 48%, 42% and 30% compared to ECMP+DCQCN, LetFlow+DCQCN and MP-RDMA, respectively.
- **How sensitive is FLB to traffic patterns and parameters, and how effective are the design components?** Deep-dive experiments (§4.2) show that FLB achieves persistent good performance under various traffic patterns and parameter settings, and validate the effectiveness of its design components.
- **How does FLB with minimal rate control (§3.3) (FLB+RC) perform in large-scale DCNs?** Using large-scale NS3 simulations (§4.3), we show that FLB+RC scales well to large and multi-tier network topologies, and performs better than various load balancing mechanisms.

4.1 Testbed Experiments

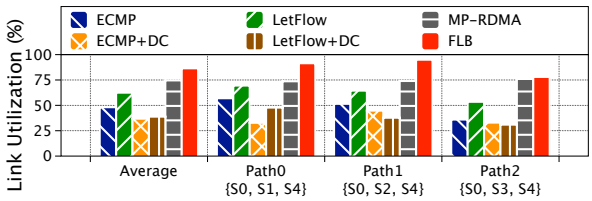
Testbed and parameter settings: We use the same topology as prior work [2, 3] (see Fig. 1) and the testbed settings are the same as in §2. Each server (Dell PRECISION TOWER 5820 server) is equipped with 10 cores Intel Xeon W-2255 CPU, 64GB memory, Mellanox ConnectX-5 100GbE NICs that support DPDK 20.08 and Ubuntu 20.04.1. By default, the link capacity is set to 40Gbps. All parameters in DCQCN [4] and MP-RDMA [7] are set to the recommended values. FLB and FLB+RC respectively indicate without minimal rate control (§3.3) and with minimal rate control (RC).

Results with realistic workload: For this experiment, hosts H0∼H16 generate dynamic traffic according to the realistic Web Search workload [55] with heavy-tailed distribution. The average flow size is 1.6MB. The flows generated from hosts H3∼H16 are concurrent bursty flows. The target load at the bottleneck link is set to 0.6. We measure the flow completion time (FCT), pause rate and link utilization to compare the performance of FLB with the other five schemes.

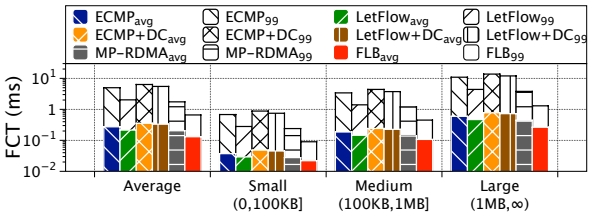
Fig. 8 (a) shows the generating rates of PFC PAUSE from different switch layers. Although the traffic changes dynamically, FLB effectively reduces PFC PAUSEs and suppresses congestion spreading. For FLB, the PAUSE rates at the core and source edge switches are smaller than that at the destination edge switch. Due to lacking of congestion control, the



(a) Generating rate of PAUSEs on different switches.



(b) Link utilization of different parallel paths.



(c) FCT with different flow sizes.

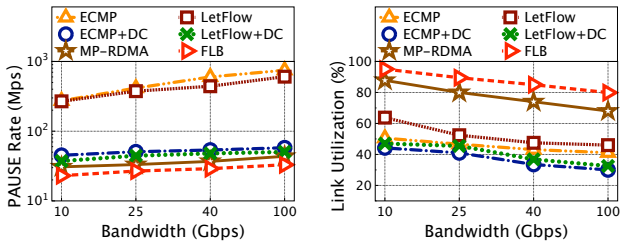
Figure 8: Results with realistic workload.

PAUSE rates of ECMP and LetFlow are larger than those of ECMP+DCQCN and LetFlow+DCQCN, respectively.

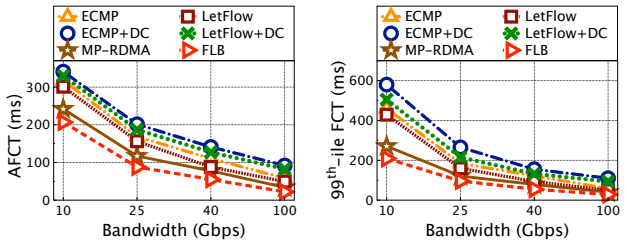
Fig. 8 (b) shows the link utilizations of different parallel paths. By balancing traffic among parallel paths, FLB and MP-RDMA achieve higher link utilization on different paths than the other schemes. In addition, since FLB reduces the number of ports affected by PFC PAUSEs through isolating the congested flows, FLB obtains the highest average link utilization over the other schemes.

Fig. 8 (c) shows the FCTs of small (0,100KB], medium (100KB,1MB] and large (1MB, ∞) flows in Web Search [56]. Since FLB can rapidly isolate and pause the congested flows, the uncongested flows are not blocked by PFC PAUSEs and complete quickly. FLB achieves the lowest average and 99^{th} percentile FCTs of all flows. Compared to ECMP+DCQCN, LetFlow+DCQCN and MP-RDMA, FLB reduces the AFCT of all flows by 48%, 42% and 30%, respectively. The improvement of FLB over ECMP+DCQCN and LetFlow+DCQCN in the 99^{th} percentile FCT is even up to 88% due to successfully avoiding HoL blocking and congestion spreading.

Results with bursty flows: We evaluate the performance of FLB under bursty flows following the same settings as described in §2. Fig. 2 and Fig. 3 show the throughput results of various schemes. We find that FLB effectively avoids the HoL blocking and congestion spreading problems. Specifically, with FLB, (1) uncongested flows f_0 and f_1 no longer suffer



(a) Pause rate (b) Link utilization



(c) AFCT (d) 99^{th} percentile FCT

Figure 9: Results with bursty flows under various link speeds. DC stands for the DCQCN congestion control.

from HoL blocking due to the congested flow f_2 (shown in Fig. 2 (f)); (2) two paths P_0 and P_1 in FLB are no longer affected by the PFC PAUSEs (shown in Fig. 3 (f)).

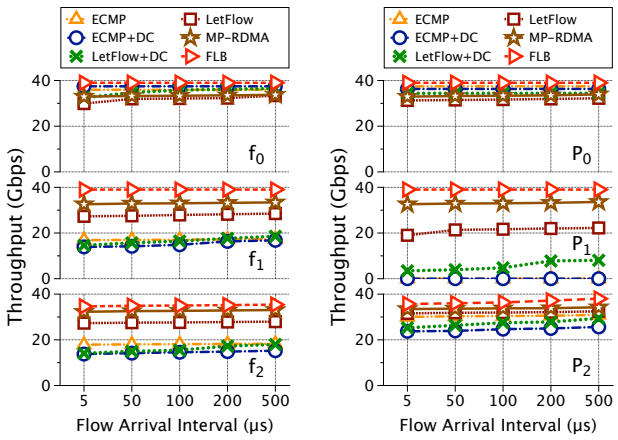
We then vary the link capacity from 10Gbps to 100Gbps, and repeat the same experiments. Fig. 9 (a) shows FLB reduces the rate of PFC PAUSE by up to 96% through isolating the culpable flows. Since the data transmission at line rate leads to fast queueing buildup especially under high-speed links (*e.g.*, 40Gbps, 100Gbps), ECMP and LetFlow trigger significantly more PAUSEs than the other schemes. With employing congestion control, ECMP+DCQCN, LetFlow+DCQCN and MP-RDMA trigger small number of PFC PAUSEs. However, as shown in Fig. 9 (b) and Fig. 9 (d), slow convergence to the target rate under DCQCN results in increments in utilization loss and tail latency, respectively.

Fig. 9 (b) shows that FLB improves link utilization by up to 95%, 78%, 166%, 144% and 28% compared to ECMP, LetFlow, ECMP+DCQCN, LetFlow+DCQCN and MP-RDMA, respectively. Fig. 9 (c) and Fig. 9 (d) show the average and 99^{th} percentile FCTs. Because FLB mitigates PFC triggering between switches and isolates the culpable flows to protect victim flows, FLB achieves lower FCT than the other schemes under various bandwidth of the bottleneck link.

4.2 FLB Deep Dive

We further conduct the testbed experiments to answer the following three questions:

- **How sensitive is FLB to bursty flows?** As traffic demand varies across time and space in DCN, FLB should be robust to dynamic traffic. We find that, even with only 5μs flow arrival interval, FLB still effectively avoids PFC's HoL blocking and achieves high link utilization.
- **How effective is the rerouting mechanism in avoiding packet reordering?** In both symmetric and asymmetric scenarios, FLB effectively avoids reordering problem, and the out-of-order degree is lower than MP-RDMA.
- **What is the effect of the optimized isolation threshold?** Compared to FLB with fixed isolation threshold, FLB with the optimized threshold reduces the PAUSE rate by up to 89% and increases the link utilization by up to 40%.
- **How effective is FLB integrated with rate control?** FLB+RC achieves the lowest flow completion time.



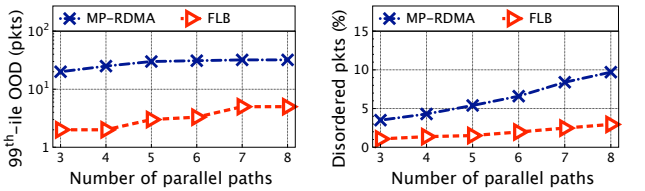
(a) Throughput of flows (b) Throughput of paths

Figure 10: Performance under bursty traffic.

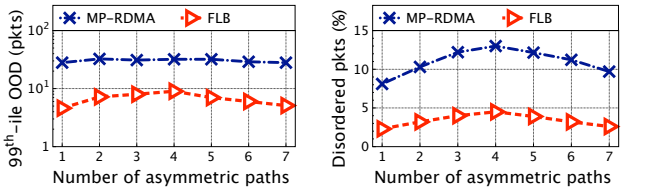
Impact of bursty flow: To explore how FLB performs in bursty scenario, we vary the arrival interval of concurrent flows [2, 57]. Each long flow is 250MB and bursty flows from H3∼H16 to R2 start at different intervals, varying from 5μs to 500μs. We measure the throughputs of each flow and each path. The results are shown in Fig. 10.

We make the following three observations: First, for the uncongested flows f_0 and f_1, FLB maintains high throughput with different arrival intervals of bursty flows. This is because FLB timely isolates the culpable flow f_2 on one path to avoid HoL blocking for uncongested flows. Compared to MP-RDMA, FLB improves the throughputs of f_0 and f_1 by 24% and 20%, respectively. Second, for the congested flow f_2, FLB also achieves higher throughput than other schemes, since f_2 resumes its transmission at line rate, rather than slowly converging to the target rate like DCQCN. In addition, f_2 can flexibly transfer packets among parallel paths when congestion does not occur. Third, FLB and MP-RDMA balance traffic across 3 parallel paths well and achieve high path throughput with varying arrival interval of bursty flows. However, since MP-RDMA splits the congested flow f_2 to all paths, all paths are paused by PFC, resulting in link utilization degradation. FLB achieves up to 19% higher throughput of paths.

Effect of rerouting under symmetric and asymmetric sce-



(a) OOD in symmetric topology (b) Disorder in symmetric topology



(c) OOD in asymmetric topology (d) Disorder in asymmetric topology

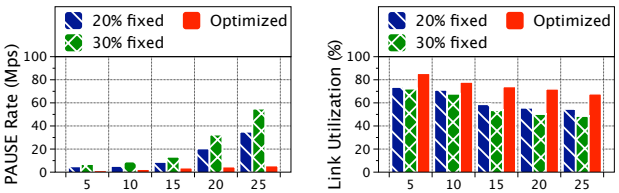
Figure 11: Out-of-order control.

narios: To show rerouting mechanism in FLB is able to well address the out-of-order issue, we conduct experiments in both symmetric and asymmetric topologies and measure the 99^{th} percentile of out-of-order degree (OOD), which is defined as the difference between the sequence numbers of an out-of-order packet and the expected one.

We first vary the number of parallel paths in the symmetric topology. Fig. 11 (a) and Fig. 11 (b) show that FLB effectively reduces the 99^{th} percentile of OOD and disordered packets ratio compared to MP-RDMA. This is because the delay difference between the new rerouting path and the current path is less than the interval between two consecutive packets in the same flow. For MP-RDMA, it can control the OOD to a preset range. Next, in the symmetric topology with 8 equal-cost paths, we change the default 40Gbps bandwidth of some parallel paths to 25Gbps to create bandwidth asymmetry. Fig. 11 (c) and Fig. 11 (d) show FLB achieves low OOD and disordered packets ratio even under the highest asymmetric degree (*i.e.*, the number of asymmetric paths is 4).

Sensitivity to isolation threshold: To validate the effectiveness of the optimized isolation threshold based on Equation (4) in §A, we compare the optimized isolation threshold with the small fixed ones to avoid PFC triggering. We test the fixed thresholds of 20% and 30% of the shared buffer size. We increase the number of concurrent flows for each burst and measure the rate of PFC PAUSE frames from the switch S4 and the average link utilization of the three parallel paths. As shown in Fig. 12, FLB with the optimized threshold achieves 82% and 89% lower PAUSE rates than those of 20% and 30% fixed thresholds, respectively.

Integrated with rate control: To validate the effectiveness of integrating FLB with rate control, we compared the AFCT of four mechanisms, i.e., FLB, FLB+RC, FLB +DCQCN and MP-RDMA under the same experimental setup and Web Search workload as in §4.1. Due to the ability to isolate congested flows and effectively suppress congestion spreading, FLB both with and without rate control is superior to that of MP-RDMA, as shown in Fig. 13. Since the minimal rate control with pause/resume mechanism at the end-host rapidly converges congested flows to the target rate, FLB+RC further reduces the AFCT by 15% compared to pure FLB. In contrast to the end-to-end congestion control protocol DCQCN, which converges the congested flow rate more slowly to the target rate, both FLB and FLB+RC outperform FLB +DCQCN.



(a) PAUSE rate (b) Link utilization

Figure 12: Sensitivity to isolation threshold.

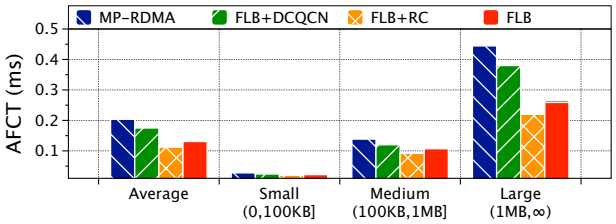


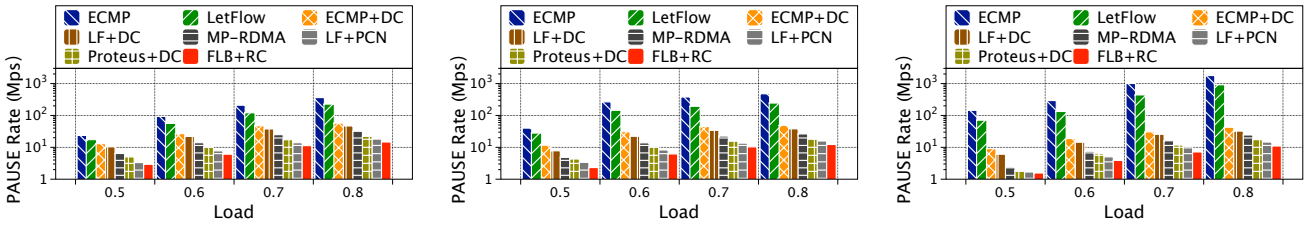
Figure 13: Performance of FLB integrated rate control.

4.3 Large-scale NS3 Simulations

We run NS3 simulations to answer the following questions.

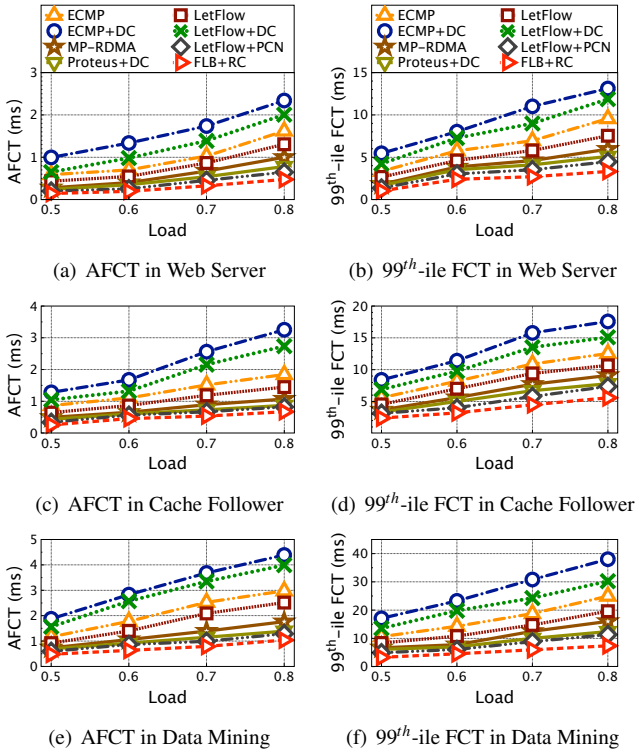
- **How does FLB+RC perform compared to other load balancing schemes with realistic workloads?** FLB+RC outperforms ECMP [43] and LetFlow [31], reducing the AFCT by 65% and 58%, respectively. Furthermore, the AFCT of FLB+RC is respectively 76%, 70%, 36%, 29% and 18% lower than those of ECMP+DCQCN [4], LetFlow+DCQCN, MP-RDMA [7], Proteus+DCQCN [26] and LetFlow+PCN [2] in large-scale scenarios.
- **How does FLB+RC perform with incast workload under the over-subscribed network?** In the incast scenario under a 3:1 over-subscribed topology, FLB+RC delivers persistently higher goodput (up to 27%) compared to other solutions.
- **How does FLB+RC perform under the multi-tier topology?** In a 12-pod Fat-tree topology, FLB+RC achieves lower PAUSE rate and FCT (up to 71% and 92%, respectively) under realistic workloads.

Simulation settings: We evaluate FLB on a leaf-spine topology with 10 leaf switches and 10 spine switches. Each leaf switch is connected to 30 hosts and 10 spine switches with 40Gbps links. The over-subscription ratio is 3:1 at the leaf



(a) Web Server (b) Cache Follower (c) Data Mining

Figure 14: PAUSE rate under realistic workloads. DC and LF stand for DCQCN and LetFlow, respectively.



(a) AFCT in Web Server (b) 99^{th}-ile FCT in Web Server

(c) AFCT in Cache Follower (d) 99^{th}-ile FCT in Cache Follower

(e) AFCT in Data Mining (f) 99^{th}-ile FCT in Data Mining

Figure 15: FCT under realistic workloads.

switch layer. The link delay is 5μs. The switch buffer size is set to 9MB. PFC is enabled to guarantee lossless transmission and the PFC threshold at each ingress port is set to 256KB. To explore the scalability of FLB, we further evaluate its performance under a multi-tier 12-pod Fat-tree topology [51, 53]. Each pod consists of 6 edge switches, 6 aggregation switches, and 84 end-hosts, and communicates with other pods through 36 core switches. So there are 36 equal-cost paths between any pair of edge switches across pods.

Realistic workloads: We use various realistic workloads including Web Server, Cache Follower, Hadoop Cluster and Data Mining [2, 55, 56, 58] with the average flow sizes ranging from 64KB to 7.41MB. In the Web Server workload, all flows are less than 1MB, while in the Data Mining scenario there are around 9% flows larger than 1MB. Each flow is generated between random pair of source and destination hosts. The traffic ratio between intra-ToR and inter-ToR is 1:3. The traffic load is changed from 0.5 to 0.8. We compare the performance of FLB+RC with ECMP, LetFlow, ECMP+DCQCN, LetFlow+DCQCN, LetFlow+PCN, MP-RDMA and Proteus+DCQCN.

Fig. 14 shows the generating rates of PFC PAUSEs from leaf and spine switches. Results validate that FLB suppresses PFC PAUSEs better than the other schemes. This is because FLB pauses the congested flows accurately to drain out the backlog packets, and balances traffic among multiple paths to avoid congestion spreading. Since ECMP and LetFlow start and resume flows at the line rate, they trigger more PFC PAUSEs. In contrast, ECMP+DCQCN, LetFlow+DCQCN, LetFlow+PCN, MP-RDMA and Proteus+DCQCN employ the sophisticated rate adjustment mechanisms to effectively reduce PFC PAUSEs, especially under the Data Mining workload containing more heavy-tailed flows.

Fig. 15 shows the average and 99^{th} percentile FCTs under different realistic workloads and load levels. FLB+RC achieves the best performance across all applications. Taking the Data Mining as an example, FLB+RC reduces the AFCT by 65%, 58%, 76%, 70%, 36%, 29% and 18% under 0.8 load over ECMP, LetFlow, ECMP+DCQCN, LetFlow+DCQCN, MP-RDMA, Proteus+DCQCN and LetFlow+PCN, respectively. This is because FLB isolates the culpable flows timely to avoid the side effects of PFC and flexibly reroutes flows to ensure high link utilization. Due to the slow convergence after resuming, the FCTs of ECMP+DCQCN and LetFlow+DCQCN are larger than ECMP and LetFlow with line-rate resuming, respectively. Since distributing traffic in a congestion-aware manner, MP-RDMA significantly outperforms ECMP and LetFlow. Although multiple paths are potentially paused by PFC in LetFlow+PCN, PCN can identify and throttle the congested flows, effectively reducing PFC triggering and outperforming MP-RDMA.

Incast workload: Next, we evaluate the network performance in the incast scenario where a client initiates concurrent requests to fetch responses from a large number of servers [4, 7, 28]. Simultaneous arrival of the response flows imposes great buffer pressure at the switches, potentially resulting in

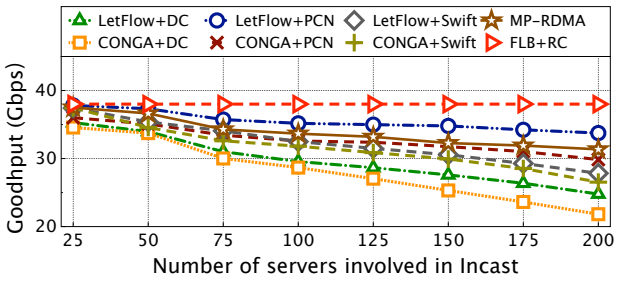


Figure 16: Incast performance vs. number of servers

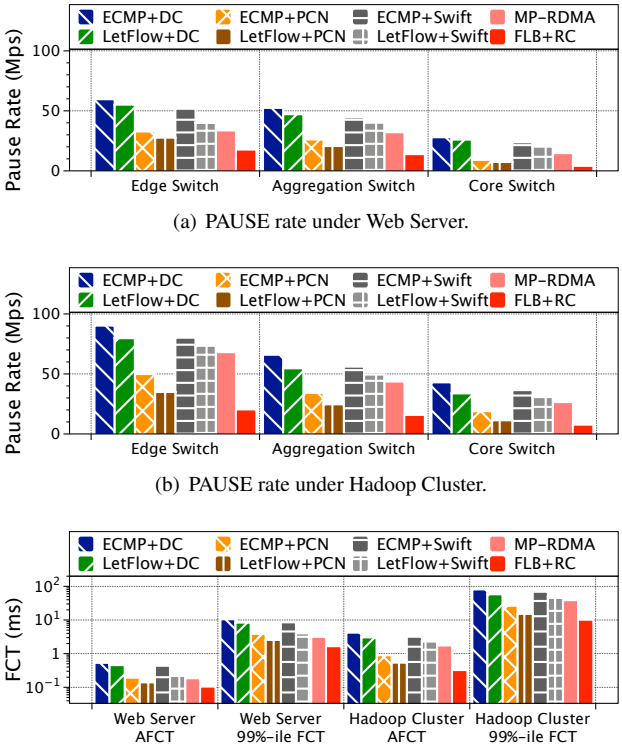
continuous PFC PAUSEs in lossless DCN.

In this test, there are N flows concurrently transmitting a fixed total amount of data from N servers under randomly selected leaf switches to one client. We set the total data size as 100MB and vary the number of servers N from 25 to 200. Each response is $\frac{1}{N}$MB. We measure the goodput at the client and repeat the test for 100 requests to get the average result.

As shown in Fig. 16, although more PFC PAUSE messages are generated with more servers, the throughput of FLB+RC is almost independent of the number of servers. The reason is that FLB pauses and resumes congested flows at a fair rate when the sender receives the CNM with congestion or non-congestion flag, respectively. Compared with other schemes, the incast throughputs in CONGA+DCQCN, LetFlow+DCQCN, CONGA+Swift and LetFlow+Swift are degraded due to the slow convergence of the sending rate. Moreover, CONGA incorrectly reroutes packets to the paused paths with lower link utilization, resulting in a larger throughput loss of up to 45%. Since PCN and MP-RDMA increase the sending rate rapidly after receiving the RESUME messages, they can obtain higher throughput under highly concurrent flows compared with DCQCN.

Multi-tier topology: In this experiment, the workloads are generated among randomly selected host pairs in the aforementioned Fat-tree topology with an exponentially distributed inter-arrival time and targeted load of 0.6.

Fig. 17 (a) and Fig. 17 (b) show the generating rates of PFC PAUSEs from edge, aggregation and core switches. PFC triggers most at the edge switches due to many-to-one communication at the destination end-hosts. Fig. 17 (c) shows the average and 99^{th} percentile FCTs. For Web Server workload, about 80% of short flows less than 10KB and about 10% flows larger than 100KB, resulting in dramatically burst in network congestion. In addition, most short-lived flows have no chance to switch path, resulting in very small difference in performance between deploying ECMP and LetFlow. Since FLB can rapidly detect and isolate the congested flows, the innocent flows complete quickly without suffering from PFC's HoL blocking. Compared with LetFlow+DCQCN, LetFlow+Swift, MP-RDMA and LetFlow+PCN, FLB+RC suppresses 71%,55%, 45% and 37% PFC PAUSEs, reduces 76%, 40%, 32% and 28% average FCT and 81%, 56%, 45%



(a) PAUSE rate under Web Server.

(b) PAUSE rate under Hadoop Cluster.

(c) Average and 99^{th} percentile FCTs.

Figure 17: Performance under Fat-tree topology.

and 32% 99^{th}-ile FCT, respectively. For heavy-tailed Hadoop Cluster workload, about 96% of traffic is provided by only 9% of long flows larger than 1MB, and around 86% short flows less than 100KB. Since FLB effectively avoids PFC's HoL blocking caused by congested long flows, the average and 99^{th}-ile FCTs are reduced by up to 92% and 83% compared with other solutions, respectively.

5 Related Work

We classify previous work on balancing traffic in DCNs into two categories: load-balance routing solutions (*e.g.*, [26–29, 31–39, 43, 59]) and multipath transport solutions (*e.g.*, [7, 30]). Most of them are designed for lossy DCNs and cannot be directly used in lossless Ethernet DCNs.

Load-balance routing solutions [28, 29, 31–39, 43, 59] generally improve link utilization by splitting traffic evenly across multiple paths. ECMP [43], FlowBender [39], Hedera [37], MicroTE [38] schedule flows at flow level, but may obtain suboptimal link utilization due to coarse routing granularity. Presto [34] spreads traffic at fixed flowcell-level granularity. RPS [35], DRILL [36], Hermes [29] and DRB [59] work at packet level and unavoidably suffer from packet reordering problem. CONGA [28], LetFlow [31], HULA [32] and CLOVE [33] employ flowlet-level switching to avoid out-of-order delivery. Proteus [26] and ConWeave [27] effectively balance traffic in the RDMA networks. However, these fine-grained solutions make more ports affected by the side effects

of PFC because they spread the congested flows among more parallel paths. Instead, FLB isolates congested flows timely to eliminate HoL blocking and reroutes uncongested flows by utilizing rich multiple paths to improve link utilization.

Multipath transport solutions [7, 30] divide a flow into multiple subflows for multipath transmission, resulting in potentially more ports to get paused by PFC. There are also efforts [2–4, 8, 10, 12, 22, 48, 60–62] targeting at minimizing FCTs for RDMA. On the one hand, for lossless RDMA, QCN [48] works at Layer 2 to alleviate congestion. IEEE P802.1Qcz [60] supports the isolation of flows from different applications. DCQCN [4] and DCQCN+ [22] are rate-based congestion control for RoCEv2 [61]. TIMELY [10] and Swift [12] use RTT as the congestion signal to adjust sending rate. PCN [2] recognizes the congested flows and employs a receiver-driven rate control algorithm to achieve fast convergence. TCD [3] defines ternary states of switch ports to detect congested ports. BFC [62] uses a per-hop per-flow control scheme to reduce HoL blocking. While the above end-to-end transmission schemes effectively alleviate congestion, they still cannot completely avoid HoL blocking especially under bursty traffic. On the other hand, for lossy RDMA, some works [8, 42, 63] explore the possibility of running RDMA without PFC. However, further efforts are still needed for large-scale deployment in production DCNs.

6 Conclusion

This paper presented FLB, a fine-grained load balancing solution for PFC-enabled lossless DCNs. The core of FLB contains: (i) a flexible rerouting scheme to improve link utilization in normal conditions; and (ii) an isolation mechanism to avoid PFC spreading upon congestion. The testbed experiments as well as large-scale NS3 simulations show that FLB is a viable solution that achieves all our design goals.

Acknowledgments

We thank the anonymous ATC reviewers and our shepherd Prof. Anil kumar Yelam for their constructive feedback and suggestions. This work is supported in part by the National Natural Science Foundation of China (62472050, 62473146, U24A20245, 62132022), the Natural Science Foundation of Hunan Province (2025JJ20070, 2024JJ3017), and the Hong Kong RGC TRS T41-603/20R. Jiawei Huang and Kai Chen are the corresponding authors.

References

[1] W. Bai, A. Agrawal, A. Bhagat, M. Elhaddad, N. John, J. Padhye, M. Pandya, et al. Empowering Azure Storage with 100×100 RDMA. In Proc. USENIX NSDI, 2023.

[2] W. Cheng, K. Qian, W. Jiang, T. Zhang, and F. Ren. Re-architecting Congestion Management in Lossless Ethernet. In Proc. USENIX NSDI, 2020.

[3] Y. Zhang, Y. Liu, Q. Meng, and F. Ren. Congestion Detection in Lossless Networks. In Proc. ACM SIGCOMM, 2021.

[4] Y. Zhu, H. Eran, D. Firestone, C. Guo, M. Lipshteyn, Y. Liron, J. Padhye, S. Raindel, M. H. Yahia, and M. Zhang. Congestion Control for Large-Scale RDMA Deployments. In Proc. ACM SIGCOMM, 2015.

[5] C. Guo, H. Wu, Z. Deng, G. Soni, J. Ye, J. Padhye, and M. Lipshteyn. RDMA over Commodity Ethernet at Scale. In Proc. ACM SIGCOMM, 2016.

[6] K. Qian, W. Cheng, T. Zhang, and F. Ren. Gentle Flow Control: Avoiding Deadlock in Lossless Networks. In Proc. ACM SIGCOMM, 2019.

[7] Y. Lu, G. Chen, B. Li, K. Tan, Y. Xiong, P. Cheng, J. Zhang, E. Chen, and T. Moscibroda. Multipath Transport for RDMA in Datacenters. In Proc. USENIX NSDI, 2018.

[8] R. Mittal, A. Shpiner, A. Panda, E. Zahavi, A. Krishnamurthy, S. Ratnasamy, and S. Shenker. Revisiting Network Support for RDMA. In Proc. ACM SIGCOMM, 2018.

[9] Y. Zhu, M. Ghobadi, V. Misra, and J. Padhye. ECN or Delay: Lessons Learnt from Analysis of DCQCN and TIMELY. In Proc. ACM CoNEXT, 2016.

[10] R. Mittal, V. T. Lam, N. Dukkipati, E. Blem, H. Wassel, M. Ghobadi, A. Vahdat, Y. Wang, D. Wetherall, and D. Zats. TIMELY: RTT-based Congestion Control for the Datacenter. In Proc. ACM SIGCOMM, 2015.

[11] Y. Li, R. Miao, H. H. Liu, et al. HPCC: High Precision Congestion Control. In Proc. ACM SIGCOMM, 2019.

[12] G. Kumar, N. Dukkipati, K. Jang, et al. Swift: Delay is Simple and Effective for Congestion Control in the Datacenter. In Proc. ACM SIGCOMM, 2020.

[13] S. Hu, Y. Zhu, P. Cheng, C. Guo, K. Tan, J. Padhye, and K. Chen. Tagger: Practical PFC Deadlock Prevention in Data Center Networks. In Proc. ACM CoNEXT, 2017.

[14] J. Xue, M. U. Chaudhry, B. Vamanan, T. N. Vijaykumar, and M. Thottethodi. Dart: Divide and Specialize for Fast Response to Congestion in RDMA-based Datacenter Networks. IEEE/ACM Transactions on Networking, 28(1):322-335, 2020.

[15] C. Tian, B. Li, L. Qin, J. Zheng, J. Yang, W. Wang, G. Chen, and W. Dou. P-PFC: Reducing Tail Latency with Predictive PFC in Lossless Data Center Networks. IEEE Transactions on Parallel and Distributed Systems, 31(6):1447-1459, 2020.

[16] Microsoft. Availability of Linux RDMA on Microsoft Azure. https://azure.microsoft.com/en-us/blog/azure-linux-rdma-hpc-available/.

[17] Alibaba. Alibaba Cloud - Super Computing Cluster. https://www.alibabacloud.com/product/scc.

[18] Google. Accelerate Your Transformation with Google Cloud. https://cloud.google.com.

[19] B. Yi, J. Xia, L. Chen, and K. Chen. Towards Zero Copy Dataflows Using RDMA. In Proc. ACM SIGCOMM Posters and Demos, 2017.

[20] IEEE 802.1 Qbb - Priority-based Flow Control. https://1.ieee802.org/dcb/802-1qbb/.

[21] Z. Guo, S. Liu, and Z. Zhang. Traffic Control for RDMA-Enabled Data Center Networks: A Survey. IEEE Systems Journal, 4(1):677-688, 2020.

[22] Y. Gao, Y. Yang, C. Tian, J. Zheng, B. Mao, and G. Chen. DCQCN+: Taming Large-scale Incast Congestion in RDMA over Ethernet Networks. In Proc. IEEE ICNP, 2018.

[23] M. Miao, F. Ren, X. Luo, J. Xie, Q. Meng, and W. Cheng. SoftRDMA: Rekindling High Performance Software RDMA over Commodity Ethernet. In Proc. ACM APNet, 2017.

[24] H. Qiu, X. Wang, T. Jin, Z. Qian, B. Ye, B. Tang, W. Li, and S. Lu. Toward Effective and Fair RDMA Resource Sharing. In Proc. ACM APNet, 2018.

[25] P. Pan, G. Chen, X. Wang, H. Dai, B. Li, B. Fu, and K. Tan. Towards Stateless RNIC for Data Center Networks. In Proc. ACM APNet, 2019.

[26] J. Hu, C. Zeng, Z. Wang, J. Zhang, K. Guo, H. Xu, J. Huang, K. Chen. Enabling Load Balancing for Lossless Datacenters. In Proc. IEEE ICNP, 2023.

[27] C. H. Song, X. Z. Khooi, R. Joshi, I. Choi, J. Li, M. C. Chan. Network Load Balancing with In-network Reordering Support for RDMA. In Proc. ACM SIGCOMM, 2023.

[28] M. Alizadeh, T. Edsall, S. Dharmapurikar, R. Vaidyanathan, K. Chu, A. Fingerhut, V. T. Lam, F. Matus, R. Pan, N. Yadav, G. Varghese. CONGA: Distributed Congestion-aware Load Balancing for Datacenters. In Proc. ACM SIGCOMM, 2014.

[29] H. Zhang, J. Zhang, W. Bai, K. Chen, and M. Chowdhury. Resilient Datacenter Load Balancing in the Wild. In Proc. ACM SIGCOMM, 2017.

[30] C. Raiciu, S. Barre, C. Pluntke, A. Greenhalgh, D. Wischik, and M. Handley. Improving Datacenter Performance and Robustness with Multipath TCP. In Proc. ACM SIGCOMM, 2011.

[31] E. Vanini, R. Pan, M. Alizadeh, P. Taheri and T. Edsall. Let It Flow: Resilient Asymmetric Load Balancing with Flowlet Switching. In Proc. USENIX NSDI, 2017.

[32] N. Katta, M. Hira, C. Kim, A. Sivaraman, and J. Rexford. Hula: Scalable Load Balancing Using Programmable Data Planes. In Proc. ACM Symposium on SDN Research, 2016.

[33] N. Katta, A. Ghag, M. Hira, I. Keslassy, A. Bergman, C. Kim, and J. Rexford. Clove: Congestion-Aware Load Balancing at the Virtual Edges. In Proc. ACM CoNEXT, 2017.

[34] K. He, E. Rozner, K. Agarwal, W. Felter, J. Carter and A. Akellay. Presto: Edge-based Load Balancing for Fast Datacenter Networks. In Proc. ACM SIGCOMM, 2015.

[35] A. Dixit, P. Prakash, Y. C. Hu, and R. R. Kompella. On the Impact of Packet Spraying in Data Center Networks. In Proc. IEEE INFOCOM, 2013.

[36] S. Ghorbani, Z. Yang, P. Godfrey, Y. Ganjali, and A. Firoozshahian. DRILL: Micro Load Balancing for Low-latency Data Center Networks. In Proc. ACM SIGCOMM, 2017.

[37] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat. Hedera: Dynamic Flow Scheduling for Data Center Networks. In Proc. USENIX NSDI, 2010.

[38] T. Benson, A. Anand, A. Akella, and M. Zhang. MicroTE: Fine Grained Traffic Engineering for Data Centers. In Proc. ACM CoNEXT, 2011.

[39] A. Kabbani, B. Vamanan, J. Hasan, and F. Duchene. FlowBender: Flow-level Adaptive Routing for Improved Latency and Throughput in Datacenter Networks. In Proc. ACM CoNEXT, 2014.

[40] C. H. Song, X. Z. Khooi, R. Joshi, I. Choi, J. Li, M. C. Chan. Network Load Balancing with In-network Reordering Support for RDMA. In Proc. ACM SIGCOMM, 2023.

[41] P. Bosshart, D. Daly, G. Gibb, et al. P4: Programming Protocol-independent Packet Processors. In Proc. ACM SIGCOMM Computer Communication Review, 44(3):87-95, 2014.

[42] Z. Wang, L. Luo, Q. Ning, et al. SRNIC: A Scalable Architecture for RDMA NICs. In Proc. USENIX NSDI, 2023.

[43] C. Hopps. Analysis of an Equal-cost Multipath Algorithm. RFC 2992, Internet Engineering Task Force. 2000.

[44] A. Gangidi, R. Miao, S. Zheng, S. J. Bondu, G. Goes, H. Morsy, R. Puri, M. Riftadi, A. J. Shetty, J. Yang, et al. RDMA over Ethernet for Distributed Training at Meta Scale. In Proc. ACM SIGCOMM, 2024.

[45] Q. Hu, Z. Ye, Z. Wang, G. Wang, M. Zhang, Q. Chen, P. Sun, D. Lin, X. Wang, Y. Luo, Y. Wen and T. Zhang. Characterization of large language model development in the datacenter. In Proc. USENIX NSDI, 2024.

[46] DPDK Plane Development Kit, Intel DPDK, 2019.

[47] Edgecore Networks. https://www.edge-core.com/productsInfo.php?cls=1&cls2=180&cls3=181&id=335.

[48] IEEE. 802.1Qau – Congestion Notification. http://www.ieee802.org/1/pages/802.1au.html.

[49] C. Lee, C. Park, K. Jang, S. Moon, and D. Han. Accurate Latency-based Congestion Feedback for Datacenters. In Proc. USENIX NSDI, 2015.

[50] J. Liu, J. Huang, W. Li, and J. Wang. AG: Adaptive Switching Granularity for Load Balancing with Asymmetric Topology in Data Center Network. In Proc. IEEE ICNP, 2019.

[51] P. Wang, G. Trimponias, H. Xu and Y. Geng. Luopan: Sampling-based Load Balancing in Data Center Networks. IEEE Transactions on Parallel and Distributed Systems, 30(1):133-145, 2018.

[52] A. Saeed, V. Gupta, P. Goyal, M. Sharif, R. Pan, M. Ammar, E. Zegura, K. Jang, M. Alizadeh, A. Kabbani, and A. Vahdat. Annulus: A Dual Congestion Control Loop for Datacenter and WAN Traffic Aggregates. In Proc. ACM SIGCOMM, 2020.

[53] M. Al-Fares, A. Loukissas, A. Vahdat. A Scalable, Commodity Data Center Network Architecture. In Proc. ACM SIGCOMM, 2008.

[54] S. Hu, K. Chen, H. Wu, W. Bai, C. Lan, H. Wang, H. Zhao, and C. Guo. Explicit Path Control in Commodity Data Centers: Design and Applications. In Proc. USENIX NSDI, 2015.

[55] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data Center TCP (DCTCP). In Proc. ACM SIGCOMM, 2010.

[56] W. Bai, L. Chen, K. Chen, D. Han, C. Tian, and H. Wang. Information-agnostic Flow Scheduling for Commodity Data Centers. In Proc. USENIX NSDI, 2015.

[57] T. Benson, A. Akella, and D. Maltz. Network Traffic Characteristics of Data Centers in the Wild. In Proc. IMC, 2010.

[58] S. Hu, W. Bai, G. Zeng, Z. Wang, B. Qiao, K. Chen, K. Tan, and Y. Wang. Aeolus: A Building Block for Proactive Transport in Datacenters. In Proc. ACM SIGCOMM, 2020.

[59] J. Cao, R. Xia, P. Yang, C. Guo, G. Lu, L. Yuan, Y. Zheng, H. Wu, Y. Xiong, and D. Maltz. Per-packet Load-balanced, Low-latency Routing for Clos-based Data Center Networks. In Proc. ACM CoNEXT, 2013.

[60] IEEE P802.1 Qcz – Congestion Isolation. https://1.ieee802.org/tsn/802-1qcz/.

[61] InfiniBand Trade Association. Infiniband Architecture Specification Volume 1 Release 1.2.1 Annex A17: RoCEv2, 2014. https://cw.infinibandta.org/document/dl/7781.

[62] P. Goyal, P. Shah, K. Zhao, G. Nikolaidis, M. Alizadeh, and T. E. Anderson. Backpressure Flow Control. In Proc. USENIX NSDI, 2022.

[63] A. Shpiner, E. Zahavi, O. Dahley, A. Barnea, R. Damsker, G. Yekelis, M. Zus, E. Kuta, and D. Baram. RoCE Rocks without PFC: Detailed Evaluation. In Proc. ACM KBNets, 2017.

[64] C. Clos. A study of non-blocking switching networks. Bell System Technical Journal, 32(2):406–424, 1953.

A Optimizing Isolation Threshold

In this section, we theoretically analyze the isolation threshold K to simultaneously avoid PFC triggering and link underutilization. We assume a typical multi-tier DCN topology, like Fat-tree [53] or 3-stage Clos [64], in which n senders send flows to n receivers via multiple equal-cost paths between the source and destination edge switches. The one-way delay between the sender S_i and the destination edge switch DS is d_i. The instantaneous queue length of DS's egress port is $Q(t)$ at time t. The sending rate of each sender is $v_i(t)$ and the link capacity of DS's egress port is C.

Note however, the isolation notification is generated from the egress queue while PFC PAUSE is from the ingress queue of DS. Thus, the isolation threshold K and PFC threshold Q_{PFC} are for the egress and ingress queues, respectively. The worst case in the above network model is that packets coming from a single ingress queue of DS are sent to all n egress

queues. To guarantee that PFC is not triggered on this corresponding ingress queue before isolation is triggered on the egress queue, the condition $K < \frac{Q_{PFC}}{n}$ should be satisfied. However, an isolation notification (CNM) will take some time to arrive at the sender. To avoid PFC triggering, the egress queue must reserve enough buffer to accommodate the in-flight packets. Therefore, we analyze the reasonable range of K with considering the above worst case.

Specifically, we assume that, when the egress queue length increases to $Q(t_P)$ at time t_P, the isolation notification CNM is generated and sent to the sender S_i for pausing the congested flow. Due to the transmission delay of CNM, the sending rate $v_i(t)$ of S_i cannot be reduced to zero immediately and the egress queue length continues to increase for a short time $2d_i$. The maximum egress queue length $Q(t_P + 2d)$ ($d = \max d_i$) can be given by

$$Q(t_P + 2d) = Q(t_P) + \sum_{i=1}^{n} \int_{t_P}^{t_P+2d} v_i(t)dt - 2d \times C. \quad (1)$$

To ensure that PFC is not triggered, the condition $Q(t_P + 2d) < \frac{Q_{PFC}}{n}$ needs to be satisfied. Thus, we obtain

$$Q(t_P) < \frac{Q_{PFC}}{n} - \sum_{i=1}^{n} \int_{t_P}^{t_P+2d} v_i(t)dt + 2d \times C. \quad (2)$$

On the other hand, we assume that when the egress queue length drains to $Q(t_R)$ at time t_R, a non-congestion notification message is sent to the sender S_i for resuming the transmission of the congested flow. Similarly, the egress queue length will keep on decreasing for a short time of $2d_i$. To avoid the link under-utilization, the isolation threshold $Q(t_R)$ should satisfy the following condition.

$$Q(t_R) \geq 2d \times C - \sum_{i=1}^{n} \int_{t_R}^{t_R+2d} v_i(t)dt. \quad (3)$$

Hence, to ensure that the isolation for congested flows is always triggered before PFC and the congested flows resume transmission before the egress queue becomes empty, the isolation threshold K is set in the range

$$\begin{aligned} K \in [&2d \times C - \sum_{i=1}^{n} \int_{t_R}^{t_R+2d} v_i(t)dt, \\ &\frac{Q_{PFC}}{n} - \sum_{i=1}^{n} \int_{t_P}^{t_P+2d} v_i(t)dt + 2d \times C). \end{aligned} \quad (4)$$

In our implementation, we set a conservative isolation threshold K in the range of $[2d \times C, \frac{Q_{PFC}}{n} - 2d \times C \times (n-1))$. Our experiments show that such isolation threshold is effective and robust to a wide range of traffic variations (§4).

SNARY: A High-Performance and Generic SmartNIC-accelerated Retrieval System

Qiaoyin Gan[2,3], Heng Pan[1], Luyang Li[2], Kai Lv[2], Hongtao Guan[2],
Zhaohua Wang[1], Zhenyu Li[2], and Gaogang Xie[1,3]

[1]*Computer Network Information Center, Chinese Academy of Sciences*
[2]*Institute of Computing Technology, Chinese Academy of Sciences*
[3]*University of Chinese Academy of Sciences*

Abstract

Industrial large-scale recommendation systems mostly follow a two-stage paradigm: retrieval and ranking stages. The retrieval stage aims to select thousands of relevant candidates from a vast corpus with millions or more items, and thus often becomes the performance bottleneck. Offloading the retrieval stage to hardware is a promising solution. Nevertheless, previous solutions either fail to achieve optimal performance or lack the sufficient generality to support fuzzy search, which has been widely used in modern retrieval systems to improve their scalability and efficiency.

In this paper, we present SNARY, a generic SmartNIC-accelerated retrieval system, to facilitate both exact and fuzzy search. Specifically, SNARY utilizes High-Bandwidth Memory (HBM) for corpus storing and scanning and designs two types of search engines: a data parallelism exact search, and a Locality-Sensitive Hashing (LSH)-based fuzzy search. Furthermore, SNARY employs a pipeline-based approach to select Top-K items and streams the data flow of the whole system. We have implemented SNARY on Xilinx commercial SmartNICs. Experimental results show SNARY achieves a 20.91%-83.88% lower latency and a 1.26×-18.27× higher latency-bounded throughput in exact search scenarios, and achieves a 85.13%-87.40%lower latency and a 20.18×-23.81× higher latency-bounded throughput in fuzzy search scenarios in comparison with the state-of-the-art hardware-based solutions.

1 Introduction

Due to the rapid growth in the amount of information, recommendation systems have been widely applied in many web-scale applications (e.g., Amazon and ByteDance) to assist users in finding their preferred items. A typical industrial recommendation system often consists of two sequential stages [20, 36]: matching (a.k.a retrieval) and ranking[1] (see Figure 1). The retrieval stage selects thousands of relevant items from a very large-scale corpus (e.g., millions of items) while the ranking stage further filters out the top dozens of items from the selected small-scale candidates with high precision and recall. In this paper, we mainly focus on the retrieval stage.

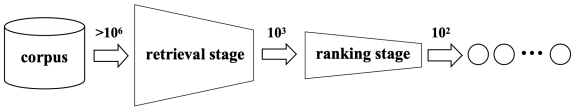


Figure 1: Hierarchical structure of the recommendation systems. Recommendation systems are composed of two distinct stages: retrieval and ranking.

With advancements in deep learning, neural retrieval models are being extensively applied in the retrieval stage, with progressively better results [5, 36]. Among them, representation learning [1], which is also called *embedding*, has become the mainstream trend to facilitate deep retrieval in practice [15, 20, 27]. Specifically, in an embedding-based retrieval (EBR) system, all corpus items and user queries are represented with semantic embedding vectors via two parallel deep neural networks. The retrieval process of the EBR system can be regarded as a Nearest Neighbor (NN) algorithm: given one user query, a matching function is then used to calculate the similarity (a.k.a distance) in the embedding space for each corpus item with the query. Each retrieval operation is associated with a recall count, K, indicating the number of embeddings that require final retrieval. After similarity calculation, a sorting algorithm is deployed to efficiently select Top-K items based on their similarity to the query. That is to say, the retrieval problem is finally converted into a similarity search problem.

According to the retrieval scenario requirements and specific algorithm implementations, retrieval methods are further divided into exact search and fuzzy search. Exact search is based on the K-Nearest Neighbor (KNN) algorithm, and the retrieval results are the top K embeddings that are closest to the user query, which are strictly sorted according to the distances between embeddings, while fuzzy search is based on

[1]Recent works further divide the ranking into three phases: pre-ranking, ranking and re-ranking [27, 28].

Approximate Nearest Neighbor (ANN) algorithms, and the retrieval results are embeddings with relatively high similarity to the user query, but not necessarily exact, which usually sacrifices a certain degree of precision to improve the retrieval speed [20]. Due to the high precision of retrieval results, exact search is very popular in various fields [25, 51]. It is worth noting that fuzzy retrieval can provide more effective solutions in scenarios where exact results are not required and lower latency is needed, such as video recommendation [5], product search [27, 47], and information retrieval [15]. At the same time, there are some scenarios where the combination of exact search and fuzzy search is required to improve the stability of the system's response time. For example, a specific recommendation system has significant traffic differences at different times of the day. Appropriately using fuzzy search during high-traffic periods and exact search during relatively stable traffic periods can effectively balance the system's response time and enhance the user experience.

However, existing EBR systems usually have some deficiencies in two aspects: (*i*) the growing corpus sizes have brought about serious performance problems; (*ii*) the requirements of different scenarios for retrieval methods bring about the issue of generality. For the performace issue, the EBR systems are required to fetch all corpus items, calculate their similarity, and choose Top-K items as long as they receive one user query. Consequently, the corpus size determines the scale of the problem. Low retrieval performance (high response delay) indeed degrades user experience and satisfaction. Therefore, the recent interest in the community has moved to the implementation of high-performance EBR systems, especially with hardware accelerators. For example, some works exploit GPUs (Graphics Processing Units) to accelerate those computation-intensive tasks in EBR, such as similarity computation [49]. There are already mature GPU retrieval system frameworks that support both exact search and fuzzy search simultaneously [25]. However, GPUs are not optimized for pipeline parallelism that is required by the data flow of the whole EBR system, especially at the Top-K selection stage due to the cost of external memory communication. To this end, the community has turned to another popular hardware accelerator, FPGA (Field Programmable Gate Array), which does not have the inherent limitations of GPUs. Unfortunately, the state-of-the-art FPGA-based EBR system [51] only works with exact search and falls short for fuzzy search, lacking support for both types of retrieval methods which brings limited generality.

In this paper, we design SNARY, a **S**mart**N**IC-**A**ccelerated **R**etrieval s**Y**stem, which can achieve high performance and generality for both exact and fuzzy search. We choose SmartNICs for two reasons: (*i*) modern SmartNICs are often equipped with FPGAs; (*ii*) SmartNICs are the nearest devices to user queries in servers because queries are often from remote users in practice. That is to say, SNARY can directly process user queries when the NIC receives them. Overall, SNARY is integrated with two types of search engines: exact search and fuzzy search. For exact search, SNARY utilizes the FPGA high bandwidth memory (HBM) to store and scan the corpus. With massive fully programmable compute elements on FPGAs, SNARY designs a data-parallelism method to accelerate similarity computation, and a pipeline-parallelism solution to facilitate Top-K item selection. In addition, the whole data flow of SNARY also follows the pipeline parallelism paradigm. For fuzzy search, SNARY proposes a Locality-Sensitive Hashing (LSH)-based [7, 8, 45] search architecture on FPGAs. Specifically, we design multiple LSH hash tables to maintain the corpus on FPGA HBM. With this basis, SNARY can quickly select relevant retrieval results for one query via hash table lookup, which constitutes a small new "corpus". Then, SNARY will further perform a more effective exact search operation on the new corpus for final fuzzy retrieval results.

We implement SNARY based on Alveo™ U50 Data Center Accelerator Card, and compare it with Faiss [25], a popular GPU-based EBR system, and FAERY [51], the state-of-the-art FPGA-based solution. Experimental results show that SNARY achieves a 78.75%-83.88% lower query latency than Faiss and a 20.91%-45.19% lower query latency than FAERY, with a 14.12×-18.27× higher latency-bounded throughput than Faiss and a 1.26×-1.64× higher latency-bounded throughput than FAERY in exact search scenarios. It also achieves a 85.13%-87.40% lower query latency than Faiss, with a 20.18×-23.81× higher latency-bounded throughput than Faiss in fuzzy search scenarios, while FAERY does not support fuzzy search. To sum up, this paper makes the following contributions:

- To the best of our knowledge, SNARY is the first SmartNIC-based retrieval system that achieves high performance for both exact and fuzzy searches.
- We utilize large HBM and programmable compute elements on FPGAs to design a data-parallelism similarity computation and a pipeline-parallelism Top-K selection to facilitate exact search. With this basis, we further propose an LSH-based fuzzy search on SmartNICs.
- We implement SNARY on commercial SmartNICs, and evaluate its performance and efficiency. Experimental results show that SNARY outperforms the popular GPU-based approach and the novel FPGA-based approach.

2 Background and Motivation

2.1 Embedding-based Retrieval

Industrial recommendation systems often adopt embedding-based retrieval (EBR) to provide relevant or personalized product candidates [20, 27]. The basic idea of EBR is to translate all corpus items and user queries into embedding vectors via representation learning [1], and convert the retrieval prob-

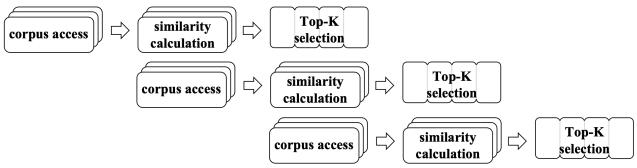
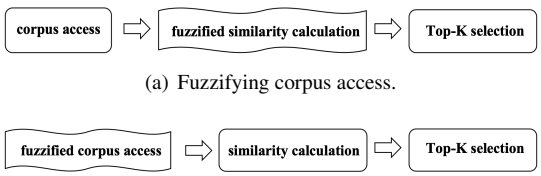


Figure 2: The workflow of an ideal and feasible EBR system.



(a) Fuzzifying corpus access.

(b) Fuzzifying similarity calculation.

Figure 3: Two methods of fuzzy retrieval solutions. Fuzzy retrieval solutions include fuzzifying similarity calculation stage and fuzzifying corpus access stage.

lem into a similarity search problem in the embedding space. It is noteworthy that corpus item embeddings often are precomputed (offline) in practice.

Overall, a typical EBR algorithm mainly consists of three steps: corpus access, similarity computation and Top-K selection. Specifically, given a user query, the algorithm is required to scan the corpus and iterate through its all items (i.e., corpus access) and calculate the similarity between each item and the user query (i.e., similarity computation). After that, the algorithm sorts the items based on their similarity and selects Top-K items (i.e., Top-K selection) as the retrieval results. It is clear that corpus access is a memory-intensive operation while both similarity calculation and Top-K selection are compute-intensive.

Ideal EBR system. With the analysis of the basic EBR algorithm, we then introduce an ideal EBR system architecture that can provide optimal retrieval performance. In summary, it should satisfy the following conditions.

- **Large high-bandwidth memory.** Due to the growing corpus sizes, the EBR system requires a large-capacity external memory to store it. In addition, the memory should provide high bandwidth in order to support the memory-intensive operation (corpus access).
- **Task/Data parallelism.** Logically, the similarity calculations for different item embedding are independent. To match the throughput of corpus access, the ideal EBR system should perform calculations with data parallelism.
- **Pipeline parallelism.** To achieve optimal latency, both inside Top-K selection and the whole EBR data flow should follow the pipeline parallelism paradigm. Otherwise, the accumulation of intermediate results is unacceptable in terms of resource consumption in scenarios with large scale corpus.

Comparison of different hardware. At present, most mainstream EBR systems are implemented on CPUs, GPUs and FPGAs. For CPUs, the relatively low memory bandwidth slows down the reading speed of the corpus, which directly affects the overall performance of the system; at the same time, the relatively small number of CPU cores also poses a challenge to data parallelism. Compared with CPUs, GPUs provide large-capacity and high-bandwidth external memories and have good support for data parallelism. However, due to the overhead of cross-core communication in GPUs and the limitations of single-core memory, GPUs have relatively large defects in supporting the overall pipelining of the system, especially in the Top-K selection stage. Research shows that currently, mature GPU frameworks consume 80% of latency in the Top-K selection stage [51]. Meanwhile, this also limits the maximum supported recall count (e.g., 1024 in [25]). Advanced FPGAs also provide large-capacity and high-bandwidth external memories through HBM. Meanwhile, unlike GPUs, FPGAs provide sufficient on-chip memories that can be directly accessed, and all computing units and interconnects are programmable, which enables FPGAs to better support the implementation of fully pipelined systems.

Performance factors. Figure 2 illustrates the workflow of a pipelined EBR system. Actually, the total latency of the pipelined system is determined by two portions: (*i*) the cost of corpus access; (*ii*) the time for one batch of corpus items to pass through the pipeline, going through similarity calculation and Top-K selection. Thus, we formalize the theoretical latency of the retrieval system in terms of pipeline cycles (denoted by L) as follows:

$$L = \frac{M}{B} + C, \tag{1}$$

where M represents the corpus size, B represents the bandwidth of external memory which determines the throughput in any pipeline cycle, and C is a constant denoting the pipeline cycles needed to process a single batch of corpus items. That is, C is dependent on the specific design and implementation of one EBR system. Since the entire system is pipelined, once the corpus has been fully read, the last batch of the corpus passes through the pipeline, and the entire system completes its operation.

2.2 Accelerating Retrieval with Fuzzy Search

The retrieval phase can be accelerated through fuzzy search, a technique widely employed in industry. It often utilizes ANN algorithms (e.g., LSH [7]) to achieve low response time, but at the cost of slight accuracy degradation. However, such a trade-off is worthwhile in many time-sensitive scenarios, such as search engines and recommendation systems.

Existing fuzzy retrieval solutions mainly focus on fuzzifying the similarity calculation [6, 22, 31] (see Figure 3(a)).

For example, some works employ lexical fuzzification approach [14, 17], synonym expansion technique [39, 43], and machine learning-based methods [29, 48] to enhance the flexibility of matching between user queries and corpus items. These approaches are mainly applied in CPU-based retrieval systems to compensate for the deficiency of data parallelism in CPU-based similarity calculation. However, for those scenarios that involve large-scale corpus, the corpus size often determines the overall latency (see Equation 1). In such cases, fuzzifying similarity calculations has little effect on performance. Instead, fuzzifying corpus access may lead to more improvement (see Figure 3(b)), which means that instead of scanning the entire corpus, only the part of the corpus that is highly relevant to the user's query is selected for retrieval. According to Equation 1, the latency for reading the corpus accounts for the vast majority of the overall system latency. Reducing the size of the corpus can effectively decrease the latency and improve the retrieval speed.

2.3 SmartNIC Opportunities

We have found that accelerating the retrieval system through a SmartNIC implementation has the following advantages:

- SmartNICs are the nearest devices to user queries in servers since queries are often from remote users. SmartNIC-accelerated retrieval systems have the opportunity to achieve optimal performance since they have the shortest data path.
- Accelerating the retrieval system through SmartNICs, i.e., implementing the retrieval phase on FPGA, leverages the hardware advantages of FPGA. It harnesses FPGA's programmability to implement custom data parallelism and pipelining strategies, ultimately achieving a retrieval system with favorable latency.
- FPGAs not only provide ample external storage space for storing large-scale corpus but also allow for preprocessing of the corpus using ANN algorithms. This enables fuzzy search acceleration without compromising the pipeline of the retrieval, thereby simultaneously supporting both exact and fuzzy search scenarios.

3 SNARY Overview

We design SNARY, a high-performance and generic SmartNIC-accelerated retrieval system, that follows the aforementioned ideal EBR system architecture. It is noteworthy that SNARY supports both exact and fuzzy searches. Figure 4 shows its conceptual architecture.

Overall, SNARY utilizes High Bandwidth Memory (HBM) on FPGAs to store a large-scale corpus and enable efficient corpus access. Furthermore, SNARY builds LSH hash tables on HBM in advance to maintain corpus items. Consequently,

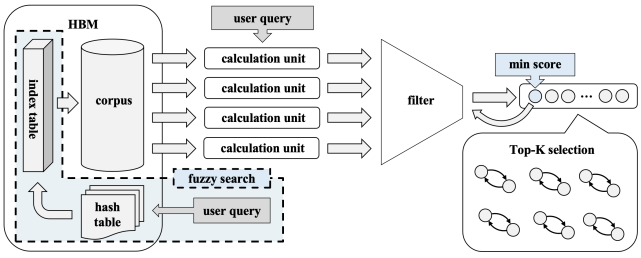


Figure 4: The conceptual architecture of SNARY. SNARY consists of a HBM, multiple similarity computation units, a filter and a pipeline-parallel Top-K selection module.

similar items will be inserted into one or neighbor hash buckets[2].

For a query that requires an exact search, SNARY will fetch a batch of items from the corpus, and feed them to multiple similarity calculation instantiations built upon massive fully programmable compute elements of FPGAs. That is, SNARY calculates the similarity between corpus items and the user query in a data-parallel manner. After that, the items will be transmitted to a parallel Top-K selection module which perfectly supports the pipelining framework, selecting the final retrieval results based on their similarity. To balance the throughput between the similarity calculation and the Top-K selection modules, SNARY inserts a filter in-between to drop relatively poor items.

If a user performs a fuzzy retrieval, SNARY will use the user query as a key to look up the LSH hash tables. Those items in the matched hash bucket and its neighbor hash buckets will constitute a new small-scale corpus. Then, SNARY will perform an exact search on the new corpus. That is, SNARY fuzzifies corpus access and significantly reduces the size of the target corpus. This indeed reduces the latency of the overall system.

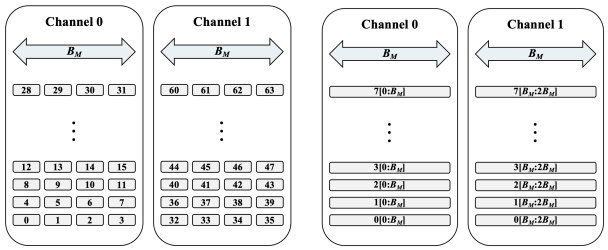
4 Design and Implementation

In this section, we begin by examining how SNARY achieves efficient exact search, providing a more detailed theoretical proof. Subsequently, we discuss the transition from exact search to fuzzy search and analyze the system design in fuzzy search. After that, we present the theoretical formula for the latency of SNARY. Finally, we elaborate on the implementation details of SNARY.

4.1 Exact Search

SNARY stores the large-scale corpus in HBM. HBM achieves high capacity and bandwidth by stacking multiple DDR chips.

[2]In SNARY, a hash bucket only stores the indexes of corpus items for saving memories.



(a) $E < B_M$. (b) $E > B_M$.

Figure 5: The layout of embeddings in HBM for two situations. According to the relationship between E and B_M, embedding concatenation or embedding splitting is performed.

HBM can be divided into multiple channels, each with a maximum transfer bandwidth B_M, meaning that within one clock cycle, each channel can transfer data of up to B_M bytes. Memory read/write operations in each channel do not interfere with each other, and based on this, HBM allows concurrent reading of multiple candidate objects. Let E represent the size of each embedding; if N channels are used to store embeddings from the corpus, the memory bandwidth in Equation 1 can be expressed as:

$$B = N \cdot B_M. \tag{2}$$

To fully utilize the memory bandwidth, a horizontal storage strategy is employed when storing the corpus. This means that the embeddings in the corpus are evenly divided into N parts, ensuring that the data size within each channel is the same. Consequently, all channels read data synchronously in the vertical direction.

Within one clock cycle, HBM can read a maximum of B_M bytes of data in one channel. As a result, the number of complete embeddings N_E read by HBM in one clock cycle is given by the following equation:

$$N_E = \frac{N \cdot B_M}{E}. \tag{3}$$

As discussed, we expect to read complete embeddings in each clock cycle, enabling the overall system to operate in a pipelined fashion. Generally, both E and B_M are powers of 2, so when they are not equal, embedding concatenation or embedding splitting is performed, as shown in Figure 5.

The similarity calculation stage involves assessing the similarity between user queries and candidate objects, which, in the EBR system, is translated into the distance between two embeddings in the embedding space. Therefore, similarity calculation involves computing the distance (e.g., inner product [42], Cosine distance [2] and Euclidean distance [8], etc.) between the embedding representing the user query and the embedding representing the candidate object to obtain a similarity score. A higher similarity score indicates a greater similarity to the user query.

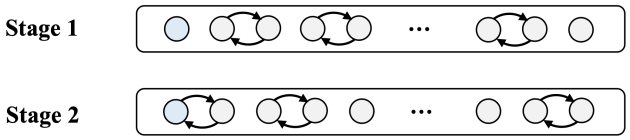


Figure 6: The working principle of the Top-K selection module. It involves two rounds of parallel swaps to maintain the 0th position as the smallest value among the current Top-K elements.

To achieve an efficient processing pipeline, the throughput of the similarity calculation stage needs to match that of the corpus retrieval. In other words, within one clock cycle, the computation of the distance between N_E embeddings and the embedding representing the user query must be completed. As there is no data dependency among these N_E computations, FPGA's data parallelism advantage can be leveraged in the design. N_E similarity computation units are instantiated, with each unit responsible for calculating the similarity score between a candidate object and the user query. Consequently, within one clock cycle, the similarity calculation module receives N_E embeddings and concurrently generates N_E similarity scores.

The Top-K selection stage requires selecting the top K highest scores generated by the similarity calculation module to obtain the final retrieval results. Different retrieval systems may have a varying magnitude of recall quantity [12, 27, 35], so the design of the Top-K selection module needs to satisfy resource constraints and meet performance and scalability goals, when the value of K changes.

The design of the Top-K selection module should support the pipelining of the system. A pipelined retrieval system requires real-time sorting to prevent issues such as pipeline blocking and excessive memory usage. We improved the Top-K sorting algorithm presented in [30] and implemented a Top-K sorting module that maintains pipelined parallelism even with a large number of recall items. We also provide a more detailed and rigorous proof to establish the correctness of the Top-K algorithm. The specific principle is illustrated in Figure 6. The core of the Top-K selection module is an array of size K, which stores the current top K largest values. At any given time, the 0th position of this array holds the smallest value among the current Top-K elements. In each pipeline cycle, a new similarity score is read and compared with the 0th position element. If the new score is larger than the current Top-K's smallest value, a replacement occurs. Subsequently, the array undergoes two rounds of parallel swaps to ensure that the 0th position continues to hold the smallest value in the current Top-K set. The first round of parallel swaps starts from the 1st position and proceeds backward, swapping pairs if they are in reverse order; the second round begins from the 0th position, similarly swapping reverse pairs as necessary. The Top-K selection module generally also needs to ensure

that the results from the retrieval stage are ordered, making them more convenient for processing in the sorting stage. To achieve this, after the final new score is read, we add $K/2$ additional pipeline cycles with empty inputs, during which K rounds of parallel swaps are performed. This guarantees that the scores in the final Top-K array are sorted. The proof of the correctness of the algorithm above can be found in Appendix A.1.

Since the swaps in both Stage 1 and Stage 2 can be performed in parallel, the Top-K selection module can complete its operations with $O(1)$ time complexity for each pipeline cycle. Additionally, each new score is compared with the current smallest value in the Top-K array (i.e., the 0th position), ensuring that the array always contains the top K largest scores at any given time. To maintain the efficiency of parallel swaps when the value of K is large, we partition the Top-K array and distribute it across the on-chip memory to increase the number of read/write ports. In scenarios with limited on-chip resources, we also offer a solution that divides the Top-K array into upstream and downstream segments. This approach splits the array into two sections of size $K/2$, with the first half storing the top $K/2$ values and the second half storing values ranked between $K/2$ and K. The upstream array receives the new scores, determines whether to replace any value and performs parallel swaps. The replaced value (or the non-replaced new score) is then passed to the downstream array, where a similar process is executed.

As stated previously, the performance of the Top-K selection module has a significant impact on the overall performance of the retrieval system. Compared with the Top-K selection module algorithm adopted in [51] (first proposed in [35]), the above-mentioned Top-K selection algorithm based on parallel exchange only occupies $O(K)$ on-chip memory, while the algorithm in [51] requires approximately $O(9K/2)$ of memory. This order-of-magnitude memory footprint optimization yields substantial performance improvements, since under a large recall count, the on-chip resources are almost fully occupied (mainly including the memory occupied by the Top-K module and the filter module). Note that with on-chip memory pressure, the wiring between the memory module and other logic modules will become complicated, which may lead to problems such as wiring congestion and tight logic resources, thereby reducing the working frequency. At the same time, the above-mentioned algorithm only occupies a few (1 or 2) pipeline stages, while the algorithm in [51] has $O(\log K)$ modules and needs to occupy $O(\log K)$ pipeline stages, while each module cannot be reused. Our Top-K selection module is helpful for reducing on-chip memory occupancy, increasing the overall clock frequency, reducing the pipeline depth, and has good scalability at the same time.

Due to the dissimilarity in throughput between the similarity calculation module, which produces N_E scores per clock cycle, and the Top-K selection module, which can only handle one score per clock cycle, a throughput imbalance exists between them. To prevent pipeline blockage, a filter module is introduced between these two modules. The filter module utilizes forward feedback from the Top-K selection module to identify the current minimum score swapped out from the Top-K array and filters out scores that are smaller than the current minimum score. In every pipeline cycle, the filter receives N_E scores, transports the single valid score into the Top-K selection module according to the current minimum Top-K value, and caches or drops other scores.

We provide a detailed analysis of the filtering efficiency and the cache memory usage for other valid scores in the same pipeline cycle in Appendix A.2. Let $\gamma = K/M$ represent the recall rate of the entire retrieval system, the condition under which the filter can be effective is given by:

$$\frac{1}{N_E^2} \geq \gamma - \gamma \ln \gamma, \tag{4}$$

which provides guidance for our system design. For instance, in the case of a recall rate of 1×10^{-3}, we get $N_E \leq 11.25$. Thus, at most 11 candidate objects can be read from the corpus in one clock cycle to make sure the filter works.

4.2 Fuzzy Search

Compared to exact search, fuzzy search reduces search latency by sacrificing accuracy. We implement fuzzy search acceleration in the retrieval process based on the structure of exact search. According to Equation 1, there are three optimization approaches to reduce retrieval system latency and improve performance: (*i*) reducing the size of the corpus, M; (*ii*) increasing the memory bandwidth, B; and (*iii*) decreasing the system's fixed latency, C.

Memory bandwidth is determined by the hardware configuration, while system-fixed latency is a constant related only to system design, accounting for a relatively small portion of the total latency. Therefore, one optional approach is to reduce the size of the corpus, M, to decrease the retrieval system's latency. Specifically, we selected the LSH algorithm, which is more friendly to FPGA, for corpus filtering in the ANN algorithm that accelerates the search by reducing the search scope. Before conducting an exact search, the corpus is filtered to obtain a smaller corpus size, M', and then the exact search is performed on this reduced corpus. This approach maintains the structure of exact search, ensuring high module reusability, and facilitates the flexible switching of system functionality in different scenarios. At this point, according to Equation 1, the theoretical formula for system latency after corpus filtering is:

$$L = \frac{M'}{B} + C. \tag{5}$$

The LSH algorithm achieves dimensionality reduction of target embeddings and partitions the target embedding set into different buckets. In comparison to regular hash algorithms,

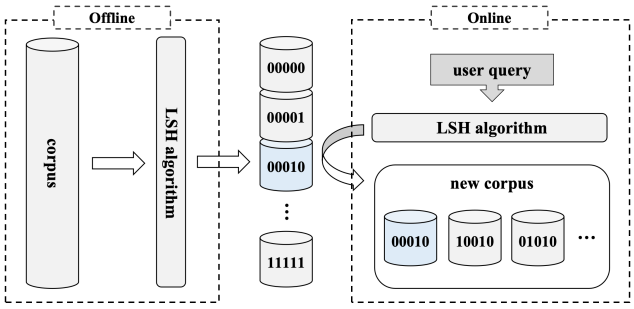


Figure 7: The workflow of corpus filtering. Corpus partitioning happens in the offline phase and the LSH query happens in the online phase.

LSH exhibits a degree of locality sensitivity by locality-sensitive hash functions. Specifically, two embeddings with high similarity in the original space will fall into the same or nearby (Hamming distance) buckets after being mapped by these functions. Conversely, two embeddings with low similarity in the original space will have relatively distant bucket assignments after the LSH mapping. In this way, we can consider the sets of embeddings within the same or nearby buckets as sets of embeddings with high similarity. For embeddings representing user queries, we also employ LSH for dimensionality reduction. We can then treat the bucket where the user queries hit and some neighboring buckets of embeddings in the origin corpus as a new corpus for exact search. This approach allows us to accelerate the search process by filtering out highly similar embeddings in the corpus.

The process of dimensionality reduction and querying through the LSH algorithm involves adjusting three main parameters: K_h, L_h, and T_h, each of which serves a specific purpose:

- K_h represents the number of bits used to represent origin embeddings after dimensionality reduction. In other words, the LSH algorithm maps a high-dimensional embedding to a collection with 2^{K_h} buckets.
- L_h stands for the total number of hash tables in LSH. The LSH algorithm can be configured for multi-table querying, where an embedding can be simultaneously mapped by L_h local sensitive hash functions during a query. This allows searching in L_h tables and identifying L_h buckets where the embedding hits.
- T_h represents the maximum Hamming distance considered when searching for nearby buckets within a single table. To improve search accuracy, LSH can match and hit buckets with Hamming distances smaller than or equal to T_h as a collection of embeddings within the table.

In the case of multi-table querying where $L_h \geq 2$, there is also a choice of query result set intersection or union strategy. The final matching result can either take the intersection or union of the embeddings in the buckets matched in each table.

The necessity of performing intersection (or union) on query results from multiple tables implies that once we have obtained the indices of all matching buckets, we cannot immediately begin pipelined access of embeddings within these buckets, since it is hard to determine whether the current embedding is part of the intersection (or is a duplicate, for the union case). This disrupts the data flow of the system, which will cause performance degradation. Calculating the intersection (or union) before corpus access would introduce significant additional delay and unnecessary memory consumption. Therefore, SNARY adopts a single-table query strategy, meaning $L_h = 1$, adjusting the value of K_h and T_h to control the scale of matched embeddings.

The LSH algorithm we have chosen should approximately distribute the embeddings in the corpus equally among the buckets. Since we expect to search among the bucket where the user queries hit and its neighbors in Hamming distance T_h, we can estimate the size M' of the filtered corpus as follows:

$$M' = \frac{\sum_{i=0}^{T_h} \binom{i}{K_h}}{2^{K_h}} \cdot M. \tag{6}$$

The workflow of our corpus filtering process using the LSH algorithm is illustrated in Figure 7. In the offline phase, we partition the corpus into different buckets based on the LSH algorithm and simultaneously construct hash tables, which is a one-time offline preprocessing phase. That is to say, it does not introduce any extra cost during online queries, since the bucket structure remains static and reusable across all queries, and the execution time of the offline phase remains within a manageable range (approximately 1-2 seconds for a corpus size of 9M). In the online phase, we similarly map user query using LSH and obtain the index of the bucket it falls into. Finally, we use the hit buckets (and neighboring buckets) as the new corpus and perform an exact search within the filtered corpus.

We adopt the widely-used simHash method [3, 34], which reduces a high-dimensional embedding to a low-dimensional signature and the specific process can be divided into the following steps:

- Map each dimension of the high-dimensional embedding to a K_h-bit embedding using a standard hash algorithm.
- Transform the embeddings obtained in the first step based on the weight of each position. Specifically, replace 1 with weight and replace 0 with the additive inverse of weight in each position.
- Accumulate the embeddings in the second step bit-wise.
- Compress the accumulated embedding dimensionally, replacing positive values with 1 and non-positive values with 0.

This process yields a signature representing the bucket number it hits in the hash table. The standard hash function in the first step can be changed to obtain different simHash methods.

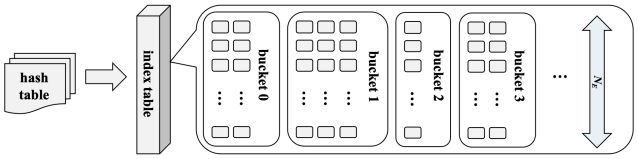


Figure 8: The data structure related to LSH. We store the hash table and index table in HBM, through which we can locate the position of hit embeddings in the corpus.

In this way, we reserve the possibility of constructing different hash tables and performing multi-table querying.

Two available memory channels in HBM are utilized to store data structures related to LSH. Specifically, the hash table and index table are stored in HBM, as illustrated in Figure 8. In the hash table, the starting positions and sizes of each bucket in the index table are recorded, while the index table keeps track of the specific positions of embeddings within each bucket in the corpus. When the bucket where a user query hits is obtained, the hash table is accessed first to mark the regions that need to be accessed in the index table. Afterward, only the elements from the marked buckets in the index table need to be accessed. For each obtained position of an embedding within the corpus, this position can be used to retrieve the specific embedding from the corpus.

During corpus access, we can simultaneously prepare for the next position's retrieval, allowing the system to continue its pipelined operation. However, it's important to note that we still aim to parallelize corpus access, which means accessing N_E embeddings per clock cycle. Due to the uneven distribution of embeddings in the buckets after allocation, it is challenging to determine if N_E continuous embeddings are located in different channels within a bucket. Additionally, it's unclear if accessing N_E positions at once will precisely cover all embeddings within a bucket (since the bucket size may not be a multiple of N_E). To maintain the normal operation of the pipeline, we perform the following steps for each bucket before writing embedding positions to the index table:

- Categorize the embeddings within the bucket into N_E classes based on the channels they are stored in. Let the size of the class with the most embeddings be denoted as S.
- Add paddings to the bucket to reach a size of $S \cdot N_E$ so that, for each class there are S elements including the added paddings.
- Rearrange the elements within the bucket so that, starting from the initial element, every consecutive N_E elements are in different channels. If the elements for a channel have been exhausted, use paddings to fill the gaps.

Through preprocessing, we can achieve parallel access of N_E elements within the current bucket, thereby ensuring a balanced throughput between index table access and corpus access. When we encounter padding elements during the reading, a flagged embedding enters the pipeline. The score generated by this embedding will be filtered out in the filter module, ensuring it doesn't affect the recall results. This optimization strategy increases the pipeline width of fuzzy search to match that of exact search, which significantly improves the efficiency of the fuzzy search process.

4.3 Implementation Details

We implement prototypes of SNARY on the Xilinx AMD Alveo™ U50 Data Center accelerator card of different recall counts K. It features integrated HBM2 (second-generation HBM), which provides up to 400 GB/s of memory bandwidth with 8GB storage and 32 channels for concurrent reads, enabling efficient handling of large datasets and reducing memory bottlenecks. We fully utilize on-chip resources to design SNARY as a completely pipelined structure, supporting seamless switching between exact and fuzzy search modes. We also utilize a 100 Gbps TCP/IP stack [18] to implement corpus uploading, updating, and result transmission on the card. Due to the fact that the FPGA baseline used in our experiment only supports recall counts that are powers of two [51], four systems with recall counts ranging from 512 to 4096 were implemented for comparative experiments.

Corpus access. 16 HBM channels are utilized to store the corpus, providing a total storage capacity of up to 4 GB. Each embedding is stored across 4 contiguous channels, allowing our system to accommodate up to 1024 million embeddings. Since read and write operations among different channels do not interfere with each other, parallel reads are performed in each pipeline cycle to maximize bandwidth. This enables the reading of 4 embeddings at once, which can then proceed to the subsequent computation stages.

Similarity calculation. In the similarity calculation module, the dot product is used to measure the similarity between different embeddings. Data from the contiguous channels belonging to an embedding is computed with the corresponding positions of the user query's embedding, integrating these calculations to form similarity scores. In the computation process, the calculation of similarity scores for different embeddings, as well as the scores for different positions of a single embedding, is fully unrolled, effectively utilizing hardware resources.

Top-K selection. The Top-K selection module adopts the parallel swapping strategy mentioned in Section 4.1 to ensure pipelined parallelism. Specifically, an array of length K is implemented in on-chip memory to store the Top-K scores, with array partitioning used to increase the read/write ports of the Top-K array. For higher recall counts, the array is further divided into upstream and downstream segments to avoid excessive resource consumption within a single pipeline cycle, while maintaining an $O(K)$ byte space usage. After the last score enters the Top-K selection module, additional $K/2$ cycles are added to perform the same parallel swapping and

Table 1: Resource usage and clock frequency of SNARY under different recall counts.

K	resource usage (%)				Freq.(MHz)
	LUT	FF	BRAM	DSP	
512	18.17	12.15	18.42	0.07	302.0
1024	26.41	14.09	18.42	0.07	285.0
2048	34.44	15.97	20.35	0.07	279.0
4096	40.16	24.19	20.35	0.07	235.3

sorting operations to ensure that the final Top-K array is completely ordered.

Filter. The filter is implemented using four FIFO pipelines with a certain depth to balance the throughput between the similarity calculation and Top-K selection modules. These FIFO pipelines receive scores from the similarity calculation and filter out or temporarily cache[3] some scores based on the current Top-K minimum value feedback from the Top-K selection module. They then provide one score to the Top-K selection module. To handle burst scenarios, the depth of each FIFO pipeline is set to accommodate the storage of $2K$ scores according to the analysis in Appendix A.2.

Fuzzy search. Two idle HBM channels are used to store the hash table and index table. Upon the arrival of user queries, the index table, which has already been preprocessed, allows for the parallel reading of the corresponding channels for every 4 indices, with each embedding stored in a different channel. This enables the reading of the corresponding positions in the next pipeline cycle while retrieving the 4 indices without impacting the overall pipelining of the system. The parameters K_h and T_h in the LSH algorithm are set as adjustable parameters, allowing users to control the balance between result accuracy and retrieval speed.

Batch support. SNARY implements batch support through multiple computing units on a single card to further enhance throughput under low-latency conditions. These computing units share the same HBM and possess identical architecture, allowing them to work in parallel coordination. Due to on-chip resource limitations, we achieved a batch size of 3 for systems with recall counts of 512 and 1024, a batch size of 2 for a recall count of 2048, and a batch size of 1 for a recall count of 4096. Furthermore, for increasing batch sizes beyond these values, multi-card coordination may provide additional opportunities for improvement.

Memory Overhead. (*i*) Corpus size. Both exact and fuzzy search operations utilize the HBM external memory to store the corpus. In our experiments, we place up to 9M embeddings in HBM, which occupies about 28% external memories (totaling 32M embeddings). (*ii*) On-chip memory. The hardware implementation for both exact and fuzzy search with the same recall count is the same. We fully utilized the limited 8MB on-chip memory to implement SNARY, and the resource consumption and the clock frequency of SNARY under different recall counts is shown in Table 1.

[3]Caching occurs when multiple scores exceed the current minimum value within one clock cycle.

5 Evaluation

We evaluate the performance of SNARY on both exact and fuzzy search scenarios and compare it with a GPU-based retrieval system Faiss and an FPGA-based retrieval system FAERY. For exact search, we evaluate the delay and the latency-bounded throughput of SNARY, Faiss, and FAERY. For fuzzy search, we first evaluate the efficiency performance of SNARY and Faiss under similar accuracy performance. We also conducted a series of self-comparison experiments to explore the impact of parameter changes on the trade-off between efficiency and accuracy. All error bars in the figures represent the standard error of the mean (SE). Our experimental results show that:

- SNARY achieves a 78.75%-83.88% lower query latency than Faiss and a 20.91%-45.19% lower query latency than FAERY, with a 14.12×-18.27× higher latency-bounded throughput than Faiss and a 1.26×-1.64× higher latency-bounded throughput than FAERY in exact search scenarios.
- SNARY achieves a 85.13%-87.40% lower query latency than Faiss, with a 20.18×-23.81× higher latency-bounded throughput than Faiss in fuzzy search scenarios, while FAERY does not support fuzzy search.
- SNARY allows for the adjustment of relevant parameters to achieve a trade-off between efficiency and accuracy, thereby better meeting various retrieval requirements in fuzzy search scenarios.

5.1 Experimental Setup

Baseline. We use Faiss [25] and FAERY [51] as the GPU-based and FPGA-based baseline in our experiments. Faiss is evaluated on a server equipped with 4 Nvidia A100 GPUs, each with 12GB of VRAM, running CUDA version 12.0. FAERY is implemented and evaluated on the Xilinx AMD Alveo™ U50 Data Center accelerator card, with the same HBM utilization configuration as SNARY. A recall count of 1024 was used as the default configuration in our experiment because FAERY implements a system specifically for this recall count [51], and it also represents the maximum recall count in Faiss. FAERY systems with the same batch size as SNARY under different recall counts were also implemented to conduct comparative experiments with SNARY.

Corpus. SNARY is designed as a generic retrieval system that is not sensitive to any specific workload. Thus, we follow the methodologies of [51] and use synthetic datasets to evaluate the generality and efficiency of SNARY. This is because SNARY focuses on how to improve the retrieval performance

via designing a search engine architecture on SmartNICs. Within this architecture, the retrieval efficiency is mainly decided by the corpus access, which is a hardware-bound process independent of dataset semantics. Specifically, the corpus consists of randomly generated 128-dimensional embeddings with 1 byte each dimension. Since each channel in the HBM provides 32B bandwidth, we utilize 4 consecutive adjacent channels to store a complete embedding.

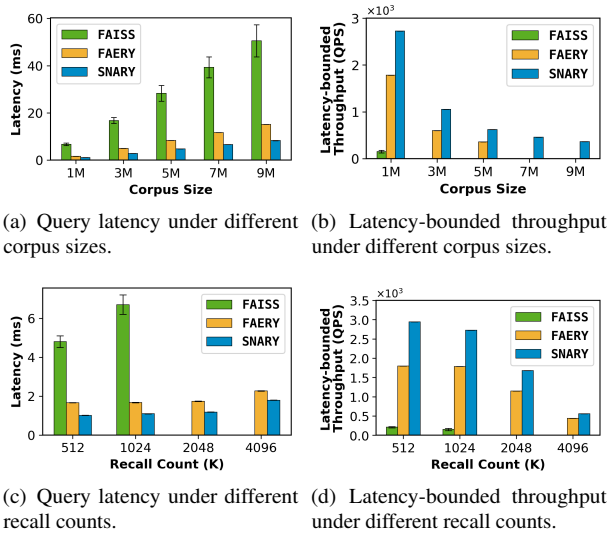
5.2 Evaluation on Exact Search

We measured the query latency and the latency-bounded throughput among different systems in exact search scenarios. Latency-bounded throughput refers to the number of queries a system can process per second (QPS) while maintaining a strict upper limit on response time, or latency. This metric is crucial in real-time retrieval systems, where rapid and efficient data processing is essential to meet user expectations and service requirements. In such systems, high throughput must be achieved without exceeding latency constraints, ensuring both speed and quality of service. In our experiments, we adopted the widely recognized upper latency limit of 10ms as a standard benchmark [4, 10, 27], ensuring the system's ability to handle queries efficiently within this time frame. The experimental results are shown in Figure 9.

We first set the recall count to 1024 and varied the corpus size to compare query latency and latency-bounded throughput among different systems. The experimental results are shown in Figure 9(a) and Figure 9(b). As can be seen, SNARY's latency exhibits a linear relationship with corpus size, which is consistent with the latency formula presented in Equation 1. SNARY achieves a 82.42%-83.63% lower query latency than Faiss and a 34.52%-45.19% lower query latency than FAERY. Under the 10ms latency constraint, SNARY achieves a 18.27× higher throughput than Faiss and a 1.52×-1.76× higher throughput than FAERY, and continues to function even when the other systems fail to meet the latency constraint.

We also fixed the corpus size at 1M and varied the recall count to compare the performance of the different systems. The results are shown in Figure 9(c) and Figure 9(d). As the recall count increases, the performance of all systems decreases, and SNARY obtains more stable performance compared with FAERY as the system architecture is not made more complex. Notably, Faiss was unable to function under higher recall counts due to a recall count limitation of 1024. SNARY achieves a 78.75%-83.88% lower query latency than Faiss and a 20.91%-38.85% lower query latency than FAERY. Under the 10ms latency constraint, SNARY achieves a 14.12× -18.27× higher throughput than Faiss and a 1.26×-1.64× higher throughput than FAERY, and continues to operate under higher recall counts.

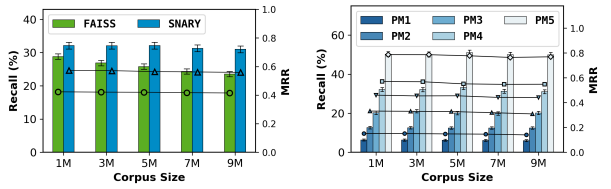
The experimental results show that in terms of both the query latency and the latency-bounded throughput, the performance of SNARY is superior to that of Faiss and FAERY. This is because, compared with GPU systems, the inherent hardware advantages of FPGA (mentioned in Section 2.1) enable the retrieval system to be fully pipelined, reducing the retrieval latency, ensuring that the system can still work properly under relatively high recall counts. Meanwhile, since the single-query latency of GPU is relatively high, it is easy to exceed the latency limit on higher batch sizes. However, SNARY adopts a fully parallelized batch support, ensuring high throughput under low latency. Compared with FAERY, the improvements made by SNARY in the Top-K selection module algorithm (mentioned in Section 4.1) are helpful in reducing on-chip resource occupation, increasing the clock frequency, and decreasing the pipeline depth, thus achieving better results.



(a) Query latency under different corpus sizes.
(b) Latency-bounded throughput under different corpus sizes.
(c) Query latency under different recall counts.
(d) Latency-bounded throughput under different recall counts.

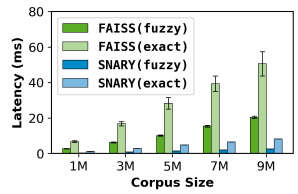
Figure 9: The comparison of query latency and latency-bounded throughput among different systems in exact search as corpus size and recall count varies.

The experimental results also indicate that, compared to Faiss, both SNARY and FAERY, which are FPGA-based implementations, exhibit significantly lower latency jitter. Their latency and latency-bounded throughput remain consistently stable (as seen in Figure 10). This is mainly due to the fact that FPGA can achieve deterministic processing logic through customized hardware circuits, thereby avoiding performance jitter. In contrast, GPUs may experience fluctuations during concurrent processing due to factors such as task scheduling and concurrent memory access. The FPGA's dedicated hardware design ensures stable performance, while the GPU's shared resource architecture introduces variability in execution times. It also can be seen that increasing the recall count results in performance degradation. Because higher recall counts significantly raise the resource occupancy of the Top-K module, which is the bottleneck in GPU-based systems

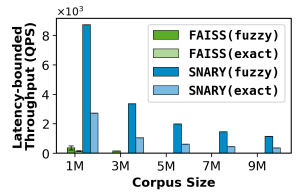


(a) Recall (bars) and MRR (line) between Faiss and SNARY.

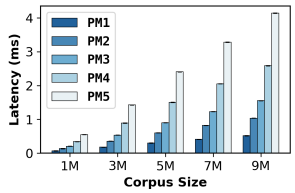
(b) Recall (bars) and MRR (line) among different parameter combinations of SNARY.



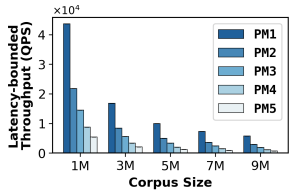
(c) Query latency between Faiss and SNARY.



(d) Latency-bounded throughput between Faiss and SNARY.



(e) Query latency among different parameter combinations.



(f) Latency-bounded throughput among different parameter combinations.

Figure 10: The comparison of query latency, latency-bounded throughput and accuracy among different systems in fuzzy search as corpus size varies. The corresponding metrics for the exact search case are also shown for reference in the comparison of SNARY and Faiss.

(discussed in Section 2.1) and dominates the on-chip memory consumption for FPGA-based systems.

5.3 Evaluation on Fuzzy Search

We measured the performance between SNARY and Faiss in fuzzy search scenarios under various corpus with a recall count of 1024 where Faiss functions normally. We used the *IndexIVFFlat* method of Faiss as the baseline, which is the most popular ANN algorithm in Faiss. It accelerates the search by partitioning the dataset into segments, where Voronoi cells are defined in the d-dimensional space, and each database vector is assigned to one cell. During the search, only the vectors in the query's cell and a few neighboring ones are compared to the query vector. It has two tunable parameters, *nlist* and *probe*, which determine the segment number and the search range respectively to balance latency and accuracy. We used the recommended configuration with $nlist = 100$ and $nprobe = 10$.

For the comparison experiment, we first adjusted our own ANN parameters to ensure that the two systems have similar accuracy when comparing latency. We select the parameter combination of $K_h = 4$ and $T_h = 1$ to participate in the comparative experiment. First, we measured the retrieval result accuracy of SNARY and Faiss under different corpus sizes. We use Recall and MRR (Mean Reciprocal Rank) to gauge the accuracy of fuzzy search. In our experimental scenario, Recall represents the ratio of fuzzy retrieval results to exact retrieval results, while MRR refers to the mean of the inverse of the ranking of the first correct result returned in multiple queries. The experimental results are shown in Figure 10(a). It can be seen that when the corpus size changes, the Recall and MRR values of SNARY are similar and slightly higher than those of Faiss. As corpus size increases, the accuracy of SNARY maintains a stable range.

We then measured the query latency and the latency-bounded throughput (with the same upper limit of 10ms) between the two systems and the experimental results are shown in Figure 10(c) and Figure 10(d) with results in exact search scenarios as references. From the experimental results, SNARY successfully reduces query latency in fuzzy search compared with exact search and also outperforms Faiss in terms of latency and latency-bounded throughput under similar accuracy metrics. SNARY achieves a 85.13%-87.40% lower latency and a 20.18×-23.81× higher throughput under the 10ms constraint than Faiss at similar accuracy of searching results. Compared with exact search, SNARY reduces the query latency by 78.05%-78.72%, while Faiss reduces that by 35.41%-39.17%, which also makes the latency-bounded throughput corresponding to the fuzzy search higher than that of the exact search.

The performance advantages of SNARY in fuzzy search come from two aspects: (*i*) high-performance exact search; (*ii*) high-efficiency acceleration effect. Due to the hardware architecture advantages of SNARY, its performance in terms of query latency and latency-bounded throughput in exact search is already better than that of GPU-based systems (mentioned in Section 5.2). Moreover, SNARY adopts the method of directly fuzzifying the reading of the corpus. For a fully pipelined parallel retrieval system, this can directly reduce the delay, and in the scenarios of large-scale corpus, it can very efficiently accelerate the entire retrieval process (mentioned in Section 2.2).

Finally, we investigated the impact of parameter changes on the trade-off between latency and accuracy. As indicated in Equation 6, the reduction in corpus size and consequently, the overall system latency can be controlled by selecting appropriate values of K_h and T_h. We use η to represent the corpus reduction ratio, defined as follows:

$$\eta = \frac{\sum_{i=0}^{T_h} \binom{i}{K_h}}{2^{K_h}}. \tag{7}$$

We selected various parameter combinations, as shown in Table 2, to assess the performance of the fuzzy search. The

Table 2: Performance in fuzzy search of different systems.

metric	parameter combination				
	param1	param2	param3	param4	param5
K_h	4	3	5	4	3
T_h	0	0	1	1	1
$\eta(\%)$	6.25	12.50	18.75	31.25	50.00

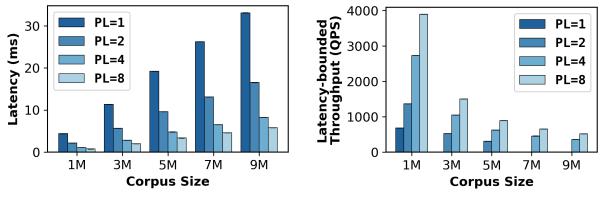
identities of different systems are arranged in ascending order based on the values of η.

We measured the query latency, latency bounded-throughput (with the same upper limit of 10ms), and accuracy of SNARY under different parameter combinations, with a recall count of 1024, the experimental results are shown in Figure 10(b), Figure 10(e) and Figure 10(f). From the experimental results, as the parameter η decreases, the proportion of corpus reduction increases, leading to a decrease in the query latency of SNARY and an improvement in latency-bounded throughput. Conversely, as η increases, the search range expands, resulting in improved precision and higher-quality retrieval results. Additionally, across different corpus sizes, SNARY consistently demonstrates stable acceleration performance, with the latency reduction compared with exact search remaining around $1-\eta$, which is consistent with the corpus reduction ratio in Equation 6.

This also indicates that we can balance retrieval efficiency and accuracy by adjusting the values of K_h and T_h. In scenarios where lower retrieval latency and higher throughput are prioritized over accuracy, a smaller η value can be used, whereas in scenarios requiring higher retrieval accuracy, a larger η value is more appropriate. Additionally, users can directly predict the latency reduction and the increase in latency-bounded throughput in a fuzzy retrieval based on the parameter combination, allowing for more flexible and efficient retrieval in different application contexts.

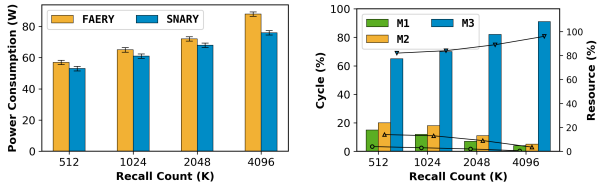
5.4 Extended System Characterization

Impact of parallel corpus access. As briefly stated in Section 4.1, parallel corpus access improves system efficiency by processing multiple embeddings simultaneously per clock cycle, compared to sequential retrieval. To validate its performance impact, we tested four parallelism levels, using exact search task with recall count 1024 and measured query latency and latency-bounded throughput under different parallelism levels. Experimental results are shown in Figure 11(a) and Figure 11(b). It can be observed that system performance demonstrates significant improvements with increasing parallelism degrees. At the higher parallelism level of 8, the performance gain becomes less pronounced due to the excessive number of embeddings processed per clock cycle. These results confirm that parallel corpus access directly enhances both critical performance metrics of the system.



(a) Query latency among different parallelism levels.

(b) Latency-bounded throughput among different parallelism levels.



(c) Energy consumption between FAERY and SNARY.

(d) Cycle and resource allocation among modules in SNARY.

Figure 11: The experimental results of parallel levels, SmartNIC energy consumption and module performance analysis.

SmartNIC energy consumption. We measured the power consumption of both FAERY and SNARY implemented on identical hardware boards. Experimental results indicate no significant correlation between corpus size and power draw. As depicted in Figure 11(c), increasing Recall Count elevates system complexity, leading to corresponding increases in power consumption. Notably, SNARY demonstrates superior power efficiency compared to FAERY, attributable to its optimized Top-K module design.

Module performance analysis. We measured and analyzed the clock cycle allocation ratios and resource utilization patterns using Vitis across SNARY's modules under varying recall counts, with detailed data presented in Figure 11(d), where corpus access, similarity calculation and Top-K module (with filter) are denoted as M1, M2 and M3, respectively. The results demonstrate that the Top-K module dominates both temporal overhead and hardware resource usage among all components. This finding underscores the critical importance of Top-K module optimization for overall system performance enhancement.

6 Discussion

HBM utilization. Our experiments use 2 channels to store the hash table and index table, and 1 channel for write-back operations. The remaining 29 channels can be fully utilized to store the corpus. This allows for the possibility of reading up to 7 embeddings at once when using 4 contiguous channels per embedding. From inequation 4, we know that the parallelism metric N_E is constrained by the recall rate γ. Under our experimental conditions, the maximum recall rate $\gamma = 1/250$, which yields $N_E \leq 6.19$, indicating that our system can achieve par-

allelism of $N_E = 6$ while utilizing 24 HBM channels. In more general conditions, a recall rate of less than 3×10^{-3} ensures that 7 channels can be simultaneously utilized. Therefore, our system can further reduce latency by utilizing more HBM channels for higher total bandwidth, potentially decreasing it by up to around 43%. When utilizing multiple channels, a limited corpus may result in significant vertical gaps in the HBM. To improve HBM utilization, it is possible to partition HBM vertically into multiple segments, allowing the empty sections to be used for storing additional corpora for other retrieval tasks or to hold a backup corpus for online updates. If fewer channels are used and the corpus involves various data types (such as text, images, and audio), the remaining channels can be allocated for storing multimodal data features.

Early corpus filtering. As discussed in Section 2.1, the performance of SNARY is directly correlated with corpus size, and our fuzzy search solution improves operational efficiency by reducing the corpus size. However, this still requires storing all data on the servers. Future work will focus on implementing early corpus filtering to preemptively remove irrelevant corpus portions based on user requirements or business types. This approach will store only task-related corpora on servers before initiating read operations. Such an implementation is expected to both alleviate server storage burdens and further enhance system efficiency.

Multi-card cooperation. SNARY implements batch support by instantiating multiple on-chip computing units, which will undoubtedly be limited by on-chip resources. For example, when the recall count increases, the batch size supported by the system becomes smaller, and multiple computing units may also bring wiring difficulties, slow timing convergence, and other problems. A feasible approach for higher latency-bounded throughput is multi-card cooperation. This collaborative approach allows for distributing the processing load across several cards, enhancing concurrency and further optimizing performance. The integration of SmartNICs facilitates efficient and low-latency communication between multiple cards, ensuring that the data transfer necessary for batch processing occurs seamlessly and swiftly.

7 Related Work

Deep retrievals. Recently, many different neural models have been proposed to address semantic gap problems, which fall mainly into two categories: representation-based learning [1] and interaction-based learning [9]. For representation-based learning, many DL models (e.g., LSTM-RNN [37], ARC-I [19] and DSSM [21]) are utilized to generate semantic representations of corpus items. Interaction-based approaches learn the complex relevant patterns between the query and items via neural models, such as DRMM [16], Match-SRNN [46], MatchPyramid [38] and K-NRM [50]. SNARY utilizes the semantic representations generated by neural networks to calculate their similarity.

Hardware-accelerated retrieval systems. To improve retrieval performance, some specific hardware accelerators have been integrated into modern retrieval systems. For example, some works utilize GPUs to accelerate similarity calculation [11, 25] or CXL (compute express link) [23] to improve the corpus access. Recently, FPGAs also have been introduced to improve performance [26, 51]. SNARY mitigates the gap and enables both exact and fuzzy search, adapts an optimized Top-K algorithm and involves SmartNIC-centric deployment compared with [51].

Accelerating retrievals with ANN. Besides hardware acceleration, the community also proposes some ANN algorithms to accelerate the retrieval. For example, IVF [44] utilizes an inverted index approach to store vectors in order to reduce the number of accessed items. HNSW [32] employs a hierarchical NSW [33] index graph to conduct an approximate nearest neighbor search. PQ [24] and OPQ [13], using product quantization, reduce the computational cost of distance calculations. For SNARY, it utilizes LSH [7, 8, 45] to reduce the scale of the corpus.

Efficient Top-K algorithms. Although many efficient Top-K methods have been implemented on GPUs [40, 41], the overhead of inter-core communication still presents a bottleneck for these algorithms. [51] introduced a FIFO-based algorithm on FPGA. However, it occupies $O(9K/2)$ memory and exhibits limited scalability with varying recall counts. [30] proposed a parallel swapping algorithm for smaller recall counts with $O(K)$ memory. SNARY presents a strategy for implementing parallel swapping algorithms under larger recall counts while providing detailed theoretical proofs.

8 Conclusion

In this paper, we design and implement SNARY, a high-performance and generic retrieval system. SNARY is built upon SmartNIC and enables both exact and fuzzy retrievals. Specifically, SNARY leverages HBM on FPGAs to store a large-scale corpus. With this basis, it designs a data parallelism exact search engine and an LSH-based fuzzy search engine. To achieve high performance, SNARY further employs a parallel pipelined method to facilitate the Top-K item selection and stream the data flow of the whole system. Experimental results show that SNARY significantly outperforms the popular GPU-based solutions in both exact and fuzzy scenarios and outperforms the novel FPGA-based solutions in exact scenarios which fall short in fuzzy scenarios.

Acknowledgments

This work is supported by the National Natural Science Foundation of China (Grant No. 62472404). Heng Pan is the corresponding author.

References

[1] Yoshua Bengio, Aaron Courville, and Pascal Vincent. Representation learning: A review and new perspectives. *IEEE transactions on pattern analysis and machine intelligence*, 35(8):1798–1828, 2013.

[2] Moses S Charikar. Similarity estimation techniques from rounding algorithms. In *Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*, pages 380–388, 2002.

[3] Beidi Chen, Zichang Liu, Binghui Peng, Zhaozhuo Xu, Jonathan Lingjie Li, Tri Dao, Zhao Song, Anshumali Shrivastava, and Christopher Re. Mongoose: A learnable lsh framework for efficient neural network training. In *International Conference on Learning Representations*, 2020.

[4] Heng-Tze Cheng, Levent Koc, Jeremiah Harmsen, Tal Shaked, Tushar Chandra, Hrishi Aradhye, Glen Anderson, Greg Corrado, Wei Chai, Mustafa Ispir, et al. Wide & deep learning for recommender systems. In *Proceedings of the 1st workshop on deep learning for recommender systems*, pages 7–10, 2016.

[5] Paul Covington, Jay Adams, and Emre Sargin. Deep neural networks for youtube recommendations. In *Proceedings of the 10th ACM conference on recommender systems*, pages 191–198, 2016.

[6] Valerie Cross. Fuzzy information retrieval. *Journal of Intelligent Information Systems*, 3:29–56, 1994.

[7] Anirban Dasgupta, Ravi Kumar, and Tamás Sarlós. Fast locality-sensitive hashing. In *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1073–1081, 2011.

[8] Mayur Datar, Nicole Immorlica, Piotr Indyk, and Vahab S Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *Proceedings of the twentieth annual symposium on Computational geometry*, pages 253–262, 2004.

[9] Suchana Datta, Debasis Ganguly, Derek Greene, and Mandar Mitra. Deep-qpp: A pairwise interaction-based deep learning model for supervised query performance prediction. In *Proceedings of the Fifteenth ACM International Conference on Web Search and Data Mining*, pages 201–209, 2022.

[10] Cong Fu, Chao Xiang, Changxu Wang, and Deng Cai. Fast approximate nearest neighbor search with the navigating spreading-out graph. *arXiv preprint arXiv:1707.00143*, 2017.

[11] Luyu Gao, Xueguang Ma, Jimmy Lin, and Jamie Callan. Tevatron: An efficient and flexible toolkit for dense retrieval. *arXiv preprint arXiv:2203.05765*, 2022.

[12] Weihao Gao, Xiangjun Fan, Chong Wang, Jiankai Sun, Kai Jia, Wenzhi Xiao, Ruofan Ding, Xingyan Bin, Hui Yang, and Xiaobing Liu. Deep retrieval: Learning a retrievable structure for large-scale recommendations. *arXiv preprint arXiv:2007.07203*, 2020.

[13] Tiezheng Ge, Kaiming He, Qifa Ke, and Jian Sun. Optimized product quantization. *IEEE transactions on pattern analysis and machine intelligence*, 36(4):744–755, 2013.

[14] Kira Gor, Svetlana Cook, Denisa Bordag, Anna Chrabaszcz, and Andreas Opitz. Fuzzy lexical representations in adult second language speakers. *Frontiers in Psychology*, 12:732030, 2021.

[15] Mihajlo Grbovic and Haibin Cheng. Real-time personalization using embeddings for search ranking at airbnb. In *Proceedings of the 24th ACM SIGKDD international conference on knowledge discovery & data mining*, pages 311–320, 2018.

[16] Jiafeng Guo, Yixing Fan, Qingyao Ai, and W Bruce Croft. A deep relevance matching model for ad-hoc retrieval. In *Proceedings of the 25th ACM international on conference on information and knowledge management*, pages 55–64, 2016.

[17] Anshita Gupta, Sanya Bathla Taneja, Garima Malik, Sonakshi Vij, Devendra K Tayal, and Amita Jain. Slangzy: A fuzzy logic-based algorithm for english slang meaning selection. *Progress in Artificial Intelligence*, 8:111–121, 2019.

[18] Z. He, D. Korolija, and G. Alonso. Easynet: 100 gbps network for hls. In *2021 31st International Conference on Field-Programmable Logic and Applications (FPL)*, pages 197–203, Los Alamitos, CA, USA, sep 2021. IEEE Computer Society.

[19] Baotian Hu, Zhengdong Lu, Hang Li, and Qingcai Chen. Convolutional neural network architectures for matching natural language sentences. *Advances in neural information processing systems*, 27, 2014.

[20] Jui-Ting Huang, Ashish Sharma, Shuying Sun, Li Xia, David Zhang, Philip Pronin, Janani Padmanabhan, Giuseppe Ottaviano, and Linjun Yang. Embedding-based retrieval in facebook search. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 2553–2561, 2020.

[21] Po-Sen Huang, Xiaodong He, Jianfeng Gao, Li Deng, Alex Acero, and Larry Heck. Learning deep structured semantic models for web search using clickthrough data. In *Proceedings of the 22nd ACM international conference on Information & Knowledge Management*, pages 2333–2338, 2013.

[22] Amita Jain and Charu Gupta. Fuzzy logic in recommender systems. *Fuzzy Logic Augmentation of Neural and Optimization Algorithms: Theoretical Aspects and Real Applications*, pages 255–273, 2018.

[23] Junhyeok Jang, Hanjin Choi, Hanyeoreum Bae, Seungjun Lee, Miryeong Kwon, and Myoungsoo Jung. {CXL-ANNS}:{Software-Hardware} collaborative memory disaggregation and computation for {Billion-Scale} approximate nearest neighbor search. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 585–600, 2023.

[24] Herve Jegou, Matthijs Douze, and Cordelia Schmid. Product quantization for nearest neighbor search. *IEEE transactions on pattern analysis and machine intelligence*, 33(1):117–128, 2010.

[25] Jeff Johnson, Matthijs Douze, and Hervé Jégou. Billion-scale similarity search with gpus. *IEEE Transactions on Big Data*, 7(3):535–547, 2019.

[26] Luyang Li, Heng Pan, Xinchen Wan, Kai Lv, Zilong Wang, Qian Zhao, Feng Ning, Qingsong Ning, Shideng Zhang, Zhenyu Li, et al. Harmonia: A unified framework for heterogeneous fpga acceleration in the cloud. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 498–514, 2025.

[27] Sen Li, Fuyu Lv, Taiwei Jin, Guli Lin, Keping Yang, Xiaoyi Zeng, Xiao-Ming Wu, and Qianli Ma. Embedding-based product retrieval in taobao search. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*, pages 3181–3189, 2021.

[28] Xiang Li, Xiaojiang Zhou, Yao Xiao, Peihao Huang, Dayao Chen, Sheng Chen, and Yunsen Xian. Autofas: Automatic feature and architecture selection for pre-ranking system. In *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 3241–3249, 2022.

[29] Shaopei Lin. Fuzzy machine learning methods. In *Fuzzy-AI Model and Big Data Exploration: A Methodological Philosophy in Solving Problems in Digital Era*, pages 117–172. Springer, 2022.

[30] Alec Lu, Zhenman Fang, Nazanin Farahpour, and Lesley Shannon. Chip-knn: A configurable and high-performance k-nearest neighbors accelerator on cloud fpgas. In *2020 International Conference on Field-Programmable Technology (ICFPT)*, pages 139–147. IEEE, 2020.

[31] Pasi Luukka. Fuzzy similarity in multicriteria decision-making problem applied to supplier evaluation and selection in supply chain management. *Advances in Artificial Intelligence*, 2011, 2011.

[32] Yu A Malkov and Dmitry A Yashunin. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE transactions on pattern analysis and machine intelligence*, 42(4):824–836, 2018.

[33] Yury Malkov, Alexander Ponomarenko, Andrey Logvinov, and Vladimir Krylov. Approximate nearest neighbor algorithm based on navigable small world graphs. *Information Systems*, 45:61–68, 2014.

[34] Gurmeet Singh Manku, Arvind Jain, and Anish Das Sarma. Detecting near-duplicates for web crawling. In *Proceedings of the 16th international conference on World Wide Web*, pages 141–150, 2007.

[35] Naoyuki Matsumoto, Koji Nakano, and Yasuaki Ito. Optimal parallel hardware k-sorter and top k-sorter, with fpga implementations. In *2015 14th International Symposium on Parallel and Distributed Computing*, pages 138–147. IEEE, 2015.

[36] Priyanka Nigam, Yiwei Song, Vijai Mohan, Vihan Lakshman, Weitian Ding, Ankit Shingavi, Choon Hui Teo, Hao Gu, and Bing Yin. Semantic product search. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 2876–2885, 2019.

[37] Hamid Palangi, Li Deng, Yelong Shen, Jianfeng Gao, Xiaodong He, Jianshu Chen, Xinying Song, and Rabab Ward. Deep sentence embedding using long short-term memory networks: Analysis and application to information retrieval. *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, 24(4):694–707, 2016.

[38] Liang Pang, Yanyan Lan, Jiafeng Guo, Jun Xu, Shengxian Wan, and Xueqi Cheng. Text matching as image recognition. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 30, 2016.

[39] Dong Qiu, Haihuan Jiang, and Shuqiao Chen. Fuzzy information retrieval based on continuous bag-of-words model. *Symmetry*, 12(2):225, 2020.

[40] Anil Shanbhag, Holger Pirk, and Samuel Madden. Efficient top-k query processing on massively parallel hardware. In *Proceedings of the 2018 International Conference on Management of Data*, pages 1557–1570, 2018.

[41] Jieming Shi, Renchi Yang, Tianyuan Jin, Xiaokui Xiao, and Yin Yang. Realtime top-k personalized pagerank over large graphs on gpus. *Proceedings of the VLDB Endowment*, 13(1):15–28, 2019.

[42] Anshumali Shrivastava and Ping Li. Asymmetric lsh (alsh) for sublinear time maximum inner product search (mips). *Advances in neural information processing systems*, 27, 2014.

[43] Jagendra Singh and Aditi Sharan. A new fuzzy logic-based query expansion model for efficient information retrieval using relevance feedback approach. *Neural Computing and Applications*, 28:2557–2580, 2017.

[44] Sivic and Zisserman. Video google: A text retrieval approach to object matching in videos. In *Proceedings ninth IEEE international conference on computer vision*, pages 1470–1477. IEEE, 2003.

[45] Malcolm Slaney and Michael Casey. Locality-sensitive hashing for finding nearest neighbors [lecture notes]. *IEEE Signal processing magazine*, 25(2):128–131, 2008.

[46] Shengxian Wan, Yanyan Lan, Jun Xu, Jiafeng Guo, Liang Pang, and Xueqi Cheng. Match-srnn: Modeling the recursive matching structure with spatial rnn. *arXiv preprint arXiv:1604.04378*, 2016.

[47] Jizhe Wang, Pipei Huang, Huan Zhao, Zhibo Zhang, Binqiang Zhao, and Dik Lun Lee. Billion-scale commodity embedding for e-commerce recommendation in alibaba. In *Proceedings of the 24th ACM SIGKDD international conference on knowledge discovery & data mining*, pages 839–848, 2018.

[48] Yan Wang, Peng Jia, Luping Liu, Cheng Huang, and Zhonglin Liu. A systematic review of fuzzing based on machine learning techniques. *PloS one*, 15(8):e0237749, 2020.

[49] Long Xiang, Bo Tang, and Chuan Yang. Accelerating exact inner product retrieval by cpu-gpu systems. In *Proceedings of the 42nd International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 1277–1280, 2019.

[50] Chenyan Xiong, Zhuyun Dai, Jamie Callan, Zhiyuan Liu, and Russell Power. End-to-end neural ad-hoc ranking with kernel pooling. In *Proceedings of the 40th International ACM SIGIR conference on research and development in information retrieval*, pages 55–64, 2017.

[51] Chaoliang Zeng, Layong Luo, Qingsong Ning, Yaodong Han, Yuhang Jiang, Ding Tang, Zilong Wang, Kai Chen, and Chuanxiong Guo. FAERY: An FPGA-accelerated embedding-based retrieval system. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 841–856, 2022.

A The Theoretical Proof and Analysis

A.1 Proof of the Correctness of Top-K Module

We now prove that the algorithm in Top-K selection module ensures the 0th position always holds the smallest value in the current Top-K array, and that the entire array can be ordered with at most $K/2$ additional parallel cycles.

- First, when array length $K = 2$, the Stage 1 swap process effectively does nothing, while Stage 2 directly compares the two values in the array and moves the smaller value to the 0th position, confirming that the algorithm works correctly.
- Next, assume that our algorithm holds for an array of length $K = k-1$. We now consider an array of length $K = k$ and divide it into two parts: the 0th position and the remaining $k-1$ elements, denoted as array A. It is important to note that for any pipeline cycle i, Stage 2 of cycle i and Stage 1 of cycle $(i+1)$, when applied to array A, effectively perform the two rounds of parallel swaps for $K = k-1$. In Stage 2 of cycle i, the new score (in the 0th position) is compared and swapped with the 0th position of array A, followed by a parallel sort of array A starting from the 1st position. Stage 1 of cycle $i+1$ then performs a parallel sort starting from the 0th position of array A.

  Since our algorithm holds for array A with length $K = k-1$, after Stage 1 of cycle $(i+1)$, the 0th position of array A contains the smallest value of array A, which means that the 1st position of the original array is the smallest value among the last $(k-1)$ elements. After Stage 2 of cycle $(i+1)$, the 0th and 1st positions of the original array are compared: if the 0th position is smaller, it remains as the minimum value, and no swap occurs; if the 0th position is larger, a swap occurs, making the 1st position the smallest value in the array. As a result, after this process, the 0th position of the original array becomes the smallest value in the entire array.

  Since the array is initialized with negative infinity values at $i = 0$, the induction step holds, and the algorithm is proven to work for $K = k$ in all stages.

From the above proof, it is established that in any pipeline cycle, the 0th position of the array always holds the smallest value in the entire array. At the end of the last clock cycle, we add additional $K/2$ clock cycles to ensure that the array becomes fully ordered. This can be guaranteed because:

- At the end of the last clock cycle, the 0th and 1st positions of the array hold the smallest and second smallest values, respectively.
- Since the swap algorithm can be applied to arrays of any length, after the i-th additional clock cycle, the smallest

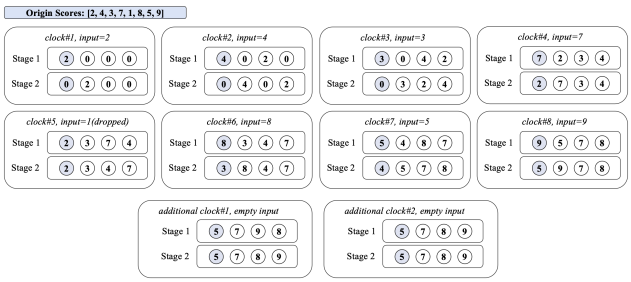


Figure 12: A simplified operational example of the Top-K module with M=8 and K=4.

and second smallest values from position $2i$ to $(K-1)$ are placed in their correct positions, while the first $2i$ elements are already the smallest $2i$ values in the array, and remain unchanged.

- By the end of the $K/2$-th additional clock cycle, the entire array is fully sorted.

We present a simplified operational example in Figure 12 with M=8 and K=4 (note that $M >> K$ in practice) to demonstrate the complete workflow of the Top-K module. It can be observed that the 0th array position consistently maintains the minimum value among all received inputs at every clock cycle termination. Furthermore, the array attains full ordering within at most $K/2$ additional clock cycles.

A.2 Analysis of the Filtering Efficiency

Assuming the total number of embeddings in the corpus is M, i.e., M scores are produced in the similarity calculation module, and the total number of scores that are dropped out by filter is M_f. In order to balance throughput, the total number of filtered scores must not be less than the throughput difference between the two modules, which is given by:

$$M_f \geq M - \frac{M}{N_E}. \tag{8}$$

Considering that the scores generated from similarity calculations are random, for the sake of simplification, let's assume that all scores follow a random uniform distribution within a certain interval. Taking the first input data to the filter as cycle 0, the first output data of the filter and the first input data into the Top-K selection module occur at cycle 1. According to the previous analysis, we know that the first valid smallest Top-K value is produced in cycle $(K+1)$, and it is fed back to the filter in the next cycle. This is because it takes K cycles to fully populate the Top-K array with valid scores. In the subsequent cycles, the current smallest Top-K value based on the valid scores will be replaced. Since the filter requires receiving the valid minimum Top-K value to function properly, it will begin to operate effectively at cycle $C_0 = K+2$ or later:

It is worth noting that the minimum Top-K value is updated every pipeline cycle. In other words, in cycle (C_0+i), the scores input to the filter are compared with the minimum value of the Top-K value of the first $(i+K)$ scores. Since the similarity scores are uniformly random, the probability that a score enters this sequence and is ranked beyond the K-th position after reordering is the probability of being filtered. Therefore, in cycle (C_0+i), the probability of the input score being filtered by the filter is denoted as $P(i)$:

$$P(i) = \frac{(i+K)+1-K}{(i+K)+1} = \frac{i+1}{i+K+1}. \tag{9}$$

In every pipeline cycle, the total number of scores input to the filter is N_E. Since the filter starts inputting scores at cycle 0, the last batch of inputs will arrive at cycle (M/N_E-1), which means:

$$i_{\max} = M/N_E - 1 - C_0. \tag{10}$$

According to Equation 9 and Equation 10, the total number of filtered scores can be expressed by the following equation:

$$\begin{aligned} M_f &= \sum_{i=0}^{i_{\max}} N_E \cdot P(i) \\ &= M - C_0 \cdot N_E - K \cdot N_E \cdot \sum_{i=0}^{i_{\max}} \frac{1}{i+K+1} \end{aligned} \tag{11}$$

Let $j = i+K+1$, we can then obtain:

$$\begin{aligned} \sum_{i=0}^{i_{\max}} \frac{1}{i+1+K} &= \sum_{j=K+1}^{i_{\max}+K+1} \frac{1}{j} \\ &\leq \int_{K}^{i_{\max}+K+1} \frac{1}{x}\,dx \\ &= \ln(\frac{M}{K \cdot N_E} - \frac{2}{K}). \end{aligned} \tag{12}$$

Since we have $M/N_E >> 2$, substituting Inequation 12 into Equation 10, we obtain:

$$M_f > M - C_0 \cdot N_E - K \cdot N_E \cdot \ln(\frac{M}{K \cdot N_E}). \tag{13}$$

Therefore, to ensure the validity of Inequation 8, it is only necessary for the following inequality to be satisfied:

$$M - K \cdot N_E - 2 \cdot N_E - K \cdot N_E \cdot \ln(\frac{M}{K \cdot N_E}) \geq M - \frac{M}{N_E}. \tag{14}$$

Let $\gamma = K/M$ represent the recall rate of the entire retrieval system. Substituting γ into Inequation 14, we obtain:

$$\frac{1}{N_E^2} \geq \gamma - \gamma\ln(\gamma) - \frac{2}{M} - \gamma\ln(N_E) \geq \gamma - \gamma\ln(\gamma), \tag{15}$$

which limits the maximum value of N_E, demonstrating the fact that it becomes increasingly challenging to balance throughput as N_E increases. It is worth noting that $\gamma < 1$, and the

derivative of the inequality's right-hand side $-\ln(\gamma)$ is greater than 0, indicating that the right-hand side of the inequality monotonically increases as γ increases. This suggests that smaller recall values can provide a larger range of possible values for N_E.

The filter can handle throughput imbalance within a certain range, however, additional cache space needs to be set up to avoid losing data that is expected to enter the Top-K selection module. Within one clock cycle, the filter receives N_E scores, and more than one score may be greater than the current minimum value in the Top-K sequence. Therefore, these scores need to be temporarily cached and passed to the Top-K selection module in the following clock cycles. Caching scores happens in two scenarios:

- Before clock C_0, when the current Top-K sequence has not produced the current minimum value yet, all scores need to be cached. In this case, the total number T of scores to be cached is given by:

$$T = C_0 \cdot (N_E - 1) = K \cdot N_E + 2 \cdot N_E - K - 2. \quad (16)$$

- After clock C_0, there may be occasional cases where more than one score is greater than the current minimum Top-K value. The specific design of the filter adopts the following strategies: (*i*) If the filter currently contains scores, it will prioritize outputting scores from the filter and cache any input scores that are greater than the current minimum Top-K value. (*ii*) If the filter is empty, it will output the first input score that is greater than the current Top-K minimum value, and cache any other eligible input scores. (*iii*) If the filter is empty and all input scores are smaller than the current minimum Top-K value, it can randomly output one score to balance throughput, which will not affect the output of the Top-K selection module.

Minos : A Lightweight and Dynamic Defense against Traffic Analysis in Programmable Data Planes

Zihao Wang[1,2] Qing Li[1] Guorui Xie[1] Dan Zhao[1] Kejun Li[1]
Zhuochen Fan[1] Lianbo Ma[3] Yong Jiang[1,2]
[1] *Pengcheng Laboratory* [2] *Tsinghua Shenzhen International Graduate School*
[3] *Northeastern University*

Abstract

Encrypted traffic analysis techniques extract valuable information from encrypted traffic and pose significant threats to user privacy. However, existing defense mechanisms against traffic analysis either incur significant bandwidth overhead and lack scalability, or fail to provide sufficient defense against evolving attacks. The emerging programmable switches provide data plane programmability with line rate packet processing to support advanced defense mechanisms. In this work, we present Minos, a lightweight and scalable programmable switch-based defense mechanism while providing both identity anonymity and traffic anonymity. Minos comprises three key modules: the Proxy Module, the Traffic Morphing Module, and the Schedule Module. In the Proxy Module, we design encryption round compression to take advantage of the match-action pipeline of programmable switches and realize line rate packet header encryption. The Schedule Module incorporates a lightweight dynamic flow scheduling method to interleave packets from different flows, so as to simulate dummy packets without causing bandwidth and delay overhead on the data plane. The Traffic Morphing Module further obfuscates the flows by dummy packet insertion and packet padding. Specifically, we devise a lightweight dummy packet scheduling method using priority dummy queues, minimizing bandwidth and delay overhead within the switch pipeline. We implement our defense on Tofino1 switches and adapt our method to defend Website Fingerprinting and IoT Fingerprinting. The results show that Minos can reduce the accuracy of previous attacks to less than 20% with only one-tenth of the overhead of existing defenses.

1 Introduction

Nowadays, user privacy has become a major concern [54–56], leading to the development of encryption protocols that secure packet payloads and shield user information. Nevertheless, even with encryption protocols, attackers can still compromise user privacy through encrypted traffic analysis techniques [26, 39, 48, 62]. These techniques assume a passive attacker who eavesdrops traffic from the victim by its 5-tuple (Source IP, Destination IP, Source port, Destination port, Protocol) to infer users' intentions–usually the resources they are requesting, such as websites or videos. Attackers can classify encrypted traffic by machine learning classifiers, without actually knowing packet contents, using features extracted from just packet sizes, packet directions, and some statistical features such as the total number of packets and the fraction of incoming or outgoing packets to the total number of packets. By leveraging the aforementioned 5-tuple, attackers can intercept the privacy of specific targets, deducing the content they access. Traffic analysis methods for different kinds of applications have been developed, including Website Fingerprinting (WF) [20, 42, 48] and IoT Fingerprinting [26, 46], posing serious threats to these applications.

The most popular defense methods against traffic analysis are Proxy-based defenses, which obfuscate the 5-tuple of user flows to hinder attackers from capturing pure traffic without background noise, thereby reducing the accuracy of classifiers. For example, IPsec gateways replace the source IP of each flow with its own public IP to create an end-to-end encrypted tunnel between two gateways. Multiple flows from various users are concurrently transmitted through this encrypted tunnel so that attackers can only capture a blend of noisy traffic, making categorizing encrypted traffic considerably more challenging [49]. However, they suffer from several critical weaknesses. First, most IPsec gateways have **limited throughput** [35] and cannot process today's high rate traffic, i.e., 100Gbps. Furthermore, security gateways, being specialized hardware with fixed capabilities, are not designed to incorporate advanced security features. This limitation not only renders them ineffective against new threats but also means they do not offer **traffic anonymity** to protect users' privacy. For example, the attack proposed in [14] applies packet labeling to each packet and successfully infers device types through noisy IoT traffic with per-packet features.

Corresponding Author: Yong Jiang (jiangy@sz.tsinghua.edu.cn), Qing Li (liq@pcl.ac.cn)

Table 1: Comparison of the existing solutions.

Scheme	Lightweight	Scalable	Anonymity	
			Identity Anonymity	Traffic Anonymity
Proxy-based Defenses	✓	✓	✓	×
Traffic morphing-based Defenses	×	×	×	✓
Ditto [28]	×	✓	×	✓
Minos	✓	✓	✓	✓

Traffic morphing-based defenses [9, 10, 52, 58] can effectively impede fingerprinting attacks by morphing characteristics of the origin traffic flows through padding, splitting, and inserting packets, reducing the accuracy of WF attacks to random guess [10]. These defense mechanisms obfuscate the metadata of user flows, providing traffic anonymity to defeat traffic analysis attacks. However, traffic morphing-based defenses share a common disadvantage: their practical deployment can be challenging due to high **bandwidth overhead**. For instance, in Tamaraw [10], the bandwidth overhead reaches 199%, resulting in a goodput of less than 40%. Besides, traffic morphing-based defenses, targeted at individual users, lack **scalability** to perform defenses for multiple users due to the escalated overhead. Finally, traffic morphing-based defenses do not offer **identity anonymity** and require a VPN to obfuscate the 5-tuple.

Lately, researchers have explored line rate traffic obfuscation on programmable switches [28] for traffic morphing-based defenses. Programmable switches, with their inherent flexibility, scalability, and customizability, offer line rate processing capability of up to 100Gbps and the ability to adapt to changing network demands. Ditto [28] establishes tunnels between programmable switches and sends packets with a static traffic pattern of fixed intervals and fixed size sequences. Although Ditto has achieved increased throughput compared to its predecessors, it still introduces significant **additional bandwidth overhead** by padding packets to specified sizes, as well as **increased latency** due to passing packets through the switch twice. Besides, Ditto requires the cooperation of IPsec gateways or MACsec gateways to hide user identity, thus fails to provide **identity anonymity**.

In this work, we present Minos, a lightweight and scalable programmable switches-based defense mechanism while providing both *identity anonymity* and *traffic anonymity* at 100Gbps line rate. In Minos, we design three modules as defense primitives in the switch hardware: the Proxy Module, the Schedule Module, and the Traffic Morphing Module. The Proxy Module performs line rate packet header encryption and acts as gateway switches of a trusted entity to realize an end-to-end encrypted tunnel to provide **identity anonymity**. The Traffic Morphing Module enhances the defense performance of Proxy-based defenses by performing lightweight dummy packet insertion and packet padding to provide **traffic anonymity**. In the Schedule Module, we dynamically schedule multiple flows to accommodate multiple end users.

The design and implementation of Minos on programmable switches face three challenges: 1) Ensuring line rate encryption with the Proxy Module is challenging since programmable switches have limited computational resources and cannot accommodate complex calculations or numerous encryption rounds; 2) Managing a multitude of flows from various users concurrently with the Schedule Module is challenging since we need to prevent packet disorder and minimize the overhead associated with traffic morphing. 3) Implementing the Traffic Morphing Module requires inserting dummy packets to obscure flows. However, since packets cannot be directly generated in the data plane and must be injected in advance by the control plane, inserting dummy packets into flows in real-time poses a significant challenge.

We address the above key algorithmic and system design challenges in Minos as follows. To achieve line rate encryption within constrained computations and resources, we devise an *Encryption Round Compression method* to implement the Proxy Module, which realizes implementation on hardware programmable switches equipped with match-action tables, eliminates the necessity for multiple pipeline passes, and ensures line rate encryption. To handle concurrent user traffic in a dynamic and scalable manner, we propose a *Dynamic Flow Scheduling Method* in the Schedule Module, which leverages packets from different flows to act as dummy packets to other flows. We formulate our flow scheduling algorithm to obfuscate flows dynamically according to multiple per-flow states and enable traffic morphing based on active flow numbers. With the scheduling algorithm, Minos is able to obfuscate flows with minimal dummy packets and packet padding. To realize real-time insertion of dummy packets, we design *Priority Queue based Dummy Packet Scheduling* to approximate real-time dummy packet insertions with little bandwidth overhead, thereby enabling the implementation of a lightweight Traffic Morphing Module.

We implement Minos on Tofino1 switches, and the evaluation shows that Minos has better throughput performance than previous defenses. The evaluation results also show that Minos can reduce the accuracy of Website Fingerprinting attacks to less than 20% with lower bandwidth and latency overhead than previous defenses.

The remainder of this paper is organized as follows. We first discuss the related work in Section 2, and then introduce our design goals and Minos system overview in Section 3. Then, we elaborate on the design of the Proxy Module, Schedule Module, Traffic Morphing Module in Section 4, 5 and 6. We evaluate the performance of Minos in Section 7, and adapt Minos to defend against Website Fingerprinting in Section 8. At last, we conclude our work in Section 9. We further expand our defense to defeat IoT Fingerprinting in Appendix A.

2 Background and Motivation

2.1 Encrypted Traffic Analysis

Encrypted traffic analysis is a passive attack in which attackers collect encrypted traffic flows by 5-tuple (Source IP, Destination IP, Source port, Destination port, Protocol) and analyze them through machine learning classifiers to infer whether victims are visiting some specific resources (e.g. websites and videos). Figure.1 shows an example of encrypted traffic analysis attacks, where users use different kinds of applications, e.g., websites, video streams, and IoT applications, inside a local network, and send encrypted traffic through a gateway. A local attacker may eavesdrop on the gateway, while a remote attacker eavesdrops on flows from the untrusted network by the 5-tuple of each flow, thereby locating the user and compromising their privacy. Considering the distinctive characteristics of different target applications, several encrypted traffic analysis attacks have been designed, e.g., Website Fingerprinting (WF) [20, 42, 48] and IoT Fingerprinting [26, 46].

Website Fingerprinting. Website Fingerprinting attacks target victims who use proxies and encrypted protocols to hide their identities and packet contents when browsing. Early works like [16, 48] focus on HTTPS traffic, where attackers are able to utilize the sizes, directions, and intervals of packets. Wang et al. [48] design an attack that utilizes the k-Nearest Neighbour classifier on a feature set including unique packet lengths, bursts, etc., and achieve an accuracy of 0.85 on early defenses like [25, 52]. Recent WF attacks aim at Tor [13], one of the most frequently used anonymous communications tools. Tor conceals clients' packet size by sending packets in fixed size (512-byte) Tor cells, rendering attacks more challenging because Tor traffic only discloses packet directions. Hayes et al. [20] develop a k-fingerprints attack with random forests to extract and represent fingerprints of different traffic traces and achieve 88% TPR in an open-world setting. Recent attacks [6, 37, 42] exploit deep learning against WF defenses, achieving accuracy over 95%.

IoT Fingerprinting. IoT Fingerprinting poses a significant threat to the rapidly growing IoT user base. IoT Fingerprinting attacks have seen success in decrypting IoT device types [30, 43, 44]. For example, [30] identifies device types with over 95% accuracy by passively observing network traffic using a two-fold classification system. Beyond device fingerprinting, [26, 45, 46] even manage to infer user activities from IoT traffic, further exacerbating user privacy concerns. [26] generates sequence profiles from packet sequences and matches them to collected traffic traces to estimate device types and user interactions. [45] develops two polynomial time algorithms to capture IoT devices' activity signatures and full activity sequences.

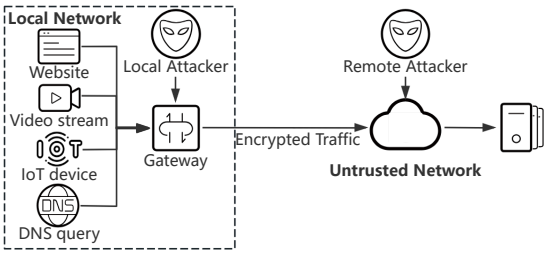


Figure 1: An example of encrypted traffic analysis.

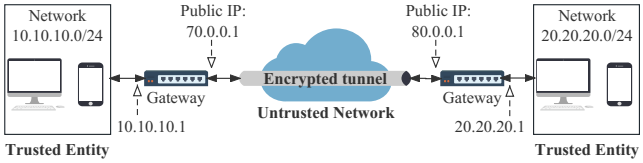


Figure 2: An example of IPsec gateway.

2.2 Defenses against Encrypted Traffic Analysis

Proxy-based defenses. Proxy-based defenses obfuscate the 5-tuple of user flows, preventing passive observers from obtaining clear traffic traces without noise. Popular Socks-based proxies like Shadowsocks [3] and V2Ray [2] break one access to a website into several flows by randomly generating new ports to request different resources in a website. Hence, the attackers can only capture partial access to the website through various 5-tuples. IPsec gateways, commonly used as LAN gateways, establish a secure and encrypted tunnel with aggregated flows by replacing the original IP headers with new IP headers with the same public IP, as illustrated in Figure. 2. In this way, attackers can only capture a noisy flow where multiple flows are actually transmitting, rather than pinpointing any individual user because they cannot capture the user's original IP address.

Proxies are widely deployed because they are lightweight, bringing little bandwidth or latency overhead, and can handle multiple end users. However, these proxies fail to defend against new encrypted traffic analysis. [27] succeeds in recognizing websites through per-flow traffic analysis and breaches socks-based proxies. [14] identifies device types by a LSTM model with per-packet labels, threatening IoT users even with the protection of IPsec gateways. Although proxies provide identity anonymity, emerging studies indicate that traditional Proxy-based defenses are no longer safe, thus traffic morphing-based defenses are necessary.

Traffic morphing-based defenses. Traffic morphing-based defenses obfuscate traffic traces to resist traffic analysis attacks by inserting, delaying, combining, or splitting packets. BuFLO [16] removes all side-channel information by sending fixed-length packets at a fixed interval for at least a fixed amount of time. [21] proposes WTF-PAD, an adaptive

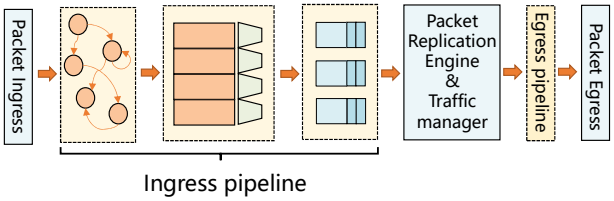


Figure 3: Standard Tofino pipeline.

padding method on top of Tor, which utilizes a finite state machine along with histograms. WTF-PAD is effective against conventional attacks, with half the bandwidth overhead of BuFLO, but fails to resist the more advanced deep learning-based attacks [42]. To defeat deep learning classifiers, researchers have applied adversarial perturbations [32] and patches [23]. [18] designs a lightweight defense that dedicates most of its bandwidth budget to obfuscate the trace front. In IoT Fingerprinting, Alshehri et al. [5] attack tunneled IoT traffic based on packet-size sequences and propose an algorithm to pad packets with uniform random noise to defend against it. According to [5] and [44], it is necessary to hide packet lengths to defend against IoT Fingerprinting.

In summary, traffic morphing-based defenses, despite their effectiveness, face three main challenges. First, their high bandwidth and latency overhead hinder real-world deployment, especially for handling multiple users within a trusted entity. Second, deploying multiple targeted defenses simultaneously is infeasible and laborious due to excessive bandwidth demands. Third, traffic morphing-based defenses can not hide user identity and require the cooperation of proxies. Although recent studies [17] have increased the packet processing speed of end hosts, it is impractical to equip every user with a dedicated network interface card. Moreover, configuring defenses for each user would bring escalated overhead and place a greater burden on the network.

2.3 Traffic Analysis Defense on Programmable Switches

Traditional switches have fixed default functions. Thus network operators have to apply specific middleboxes or upload packets to servers [29] to support customized functions, which bring extra costs with limited throughput [57]. The emerging programmable switches provide data plane programmability with line rate packet processing capability to support customized functions [53, 59, 63]. This advancement presents a valuable opportunity to enhance the throughput of traffic morphing-based defense mechanisms by integrating traffic morphing directly into the programmable pipeline [28]. Our work focuses on PISA (Protocol Independent Switch Architecture) processors, which allow packet processing at line rate (100 Gbps) with flexible modification of packet headers by P4 [8]. A typical Tofino architecture is shown in Figure. 3.

Packets sent to the data plane pass through two programmable pipelines and a Traffic Manager (TM). In the ingress pipeline, packets are first parsed by the ingress parser and mapped into the Packet Header Vector (PHV). After leaving the parser, PHVs are sent into the match-action pipeline where network admins can design their own packet process logic by configuring match-action tables. Packets are then assembled by the deparser and passed to the TM, where packets are replicated, recirculated, or buffered in the Traffic Manager's round-robin queues and then passed forward to the egress pipeline which is identical to the ingress pipeline.

With the help of the programmable pipeline and multi-functional Traffic Manager, programmable switches can potentially deploy proxy-based defense and traffic morphing-based defense at the same time, providing both identity anonymity and traffic anonymity. Several studies have attempted implementing encryption algorithms on programmable switches. [11] and [60] implement standard encryption algorithms, e.g., AES, on programmable switches. However, they must send packets through the pipeline multiple times and thus cannot reach line rate. [12, 19, 33] either lack hardware implementation or require extensive communication with the control plane. [47] implements the two-round Even-Mansour scheme on hardware programmable switches, at the cost of excessive computing resource consumption.

A recent work, Ditto [28], establishes tunnels between programmable switches and obfuscates traffic with packet padding and dummy packets. However, Ditto exhibits a few limitations. First, Ditto relies on IPsec and MACsec gateways for encryption, which will bring additional deployment difficulties and costs. Besides, IPsec gateways have limited throughput compared to programmable switches and may become the bottleneck of the whole system. In Minos, we provide traffic obfuscating primitives along with encryption primitives and no extra devices are required. Second, Ditto realizes complete traffic anonymity by sending packets through a static traffic pattern with fixed intervals and fixed size sequences. Consequently, each packet must be padded to a predetermined size and sent through the pipeline twice for packet scheduling, resulting in significant latency and bandwidth overhead. Third, Ditto cannot adapt to dynamic network environments due to its reliance on a static traffic pattern.

3 Minos Overview

3.1 Design Goals

In this work, we propose Minos, a line rate traffic analysis defense scheme based on programmable switches. Based on the shortcomings of proxy-based and traffic morphing-based defenses, we propose the following design goals.

Lightweight. Minos is lightweight and easy to deploy in the real world. We design each Minos module with minimal

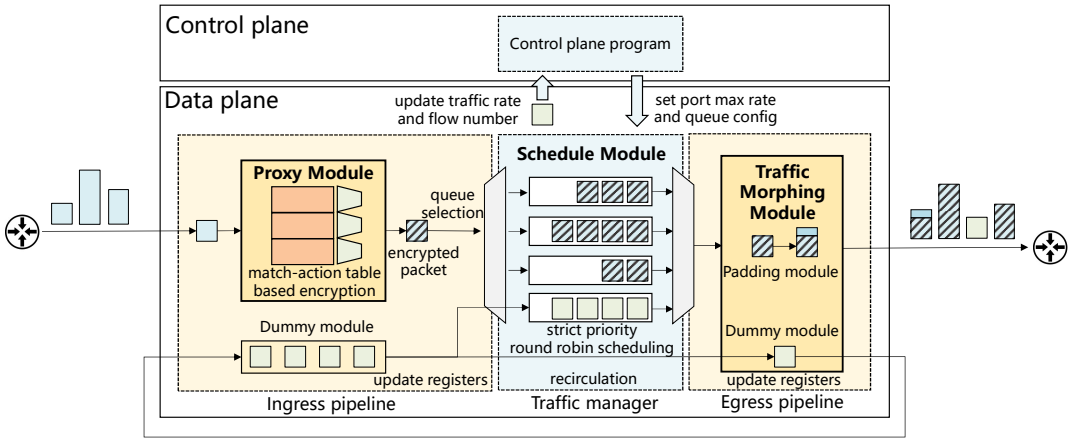


Figure 4: System overview of Minos.

bandwidth overhead and hardware resources.

Scalable. Minos is able to dynamically adjust its defense strategy to accommodate different circumstances and handle concurrent flows from multiple users.

Anonymity. Minos provides both identity anonymity and traffic anonymity. Minos not only hides user identities through encryption but also performs traffic obfuscation, concealing metadata within the traffic.

Like IPsec, Minos establishes an end-to-end tunnel between an obfuscating gateway (e.g., an organization's gateway) and a corresponding de-obfuscating gateway (e.g., ISPs that offer enhanced privacy or IoT operators that secure user data) for traffic obfuscation. We assume a passive attacker [20,48] who collects traffic from any point between Minos gateways and classifies them from their 5-tuples but cannot modify or insert packets.

Unlike Tor, the goal of Minos is not to hide the identities of the endpoints involved, but to establish an encrypted channel between the sender and the receiver and to conceal the metadata and the user identities. This objective is similar to the setup in Ditto. In real-world deployment, for instance, when deploying Minos to defend against IoT Fingerprinting attacks, it is necessary for IoT devices to transmit essential information to the IoT service provider. The objective of Minos is not to conceal the identity of the IoT service provider, but rather to ensure that attackers are unable to infer users' IoT activity from the communication between the IoT device and the service provider. This principle remains consistent when deploying Minos to mitigate other types of attacks.

3.2 System Overview

The main process of Minos runs on the programmable data plane and communicates with the control plane to configure Traffic Manager settings. Packets entering the Minos switch will pass three modules before exit: the Proxy Module, the Schedule Module, and the Traffic Morphing Module.

The **Proxy Module** acts as a gateway switch that replaces the source IP of each packet with the gateway IP and sends packets at line rate. The original source IP and other fields will be encrypted and placed in a new header. Particularly, we design Encryption Round Compression exploiting match-action tables to achieve line rate implementation of packet header encryption on programmable switches.

The **Schedule Module** aggregates different flows to the same destination to obfuscate the original traffic with minimal dummy packets. Packets entering Minos switches will be inserted into specific queues and interleaved by the switch's Traffic Manager. Specifically, we propose a dynamic schedule algorithm to perform flow interleaving and decide whether to enable the Traffic Morphing Module. With the co-action of the Proxy Module and Schedule Module, Minos aggregates each incoming flow with limited overhead.

The **Traffic Morphing Module** provides primitives to implement traffic morphing-based defenses, consisting of the *Dummy Module* and the *Padding Module*. The former module manages dummy packets and the latter module pads and assembles packet headers. We design priority queue-based dummy packet scheduling to realize real-time dummy packet insertion.

Figure. 4 shows the workflow of Minos. When a flow enters a Minos switch, its source IP will be replaced and encrypted by the **Proxy Module** in the ingress pipeline, and scheduled into the Traffic Manager by the **Schedule Module**. Meanwhile, the **Schedule Module** examines the flow state and decides whether the **Traffic Morphing Module** should be applied. When few flows are active, simply interleaving flows is not sufficient to defeat advanced attacks. Thus, the **Traffic Morphing Module** will be activated, and its Dummy module and Padding module will perform dummy packet insertion

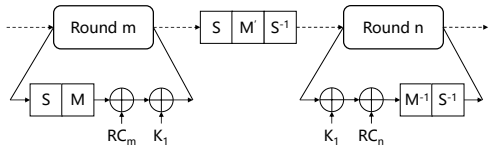


Figure 5: Simplified encryption process of $PRINCE_{core}$.

and packet padding, respectively. When there is a substantial number of active flows, the Traffic Morphing Module is not needed. The control plane periodically adjusts queue configurations and changes the port max rate limitation to perform strict priority round-robin scheduling in the Schedule Module.

4 Proxy Module

The Proxy Module acts as a secure gateway switch and performs line rate packet header encryption. In this section, we introduce how we design and implement the Proxy Module to reach line rate.

4.1 Choose of cipher

Similar to IPsec gateways, Minos Proxy Module replaces the original source IP of each flow with the public IP of the gateway. Related information (e.g., the original IP header) is encrypted and placed in a new header. Minos is typically used for Website Fingerprinting, IoT Fingerprinting, etc., and the receiver end is often a large service provider. Since these providers commonly deploy technologies such as anycast and CDN with publicly accessible IP addresses, encrypting the destination IP is not necessary. Therefore, Minos encrypts the source IP address and source port to provide identity anonymity.

However, achieving line rate implementation of an encryption algorithm in programmable switches is a challenging task, due to the limited resources and computations of the programmable switches. The selection of cipher should take into account the following aspects. First, the encryption algorithm should exclusively employ computational operations compatible with P4 switches. Asymmetric encryption algorithms that require prime-based computations [22, 38] and resource-intensive symmetric encryption algorithms like IDEA [1] are impractical choices. Second, as Tofino1 programmable switches consist of 12 stages per pipeline and 24 stages in total, the chosen encryption algorithm should be completed within 24 rounds. For instance, implementing a 64-round lightweight cipher TEA [51] on a programmable switch would necessitate multiple pipeline passes, resulting in increased latency overhead. Third, the cipher should utilize as few match action tables as possible to avoid resource contention with other programs.

In Minos, we choose PRINCE [7], a low-latency, lightweight SPN-based block cipher as our encryption algorithm on the programmable switch. Figure. 5 shows the basic core encryption process of PRINCE, with one encryption round on each side. PRINCE takes a 64-bit string as input and cuts it into sixteen 4-bit strings. In each encryption round, a 4-bit Sbox maps a 4-bit input to a 4-bit output, which is multiplied with a matrix M and then $XORed$ with a 64-bit RC as well as the key k_1. String-to-string mapping can be easily realized by match-action tables, making Sbox-and-permutation-based ciphers well-suited for programmable switches. Also, PRINCE, with its maximum of 10 encryption rounds (6 or 8 rounds also viable), aligns with typical 12-stage pipelines. Additionally, PRINCE features a α-reflection property, enabling encryption and decryption with the same set of match-action tables.

4.2 Encryption round compression

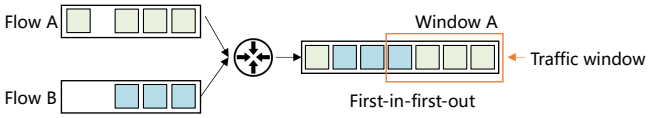
Although the PRINCE algorithm is well-suited for programmable switches, it cannot be directly implemented on programmable switches due to the limitation of stages. Each encryption round involves a sequence of non-parallelizable operations. Allocating each step of an encryption round to a separate stage would necessitate four stages per round and a total of at least forty stages, exceeding the capacity of a single 12-stage pipeline. To address this limitation, we add k_1 and RC_i together and pre-add the k_1-RC_i to each Sbox in each round to combine the Sbox together with XOR. It's hard to implement the matrix multiplication in the data plane because this computation is not supported in programmable switches. However, we observe that the actual input and output of each 4*4 matrix are four 4-bit strings thus we can combine the matrix multiplication step with the Sbox-xor table by implementing a 4-bit to 4-bit mapping table for each of the 16 4-bit strings.

4.3 Memory Consumption Reduction

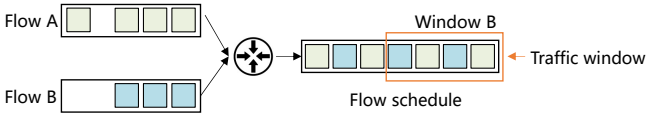
The α-reflection feature of PRINCE cipher allows easier implementation on programmable switches. With a 128-bit key k, PRINCE divides it into two 64-bit halves k_0 and k_1, and employs these halves to generate a third 64-bit chunk $k'_0 = (k_0 \ggg 1) \oplus (k_0 \gg 63)$. PRINCE encrypts a packet on the sender switch with k_0, k'_0, k_1 the receiver switch decrypts the packet by encrypting the packet again with $k'_0, k_0, k_1 \oplus \alpha$:

$$D(k_0||k'_0||k_1)(message) = E(k'_0||k_0||k_1 \oplus \alpha)(message), \quad (1)$$

where α is a constant derived from the fraction part of π. Taking advantage of this feature, when implementing Minos on a P4 switch, we employ a single set of keys and match-action tables for both encryption and decryption, effectively reducing the SRAM consumption by half. The match-action tables are determined by key k_1, and the Minos sender switch



(a) An example of IPsec gateways.



(b) An example of Minos flow schedule.

Figure 6: A comparison of IPsec gateway and Minos.

can hold multiple sets of keys to establish encrypted tunnels with different receiver switches by altering k_0, without the need to modify match-action tables. As both ends of the tunnel are trusted, keys can be exchanged and updated by operators through the control plane.

5 Schedule Module

In Minos, the Schedule Module mixes multiple flows by performing dynamic flow scheduling. Figure. 6a illustrates how an IPsec gateway mixes flows. An IPsec gateway mixes packets from multiple flows and sends them out in a first-in-first-out manner, which unavoidably includes consecutive packets from the same flows. Traffic analysis attackers can potentially exploit traffic windows comprising consecutive packets to deduce privacy-related information, such as device types [14,41]. Prior traffic morphing-based defense schemes [9, 10] insert dummy packets between original packets to address this flaw, but at the expense of increased bandwidth and latency overhead.

In Minos, the Schedule Module addresses this with greater efficiency by cleverly exploiting multiple flows, using packets from different flows as dummy packets for one another. As illustrated in Figure 6b, the Schedule Module interleaves packets from flow A and flow B, enabling packets from one flow to act as dummy packets for the other. This innovative scheduling strategy disrupts the attacker's model, significantly reducing its accuracy while providing an effective and lightweight defense mechanism.

The detailed scheduling process of Minos on the data plane is illustrated in Figure. 7. In Minos, our aim is to achieve per-flow scheduling, but the match-action pipeline in a programmable switch typically handles flows on a per-packet basis, lacking consideration for flow states. To enable flow interleaving, maintaining per-flow states becomes essential. In Minos, we address this challenge by utilizing registers within the programmable data plane to store the following flow states:

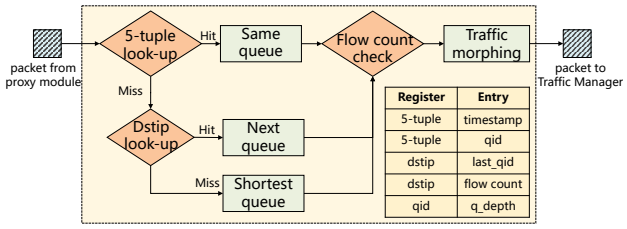


Figure 7: Schedule algorithm of Minos.

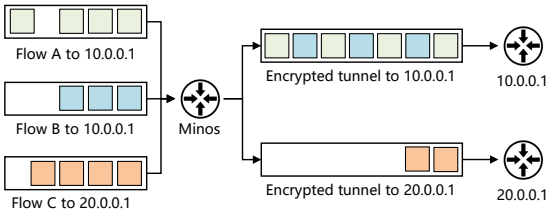


Figure 8: Example of multiple encrypted tunnels.

- **Timestamp**. The timestamp for a flow identified by the 5-tuple is stored in the corresponding entry. Upon packet ingress, Minos records the entry timestamp, signifying when the last packet from this flow entered the pipeline. Inactive flow entries are subject to expiration and removal from the register through the recirculation of dummy packets.
- **Queue ID**. The queue ID to which the flow is sent.
- **Flow count**. A Minos sender switch maintains multiple encrypted tunnels with multiple destination IPs, each representing a Minos receiver switch that performs decryption and de-obfuscation. We monitor the number of flows currently transmitting to each destination IP. Figure. 8 shows the scheduling of three flows. Flow A and B are sending to 10.0.0.1, so they will be mixed into an encrypted tunnel to 10.0.0.1. Flow C is sending to 20.0.0.1, so it will be sent into another encrypted tunnel. We assume that passive attackers will eavesdrop on each encrypted tunnel based on sender and receiver Minos switches' IP addresses. Recording flow numbers in each encrypted tunnel helps Minos decide whether traffic morphing is needed.
- The last assignment of the queue ID of each destination IP.

For each packet entering the pipeline, Minos first maps its 5-tuple into the timestamp register and performs a lookup in the queue ID register. If an entry exists, it means that previous packets of this flow have been sent into a specific queue so that the following packets should be sent through the same queue to avoid packet reordering. If no entry is found, it means

the packet is initializing a new flow. Then, the 5-tuple will be matched in the destination IP register. If an entry is found in the destination IP register, the flow will be assigned to the next queue based on the recorded queue ID. This new flow's queue ID will be the origin queue ID plus one. In this way, flows in the same encrypted tunnel will be evenly spread into different queues and scheduled in a round-robin manner, just like Flow A and B in Figure. 8. If the destination IP is not found, it means that we are building a new encrypted tunnel and no restrictions are applied to this flow. In this case, the flow will be sent into the shortest queue to balance queue length.

The flow count table keeps track of the number of flows sent to each destination to decide whether the Traffic Morphing Module should be activated. When the number of flows in an encrypted tunnel is very small, a mixture of these flows is insufficient to conceal their information effectively. Moreover, when only a few flows are active, the Schedule Module may encounter the problem of asymmetric flow interleaving, where one flow sends significantly more packets than others, resulting in consecutive packets from the same flow. This could potentially expose features from the aggregated flow to traffic analysis attackers. In such cases, the Traffic Morphing Module will perform dummy packet insertion to supplement packets for flow mixing. Besides, it will also conduct packet padding to further obfuscate flows. The flow count table is regularly checked to decide whether to invoke the Traffic Morphing Module to make sure sufficient protection is always in place. In the Schedule Module, we only track the target flows that require defenses, rather than recording every incoming flow. For detailed implementation, please refer to Section 7.1.

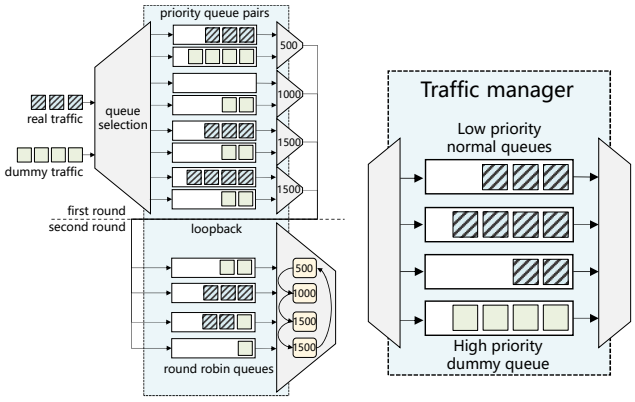
6 Traffic Morphing Module

The Traffic Morphing Module performs dummy packet insertion and packet padding to implement traffic morphing. In this section, we introduce how we implement the Dummy Module and the Padding Module in a lightweight manner.

6.1 Dummy Module

Although programmable switches cannot generate packets directly from the data plane, programmers can incorporate template packets into the pipeline. A typical workflow of this process is as follows. First, packet templates are sent by the switch CPU or through specific ports, marking every subsequent packet from that port as a dummy packet. Second, dummy packets are passed through the pipeline and essential functions are executed, e.g. updating registers. Third, after going through the ingress pipeline, dummy packets can either be sent into the egress pipeline or inserted into the recirculation port for another pass through the pipeline.

However, sending dummy packets in a pipeline is challenging, as they must be generated in advance and can not be sent



(a) Ditto Traffic Manager (b) Minos Traffic Manager

Figure 9: Different dummy packet scheduling methods.

on demand. In Minos, for instance, obfuscating a single flow within a tunnel requires the use of dummy packets. However, the inability to generate these packets instantaneously leads to unpredictable delays before a dummy packet can enter the pipeline. This limitation complicates the timely and seamless integration of dummy packets for effective obfuscation. Prior works [24, 28] have attempted to buffer dummy packets, but they cause significant bandwidth overhead.

In IMap [24], control plane programs generate dummy packets on the switch CPU, and recirculate them through the pipeline to accumulate speed. This approach handles dummy packet buffering by filling the pipeline entirely with dummy packets, enabling the generation of probe packets as needed. However, this approach introduces excessive dummy packets that would reduce overall throughput, contradicting our design goal of achieving a lightweight, line-rate defense model.

Another previous work, Ditto [28], conducts traffic morphing on the programmable data plane at the cost of high overhead. Ditto morphs traffic so that the sizes of packets sent out strictly adhere to a predefined pattern, e.g., [500,1000,1500,1500]. To achieve this, Ditto pads packets when their sizes are smaller than the intended size. Conversely, if the original packets are larger than the target size, Ditto uses dummy packets to simulate smaller ones. Consequently, a constant supply of dummy packets is required. As shown in Figure. 9a, Ditto uses hierarchical queues to buffer dummy packets, causing each packet to pass through the pipeline twice. In the first round, queues are organized by packet size (e.g., 500B). Real packets go to higher priority queues, while dummy packets fill lower priority ones. In the second round, packets are sorted by size and the traffic manager schedules them out in a round-robin manner to match the fixed pattern.

Previous studies fail to design a lightweight method to schedule dummy packets. In Minos, we design a priority-based dummy packet scheduling structure which utilizes priority queues in the Traffic Manager to approximate instant

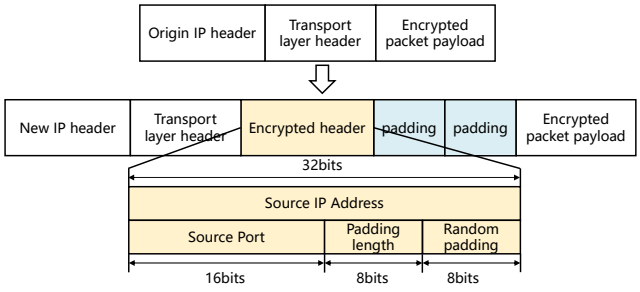


Figure 10: Minos packet format.

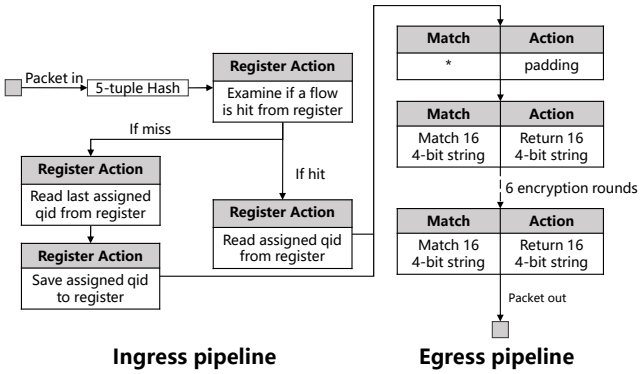


Figure 11: Minos flow table.

dummy packet insertion with strict priority queues. As Figure. 9b shows, in the Minos Traffic Manager, a set of queues for normal packets shares the same low priority, and a special dummy queue owns the highest priority. Normal packets are scheduled as Figure. 7 illustrates, and are sent in a round-robin manner. The dummy queue, while owning the highest priority, will be paused when it is not needed. Unfortunately, directly pausing and resuming queues is not a supported feature in P4 switches [40]. In Minos, we modify the queue state with the assistance of the control plane. Specifically, we send marker packets to the control plane at regular intervals to report the number of active flows. Then, according to the active flow number, the control plane decides the proper cycle lengths for periodically pausing and resuming queues. The control plane program adjusts the dummy packet rate as follows:

$$Dummy_rate = \left(\frac{1}{n} - d - i\right) * \frac{r}{R}, \tag{2}$$

where n is the number of rounds in the resume period, i is the interval between resume and pause periods, r is the output rate of the dummy queue, R is the current output rate of target flow, and d is a constant representing the delay of instructions between the control plane and the data plane. This approach allows us to efficiently blend normal packets with dummy packets at minimal cost, requiring only a single buffer queue in the Traffic Manager.

6.2 Padding Module

Besides dummy packets, packet padding is also necessary to hide packet length to defend against packet length attacks, such as PINGPONG [44]. In Minos, we apply random padding based on the average size of the original traffic. We can set different padding sizes according to different defensive needs, as discussed in Appendix A. The programmable switch pads packets by adding packet headers after regular packet headers. From the attackers' perspective, these headers are disguised as packet payload. As Figure. 10 shows, Minos adds multiple headers of different sizes after the transport layer header to realize dynamic padding. To de-obfuscate and remove these headers, the receiver Minos switch must be informed of the exact padding length.

Therefore, the padding length is also encrypted with the Proxy Module. A total of 64 bits will be encrypted and inserted after the transport layer header, including 32-bit source IP, 8-bit padding length, 16-bit source port, and 8-bit random padding. The padding size is calculated in the ingress pipeline and encrypted along with the source IP while padding headers are added in the egress pipeline. On the receiver switch, padding size and source IP will be decrypted in the ingress pipeline, so that extra headers can be stripped in the egress pipeline. Overall, the Proxy Module will bring a static 8 Bytes to each packet and the bandwidth overhead of the Padding Module depends on defense settings.

7 Hardware Prototype Evaluation

We implement a hardware prototype of Minos on two Barefoot Tofino1 switches with 32 × 100 Gbps ports. We use a traffic generator [4] to inject packets and record output traffic.

7.1 Implementation

The switch pipeline and flow tables are depicted in Figure. 11. Specifically, packets entering the ingress pipeline will first be hashed. Utilizing the inherent CRC32 hash function of the Tofino ASIC, the five-tuple of each flow is hashed into a unique flow identifier (ID). This flow ID subsequently serves as the key for storing flow-level information within the register. A Tofino1 switch offers 80 pages, each containing 1000 entries of 128b RAMs, which can be allocated for storing the flow-level information. These hardware resources enable Minos to manage over 10,000 concurrent flows. Nevertheless, as a gateway switch, the number of flows handled by Minos may exceed this capacity. It is important to note that Minos is designed to monitor only those flows that require defense, rather than monitoring the information of every single flow. Therefore, the register of Tofino does not affect the

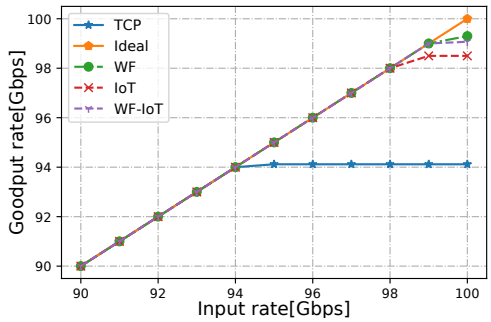


Figure 12: Throughput of Minos proxy module.

scalability of Minos. The dummy packets are injected from a specific port and assigned to the high priority queue. We use bfrt python scripts to pause and resume the dummy queue. In the egress pipeline, Minos first performs packet padding by adding packet headers. Then, in each encryption round, the original 64-bit string is segmented into 16 4-bit strings, and the match action table executes string-to-string mapping. Owing to the limitation of the number of stages, the maximum number of encryption rounds that can be performed is six.

Table 2 shows the utilization of each hardware resource. The Minos Proxy Module consumes only 6.15% of VLIWs and does not use any ALUs or TCAMs. Instead, it relies on 55.96% of SRAM to support match-action table-based encryption. Notably, many functions commonly deployed on P4 switches, such as Sketch [61] and Network Attack Detection [31], demand substantial ALU (>20%) or VLIW ($\approx$ 10%) resources but have low SRAM requirements. Therefore, Minos can coexist efficiently with these P4 programs. The receiving switch and the sending switch have the same hardware resource consumption because the same P4 program and the corresponding decryption flow tables need to be deployed.

7.2 Evaluation of the Proxy Module.

First, we evaluate the throughput of the Proxy Module. We send TCP packets of 128B at different rates to examine if Minos can perform line rate encryption and also evaluate the performance of the Proxy Module using realistic flows. The detailed construction of each dataset will be illustrated in section 7.6. In Figure. 12, the X-axis reflects the input rate of packets, and the Y-axis represents the actual output rate of real traffic without extra bytes brought by the Minos Proxy Module. The result proves that Minos is able to perform line rate source IP encryption in the case of realistic datasets. Minos reached an upper limit of 94 Gbps in the case of TCP streams because the packets in TCP streams are relatively small, which means that adding an 8-byte packet header has a greater impact and thus reduces the throughput. As Minos has a static 8-byte overhead per packet, the output ratio of real traffic will be higher if larger packets are sent.

Table 2: Hardware Resource utilization.

Module name	Stages	VLIWs	ALUs	TCAM	SRAM
Proxy module	10	6.15%	0%	0%	55.96%
Minos	12	7.81%	0%	0%	59.06%

Table 3: Arrival time of 1000th packet(s).

Origin	Limited output rate	Two flows	Three flows	Four flows
1	1.095861	1.102307	1.106605	1.108754

7.3 Evaluation of the Schedule Module.

We implement a simple scenario to evaluate the latency brought by the Schedule Module. In each flow, we send 1000 packets per second and record the arrival time of the 1000th packet. Then, we mix different numbers of flows to see if the flow mixture will bring extra latency overhead. In Minos, we record the current traffic rate and report to the control plane periodically to update port speed limitation, usually 99% of the current traffic rate. Otherwise, packets will not be cached in queues and will be sent out directly in the First-in-First-out manner. In this experiment, we set the output rate of the tested port at 90% of the current traffic rate to magnify the delay effect and evaluate the arrival time of 1000th packets of each flow.

As Table 3 shows, adding flow numbers only incurs less than 1% latency, which is insignificant compared to output rate limiting. The results show that Minos can handle multiple flows without incurring much delay. Therefore, we should keep track of the current traffic rate and synchronize it with the control plane frequently to reduce latency overhead.

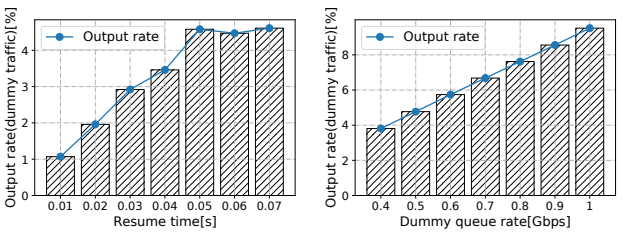
7.4 Evaluation of the Dummy Module

Minos enables real time dummy packet insertion by pausing and resuming priority queues. The control plane periodically pauses and resumes queues by altering queue status according to equation 2. We set the number of sending rounds n to 10 rounds per second and the output rate of normal queues to 1Gbps, and evaluate the impacts of resume interval i (X-Axis in 13a) and dummy queue output rate r (X-Axis in 13b) on dummy traffic rate.

When evaluating the impact of the interval between resume and pause dummy queue on output rate, we set the default output rate of the dummy queue to 0.1Gbps and alter interval lengths. As shown in Figure. 13a, the output rate of dummy packets increases as the interval increases and reaches its limit of about 4.5%. This is because we repeat each sending round 10 times per second, and each round lasts for 0.1 second. In this way, if we resume a queue for too long, the next resume instruction will come as soon as the previous pause instruction finishes, which means the queue will always be open.

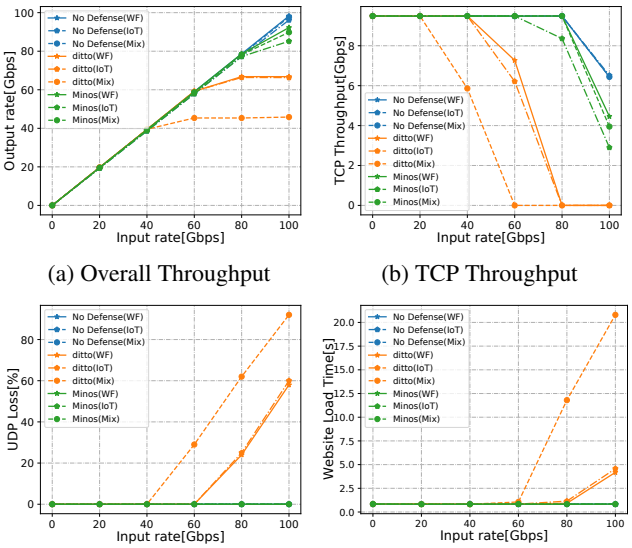
Table 4: Overhead of Minos

Flow type	arrival time of 500,000th packet(s)	Goodput rate(%)
Base	5	100%
One flow	5.02	95.3%
Multiple flows	5.12	99.2%



(a) Different intervals (b) Different output rates

Figure 13: Evaluation of the dummy module



(a) Overall Throughput (b) TCP Throughput

(c) UDP Packet Loss (d) Website Load Time

Figure 14: Evaluation results in comparison with Ditto.

In the evaluation of the impact of different dummy queue rates on output rate, we set the interval between resume and pause to 0.01 seconds. As shown in Figure. 13b, the output rate of dummy packets and the queue output rate exhibit an almost linear relationship, which means we can precisely control the output rate of dummy packets through the dummy queue output rate.

7.5 Overall Evaluation

At last, we evaluate Minos's overall latency overhead and bandwidth overhead of all three modules. We send packets of 1024B with 10,000 packets per second and evaluate Minos in both one-flow scenario and multiple-flow scenario. In the former case, we add 64B padding to each packet and insert dummy packets with 0.01s interval and 0.5Gbps output rate. When multiple flows are transmitting, we do not perform traffic morphing because flow interleaving is sufficient to defend against traffic analysis attacks.

Table 4 shows that Minos can perform with limited latency and bandwidth overhead. In the one-flow scenario, Minos reaches 95% goodput with little latency overhead. In the multiple-flow scenario, the goodput rate is close to 100% because we only add 8B of encryption header to each packet. The arrival time of the 500,000th packet experiences a slight delay of just around 2.4%. This delay is caused by the fact that the output rate of the tested port is limited to enable round-robin queues, as discussed in the evaluation of the Schedule Module.

7.6 Comparison with SOTA work

We compare Minos with the state-of-the-art traffic morphing-based defense on programmable switches, as presented in Ditto [28]. Ditto is an end-to-end system that incorporates two Tofino switches, responsible for obfuscation and de-obfuscation respectively. To assess the performance of Minos and Ditto across various network conditions, we choose three datasets for our evaluation: Website Fingerprinting dataset (WF) [37], IoT Fingerprinting dataset (IoT) [36], and a mixture of both (Mix). We send background traffic ranging from 0 to 100Gbps with these datasets and conduct four experiments. Besides, we configure Minos to perform a 64B padding to each packet. Note that the Dummy Module is disabled because the evaluation involves transmitting multiple flows, and dummy packet insertion are unnecessary.

First, we assess the overall throughput of Minos and Ditto. Figure. 14a illustrates that Minos is capable of adapting to diverse scenarios, whereas Ditto's throughput is capped at 45Gbps when dealing with packets of more uniform sizes in the Mixed dataset. In the TCP throughput experiments, we send background traffic ranging from 0Gbps to 100Gbps along with a 10Gbps TCP flow. Figure. 14b reveals that the throughput of Ditto drops to near 0 when sending 80Gbps background traffic, whereas Minos maintains more than 8Gbps throughput at 80Gbps and approximately 4Gbps at 100Gbps. Additionally, we examine the performance of Minos and Ditto in terms of the UDP packet loss rate and the loading time of Google. The results show that Minos's performance is almost the same as the ideal case, with no significant impact on applications. However, Ditto can cause packet loss and website loading delays under high throughput conditions.

8 Use Case: Website Fingerprinting

In this section, we implement Minos to defend against Website Fingerprinting. We introduce our dataset, baseline attacks, and defenses in section 8.1 and introduce how we adapt Minos to defend WF attacks in section 8.2. At last, we evaluate Minos against baseline attacks and defenses.

8.1 Evaluation Setup

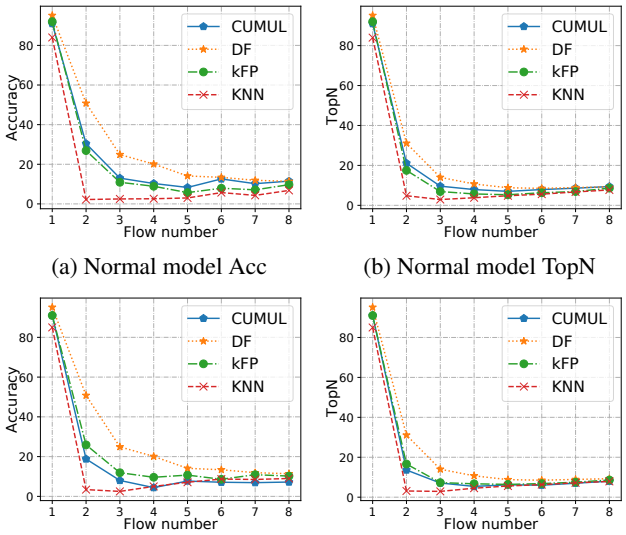
We request the parsed dataset with all meta-data including timestamps from [37]. The closed world dataset contains 900 websites, and 5,000 visits each. In our experiments, we select 300 websites and 100 traces each, 30,000 traces in total from the closed world dataset. Each trace acts as a real traffic trace and is sent to Minos according to its timestamp.

We evaluate Minos against four up-to-date Website Fingerprinting attacks: kNN [48], CUMUL [34], kFP [20], DF [42]. We use the simulation code provided by Gong et al. [18] and keep the suggested parameters from the original papers, but we change the maximum length of traces, which will be illustrated in section 8.3. As for defenses, we also adopt defenses provided by [18]: Tamaraw [10] and WTF-PAD [9]. The former is a heavyweight obfuscation defense, and the latter is a lightweight defense. We compare these two defenses to Minos on overhead and effectiveness.

8.2 Defense Design

Recall that in Minos, traffic morphing-based defense against Website Fingerprinting is only needed when we have a limited number of flows transmitting in an encrypted tunnel. When the number of transmitted flows is large, Minos performs flow-interleaving to obfuscate the flows. When the number of transmitted flows falls below a threshold, it supplements the defense with traffic morphing-based Website Fingerprinting defense. The exact threshold will be examined in section 8.3. In this section, we describe our lightweight defense mechanism.

Based on previous studies, the first few seconds of each trace leak the most useful features for website fingerprinting [20, 50]. [18] proposes FRONT, a defense that dedicates most of its bandwidth budget to obfuscate the trace front. FRONT samples a dynamic traffic window and sends packets based on Rayleigh Distribution, and uses four parameters to control this process. In Minos, we simplify this defense to two parameters: front window W and obfuscating parameter Ω. In the first W packets, we randomly insert dummy packets with a given probability Ω, and in the rest of the traffic, no dummy packets are inserted. To implement the above defense mechanism, we can use the estimated overhead budget and equation 2 to configure the dummy queue. To make this defense as lightweight as possible, we choose not to pad or delay packets and only insert dummy packets at a given possibility.



(a) Normal model Acc (b) Normal model TopN

(c) 5000 model Accuracy (d) 5000 model TopN

Figure 15: Evaluation of flow mixture

8.3 Evaluation

First, we evaluate the relationship between the flow number and the accuracy of attack models. The attacks are trained with the original recommended parameters, except for flow length. To avoid disturbance brought by flow aggregation, we train models at two fixed lengths: 5000 and 10000, and cut off input traffic traces to them. We apply two metrics to evaluate each attack model: the accuracy metric and the TopN metric. In both cases, we select the top N labels with the highest probability, where N equals the number of mixed flows. The accuracy metric evaluates if machine learning models return any right label of the original flows, while the latter metric evaluates how many labels are correct. Each correct label is counted as $1/N$ correct answer.

Figure. 15 shows our results. Figure. 15a and 15b show how metrics change with flow numbers. We can see that both evaluation metrics quickly decrease as the flow number n increases, and converge at n=5. When flow numbers are larger than 5, both metrics are lower than 20%, which is effective and able to defend against any attack models. As for the two graphs evaluating the top 5000 models, although different in exact numbers, the overall tendencies are the same, and also coverage at n=4 or 5. To conclude, mixing up flows can significantly obfuscate attackers with only a few flows. Based on this experiment, we set the threshold of enabling dummy packet insertion to 4, which decreases attackers' accuracy to lower than 20% without any extra overhead.

Then we evaluate our defense mechanism against other defenses. Here, we set the obfuscating parameter Ω to 0.6 and evaluate the defense effect of window sizes of 500 and 1000. The bandwidth overhead of each defense is shown in Table 6. Table 5 shows all evaluation results. We use accuracy, preci-

Table 5: **Defense performances**

Defense	**Accuracy**%				**Precision**%				**F1**			
	kNN	CUMUL	kFP	DF	kNN	CUMUL	kFP	DF	kNN	CUMUL	kFP	DF
No defense	84.62	91.47	91.68	96.9	83.46	90.72	91.56	96.75	0.831	0.9091	0.9147	0.9676
Tamaraw	2.55	4.42	4.99	1.47	3.57	8.18	6.43	5.32	0.0223	0.049	0.0528	0.0206
WTF-PAD	20.38	44.42	57.28	81.06	18.43	44.14	56.11	78.86	0.1766	0.4305	0.5491	0.7857
Minos-500	5.63	39.82	31.73	7.06	7.96	55.68	38.96	6.7	0.05	0.4	0.28	0.05
Minos-1000	7.38	38.73	32.66	6.67	8.49	52.61	38.31	6.7	0.06	0.38	0.28	0.045

Table 6: Overhead of each defense mechanism

Defense	Parameters	Overhead	
		Latency(%)	Bandwidth(%)
Tamaraw	ρ_{out} = 0.04,ρ_{in} = 0.012,L = 50	14.23	143.82
WTF-PAD	Normal rcv	0	60
Minos-500	$\Omega = 0.6$, window=500	0	6
Minos-1000	$\Omega = 0.6$, window=1000	0	12

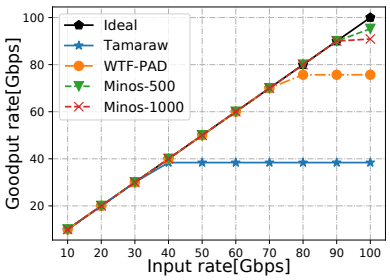


Figure 16: Throughput of each defense.

sion, and F1 score to evaluate defense performances. Without any defense, each attack model can achieve around 90% accuracy, and the DF model has the highest accuracy of 96.9%. Tamaraw decreases the accuracy of each attack to single digits at the cost of its high bandwidth overhead. On the other hand, lightweight defense WTF-PAD fails to defend against DF which still has more than 80% accuracy. In Minos, both defenses achieve less than 40% accuracy and 60% precision. With only one-third of the overhead of WTF-PAD, Minos achieves a satisfying defense effect. Besides, as a practical, lightweight defense mechanism, we are concerned about the actual throughput with each defense. As Figure. 16 shows, the previous defense schemes have lower output rates because of their high bandwidth overhead, even the lightweight defense model WTF-PAD reaches less than 80Gbps of real traffic. In Minos, we can achieve about 95Gbps throughput.

To sum up, although our simple and intuitive Website Fingerprinting defense cannot reduce the accuracy of Website Fingerprinting attacks to random guess, we believe it's acceptable because of its low overhead and high throughput. Besides, in more usual scenarios, we have more than four flows transmitting in one encrypted tunnel on the programmable switch, which can naturally decrease the accuracy of attackers to lower than 20% without any bandwidth overhead.

9 Conclusion

In this paper, we present Minos, a lightweight and scalable defense against traffic analysis attacks using programmable switches, providing both identity anonymity and traffic anonymity. Minos performs line rate packet header encryption with the Proxy Module and performs lightweight dummy packet insertion and packet padding with the Traffic Morphing module. Besides, Minos applies a dynamic flow scheduling algorithm to accommodate different network scenarios. Our evaluation shows that Minos can perform line rate packet encryption and flow scheduling at a low cost, and defend against Website Fingerprinting attacks with little overhead.

10 Acknowledgements

This work is supported by the Major Key Project of PCL under grant No. PCL2023A06, Shenzhen RD Program under grant NO. KJZD20230923114059020, and in part by Shenzhen Key Laboratory of Software Defined Networking under Grant ZDSYS20140509172959989.

References

[1] International data encryption algorithm. https://en.wikipedia.org/wiki/International_Data_Encryption_Algorithm, Accessed August, 2023.

[2] Project v. v2ray. https://www.v2ray.com/, Accessed August, 2023.

[3] Shadowsocks. https://shadowsocks.org/en/download/clients.html, Accessed August, 2023.

[4] Spirent spt-n11u. https://assets.ctfassets.net/wcxs9ap8i19s/7mjZqW5guntME8DD0gf27g/dcc39be517f3296d4ada20b3ecaec900/SPT-N11U_Mainframe_Chassis_Datasheet.pdf, Accessed Janurary, 2024.

[5] Ahmed Alshehri, Jacob Granley, and Chuan Yue. Attacking and protecting tunneled traffic of smart home devices. In *Proceedings of the Tenth ACM Conference on Data and Application Security and Privacy*, pages 259–270, 2020.

[6] Sanjit Bhat, David Lu, Albert Kwon, and Srinivas Devadas. Var-cnn: A data-efficient website fingerprinting attack based on deep learning. *arXiv preprint arXiv:1802.10215*, 2018.

[7] Julia Borghoff, Anne Canteaut, Tim Güneysu, Elif Bilge Kavun, Miroslav Knezevic, Lars R Knudsen, Gregor Leander, Ventzislav Nikov, Christof Paar, Christian Rechberger, et al. Prince–a low-latency block cipher for pervasive computing applications. In *Advances in Cryptology–ASIACRYPT 2012: 18th International Conference on the Theory and Application of Cryptology and Information Security, Beijing, China, December 2-6, 2012. Proceedings 18*, pages 208–225. Springer, 2012.

[8] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, 44(3):87–95, 2014.

[9] Xiang Cai, Rishab Nithyanand, and Rob Johnson. Cs-buflo: A congestion sensitive website fingerprinting defense. In *Proceedings of the 13th Workshop on Privacy in the Electronic Society*, pages 121–130, 2014.

[10] Xiang Cai, Rishab Nithyanand, Tao Wang, Rob Johnson, and Ian Goldberg. A systematic approach to developing and evaluating website fingerprinting defenses. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 227–238, 2014.

[11] Xiaoqi Chen. Implementing aes encryption on programmable switches via scrambled lookup tables. In *Proceedings of the Workshop on Secure Programmable Network Infrastructure*, pages 8–14, 2020.

[12] Trisha Datta, Nick Feamster, Jennifer Rexford, and Liang Wang. {spine}: Surveillance protection in the network elements. In *9th USENIX Workshop on Free and Open Communications on the Internet (FOCI 19)*, 2019.

[13] Roger Dingledine, Nick Mathewson, Paul F Syverson, et al. Tor: The second-generation onion router. In *USENIX security symposium*, volume 4, pages 303–320, 2004.

[14] Shuaike Dong, Zhou Li, Di Tang, Jiongyi Chen, Menghan Sun, and Kehuan Zhang. Your smart home can't keep a secret: Towards automated fingerprinting of iot traffic. In *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*, pages 47–59, 2020.

[15] Chenxin Duan, Hao Gao, Guanglei Song, Jiahai Yang, and Zhiliang Wang. Byteiot: A practical iot device identification system based on packet length distribution. *IEEE Transactions on Network and Service Management*, 19(2):1717–1728, 2021.

[16] Kevin P Dyer, Scott E Coull, Thomas Ristenpart, and Thomas Shrimpton. Peek-a-boo, i still see you: Why efficient traffic analysis countermeasures fail. In *2012 IEEE symposium on security and privacy*, pages 332–346. IEEE, 2012.

[17] Hamid Ghasemirahni, Alireza Farshin, Mariano Scazzariello, Gerald Q. Maguire, Dejan Kostić, and Marco Chiesa. Fajita: Stateful packet processing at 100 million pps. *Proc. ACM Netw.*, 2(CoNEXT3), August 2024.

[18] Jiajun Gong and Tao Wang. Zero-delay lightweight defenses against website fingerprinting. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 717–734, 2020.

[19] Frederik Hauser, Marco Häberle, Mark Schmidt, and Michael Menth. P4-ipsec: Site-to-site and host-to-site vpn with ipsec in p4-based sdn. *IEEE Access*, 8:139567–139586, 2020.

[20] Jamie Hayes and George Danezis. k-fingerprinting: A robust scalable website fingerprinting technique. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 1187–1203, 2016.

[21] Marc Juarez, Mohsen Imani, Mike Perry, Claudia Diaz, and Matthew Wright. Toward an efficient website fingerprinting defense. In *Computer Security–ESORICS 2016: 21st European Symposium on Research in Computer Security, Heraklion, Greece, September 26-30, 2016, Proceedings, Part I 21*, pages 27–46. Springer, 2016.

[22] Neal Koblitz, Alfred Menezes, and Scott Vanstone. The state of elliptic curve cryptography. *Designs, codes and cryptography*, 19:173–193, 2000.

[23] Ding Li, Yuefei Zhu, Minghao Chen, and Jue Wang. Minipatch: Undermining dnn-based website fingerprinting with adversarial patches. *IEEE Transactions on Information Forensics and Security*, 17:2437–2451, 2022.

[24] Guanyu Li, Menghao Zhang, Cheng Guo, Han Bao, Mingwei Xu, Hongxin Hu, and Fenghua Li. {IMap}: Fast and scalable {In-Network} scanning with programmable switches. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 667–681, 2022.

[25] Xiapu Luo, Peng Zhou, Edmond WW Chan, Wenke Lee, Rocky KC Chang, Roberto Perdisci, et al. Httpos: Sealing information leaks with browser-side obfuscation of encrypted flows. In *NDSS*, volume 11, 2011.

[26] Xiaobo Ma, Jian Qu, Jianfeng Li, John CS Lui, Zhenhua Li, Wenmao Liu, and Xiaohong Guan. Inferring hidden iot devices and user interactions via spatial-temporal traffic fingerprinting. *IEEE/ACM Transactions on Networking*, 30(1):394–408, 2021.

[27] Xiaobo Ma, Mawei Shi, Bingyu An, Jianfeng Li, Daniel Xiapu Luo, Junjie Zhang, and Xiaohong Guan. Context-aware website fingerprinting over encrypted proxies. In *IEEE INFOCOM 2021-IEEE Conference on Computer Communications*, pages 1–10. IEEE, 2021.

[28] Roland Meier, Vincent Lenders, and Laurent Vanbever. Ditto: Wan traffic obfuscation at line rate. In *NDSS Symposium*, 2022.

[29] Congcong Miao, Minggang Chen, Arpit Gupta, Zili Meng, Lianjin Ye, Jingyu Xiao, Jie Chen, Zekun He, Xulong Luo, Jilong Wang, and Heng Yu. Detecting ephemeral optical events with optel. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI), Renton, WA, USA, April 4-6, 2022*, pages 339–353. USENIX Association, 2022.

[30] Markus Miettinen, Samuel Marchal, Ibbad Hafeez, N Asokan, Ahmad-Reza Sadeghi, and Sasu Tarkoma. Iot sentinel: Automated device-type identification for security enforcement in iot. In *2017 IEEE 37th international conference on distributed computing systems (ICDCS)*, pages 2177–2184. IEEE, 2017.

[31] Seyed Mohammad Mehdi Mirnajafizadeh, Ashwin Raam Sethuram, David Mohaisen, DaeHun Nyang, and Rhongho Jang. Enhancing network attack detection with distributed and in-network data collection system. In Davide Balzarotti and Wenyuan Xu, editors, *33rd USENIX Security Symposium, USENIX Security 2024, Philadelphia, PA, USA, August 14-16, 2024*. USENIX Association, 2024.

[32] Milad Nasr, Alireza Bahramali, and Amir Houmansadr. Defeating {DNN-Based} traffic analysis systems in {Real-Time} with blind adversarial perturbations. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2705–2722, 2021.

[33] Isaac Oliveira, Emídio Neto, Roger Immich, Ramon Fontes, Augusto Neto, Fabrício Rodriguez, and Christian Esteve Rothenberg. Dh-aes-p4: on-premise encryption and in-band key-exchange in p4 fully programmable data planes. In *2021 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*, pages 148–153. IEEE, 2021.

[34] Andriy Panchenko, Fabian Lanze, Jan Pennekamp, Thomas Engel, Andreas Zinnen, Martin Henze, and Klaus Wehrle. Website fingerprinting at internet scale. In *NDSS*, 2016.

[35] Maximilian Pudelko, Paul Emmerich, Sebastian Gallenmüller, and Georg Carle. Performance analysis of VPN gateways. In *2020 IFIP Networking Conference, Networking 2020, Paris, France, June 22-26, 2020*, pages 325–333. IEEE, 2020.

[36] Jingjing Ren, Daniel J Dubois, David Choffnes, Anna Maria Mandalari, Roman Kolcun, and Hamed Haddadi. Information exposure from consumer iot devices: A multidimensional, network-informed measurement approach. In *Proceedings of the Internet Measurement Conference*, pages 267–279, 2019.

[37] Vera Rimmer, Davy Preuveneers, Marc Juarez, Tom Van Goethem, and Wouter Joosen. Automated website fingerprinting through deep learning. *arXiv preprint arXiv:1708.06376*, 2017.

[38] Ronald L Rivest, Adi Shamir, and Leonard Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[39] Roei Schuster, Vitaly Shmatikov, and Eran Tromer. Beauty and the burst: Remote identification of encrypted video streams. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 1357–1374, 2017.

[40] Naveen Kr Sharma, Chenxingyu Zhao, Ming Liu, Pravein G Kannan, Changhoon Kim, Arvind Krishnamurthy, and Anirudh Sivaraman. Programmable calendar queues for high-speed packet scheduling. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 685–699, 2020.

[41] Sandra Siby, Marc Juarez, Claudia Diaz, Narseo Vallina-Rodriguez, and Carmela Troncoso. Encrypted dns–> privacy? a traffic analysis perspective. *arXiv preprint arXiv:1906.09682*, 2019.

[42] Payap Sirinam, Mohsen Imani, Marc Juarez, and Matthew Wright. Deep fingerprinting: Undermining website fingerprinting defenses with deep learning. In *Proceedings of the 2018 ACM SIGSAC Conference on*

Computer and Communications Security, pages 1928–1943, 2018.

[43] Arunan Sivanathan, Hassan Habibi Gharakheili, Franco Loi, Adam Radford, Chamith Wijenayake, Arun Vishwanath, and Vijay Sivaraman. Classifying iot devices in smart environments using network traffic characteristics. *IEEE Transactions on Mobile Computing*, 18(8):1745–1759, 2018.

[44] Rahmadi Trimananda, Janus Varmarken, Athina Markopoulou, and Brian Demsky. Packet-level signatures for smart home devices. In *Network and Distributed Systems Security (NDSS) Symposium*, volume 2020, 2020.

[45] Yinxin Wan, Kuai Xu, Feng Wang, and Guoliang Xue. Iotathena: Unveiling iot device activities from network traffic. *IEEE Transactions on Wireless Communications*, 21(1):651–664, 2021.

[46] Yinxin Wan, Kuai Xu, Feng Wang, and Guoliang Xue. Iotmosaic: Inferring user activities from iot network traffic in smart homes. In *IEEE INFOCOM 2022-IEEE Conference on Computer Communications*, pages 370–379. IEEE, 2022.

[47] Liang Wang, Hyojoon Kim, Prateek Mittal, and Jennifer Rexford. Programmable in-network obfuscation of dns traffic. In *NDSS: DNS Privacy Workshop*. sn, 2021.

[48] Tao Wang, Xiang Cai, Rishab Nithyanand, Rob Johnson, and Ian Goldberg. Effective attacks and provable defenses for website fingerprinting. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 143–157, 2014.

[49] Tao Wang and Ian Goldberg. On realistically attacking tor with website fingerprinting. *Proc. Priv. Enhancing Technol.*, 2016(4):21–36, 2016.

[50] Tao Wang and Ian Goldberg. {Walkie-Talkie}: An efficient defense against passive website fingerprinting attacks. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 1375–1390, 2017.

[51] David J Wheeler and Roger M Needham. Tea, a tiny encryption algorithm. In *Fast Software Encryption: Second International Workshop Leuven, Belgium, December 14–16, 1994 Proceedings 2*, pages 363–366. Springer, 1995.

[52] Charles V Wright, Scott E Coull, and Fabian Monrose. Traffic morphing: An efficient defense against statistical traffic analysis. In *NDSS*, volume 9, 2009.

[53] Jingyu Xiao, Qing Li, Dan Zhao, Xudong Zuo, Wenxin Tang, and Yong Jiang. Themis: A passive-active hybrid framework with in-network intelligence for lightweight failure localization. *Computer Networks*, 255:110836, 2024.

[54] Jingyu Xiao, Zhiyao Xu, Qingsong Zou, Qing Li, Dan Zhao, Dong Fang, Ruoyu Li, Wenxin Tang, Kang Li, Xudong Zuo, et al. Make your home safe: Time-aware unsupervised user behavior anomaly detection in smart homes via loss-guided mask. In *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 3551–3562, 2024.

[55] Jingyu Xiao, Qingsong Zou, Qing Li, Dan Zhao, Kang Li, Wenxin Tang, Runjie Zhou, and Yong Jiang. User device interaction prediction via relational gated graph attention network and intent-aware encoder. In *ACM International Conference on Autonomous Agents and Multiagent Systems (AAMAS), London, United Kingdom, 29 May-2 June, 2023*, pages 1634–1642. ACM, 2023.

[56] Jingyu Xiao, Qingsong Zou, Qing Li, Dan Zhao, Kang Li, Zixuan Weng, Ruoyu Li, and Yong Jiang. I know your intent: Graph-enhanced intent-aware user device interaction prediction via contrastive learning. *ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies (IMUWT/UbiComp)*, 7(3):1–28, 2023.

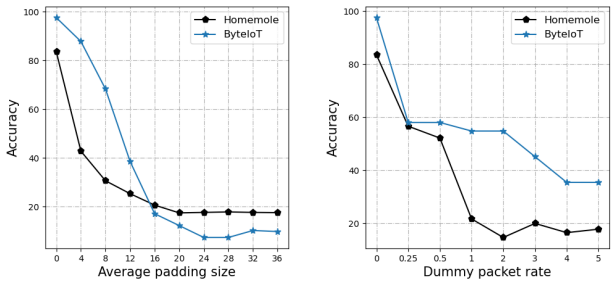
[57] Jingyu Xiao, Xudong Zuo, Qing Li, Dan Zhao, Hanyu Zhao, Yong Jiang, Jiyong Sun, Bin Chen, Yong Liang, and Jie Li. Flexnf: Flexible network function orchestration for scalable on-path service chain serving. *IEEE/ACM Transactions on Networking (ToN)*, 2023.

[58] Sijie Xiong, Anand D Sarwate, and Narayan B Mandayam. Network traffic shaping for enhancing privacy in iot systems. *IEEE/ACM Transactions on Networking*, 30(3):1162–1177, 2022.

[59] Lianjin Ye, Qing Li, Xudong Zuo, Jingyu Xiao, Yong Jiang, Zhuyun Qi, and Chunsheng Zhu. PUFF: A passive and universal learning-based framework for intra-domain failure detection. In *IEEE International Performance, Computing, and Communications Conference (IPCCC), Austin, TX, USA, October 29-31, 2021*, pages 1–8. IEEE, 2021.

[60] Sophia Yoo and Xiaoqi Chen. Secure keyed hashing on programmable switches. In *Proceedings of the ACM SIGCOMM 2021 Workshop on Secure Programmable network INfrastructure*, pages 16–22, 2021.

[61] Hao Zheng, Chengyuan Huang, Xiangyu Han, Jiaqi Zheng, Xiaoliang Wang, Chen Tian, Wanchun Dou, and Guihai Chen. μmon: Empowering microsecond-level network monitoring with wavelets. In *Proceedings of the ACM SIGCOMM 2024 Conference, ACM SIGCOMM 2024, Sydney, NSW, Australia, August 4-8, 2024*, pages 274–290. ACM, 2024.



(a) Impact of different padding sizes.

(b) Impact of different dummy packet rates.

Figure 17: Accuracy of attacks against different defenses

[62] Qingsong Zou, Qing Li, Ruoyu Li, Yucheng Huang, Gareth Tyson, Jingyu Xiao, and Yong Jiang. Iotbeholder: A privacy snooping attack on user habitual behaviors from smart home wi-fi traffic. *ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies (IMWUT/UbiComp)*, 7(1):1–26, 2023.

[63] Xudong Zuo, Qing Li, Jingyu Xiao, Dan Zhao, and Jiang Yong. Drift-bottle: a lightweight and distributed approach to failure localization in general networks. In *ACM International Conference on emerging Networking EXperiments and Technologies (CoNEXT), Roma, Italy, December 6-9, 2022*, pages 337–348. ACM, 2022.

A Use case: IoT Fingerprinting

In this section, we adapt Minos to defend IoT Fingerprinting, which utilizes different features from Website Fingerprinting attacks. We apply public dataset from [36] and develop our defense against the state-of-the-art HomeMole [14] and ByteIoT [15]. All the experiments in this section are simulated in software. We evaluate the accuracy of both model against different padding sizes and dummy packet rates.

In Figure. 17a, we pad packets to different sizes to evaluate the impact of packet padding. The result shows that an average padding of 16 Bytes can reduce the accuracy of both attacks to 20%, with 98% goodput rate. On the contrary, inserting dummy packets is less effective. Figure. 17b shows the impact of inserting dummy packets with different rates, from adding 1 dummy packet for every 4 real packets (0.25), to adding 5 dummy packet for each real packet (5). The result shows that only with large amounts of dummy packets can we reduce the accuracy of ByteIoT to less than 40%.

That is because IoT Fingerprinting attacks utilizes packet length information with timing and directions, which makes packet padding more effective than dummy packet insertion based defense like [9]. For example, ByteIoT utilizes the frequency distribution of packet lengths, so that a large number of dummy packets has to be inserted to obfuscate the original frequency distribution. The results shows that different defense mechanisms are required to defense different traffic analysis attacks, and the adaptability of Minos can satisfy this need well.

GeneralSparse: Bridging the Gap in SpMM for Pruned Large Language Model Inference on GPUs

Yaoyu Wang[1,2,†], Xiao Guo[1,2,†], Junmin Xiao[1,2], De Chen[1,2], Guangming Tan[1,2,*]
[1]*SKLP, Institute of Computing Technology, CAS,* [2]*University of Chinese Academy of Sciences*

Abstract

The rapid growth of generative model parameters poses challenges in deployment, especially regarding weight storage and inference latency. The weight pruning is an effective technique to reduce the computational and memory overhead of Large Language Models (LLMs) while maintaining accuracy, which transforms the matmuls to Sparse Matrix Multiplication (SpMM) computation. However, the diverse pruning methods introduce varying sparsity patterns that challenge high-performance SpMM on GPUs. Existing solutions are limited with adaptability to these patterns, flexibility in handling different sparsity levels, and support for efficient optimizations.

In this work, we present GeneralSparse, a novel solution that bridges this gap by leveraging the abstraction of memory access and reduction spaces. GeneralSparse designs the process of dividing box to adapt dynamically to diverse pruning patterns and proposes hierarchical reduction algorithms tailored to GPU hierarchies. Through evaluations on pruned LLM weight matrices and the SuiteSparse collection, GeneralSparse achieves up to 20.82× speedup over cuSPARSE libraries. At end-to-end inference time on LLMs, GeneralSparse achieves up to 2.33× speedup over counterparts.

1 Introduction

Generative models, particularly Large Language Models (LLMs), have achieved remarkable success in various natural language processing tasks [4, 35], such as machine translation [4, 37], text summarizing [50, 51], etc. However, with the rapid growth of the parameter size, it becomes increasingly challenging to efficiently deploy these models. On one hand, their weights could be too large to be placed on GPUs. On the other hand, LLMs usually cause very high inference latency even using multiple GPUs as large amounts of computation and memory access are required [27, 31, 39].

The weight pruning methods [12, 15, 43] have been demonstrated to be effective in reducing memory usage and computations for model inference while retaining good accuracy. There are plenty of pruning algorithms focused on accelerating tensor operations such as matrix-matrix multiplications, which form **Sparse Matrix Multiplication (SpMM)** computation. Different pruning methods remove elements from different positions in the weight matrix, resulting in varying sparsity patterns in the matrix. However, because of the diversity of pruned matrix sparsity patterns, and the close association between pruned matrix sparsity and the SpMM performance, it is unrealistic to find a one-fits-all method to achieve high performance for all pruned weight matrices.

Many studies focus on improving the performance of SpMM on GPUs in the past decades from artificial designs, auto-tuners, and domain-specific compiler technologies. Artificial design methods [16, 22, 23] rely on human expertise to design optimization techniques. Traditional auto-tuners [9, 28, 46] are designed to select the most appropriate methods for a given sparse matrix from a set of artificial candidate methods. Compilers technologies [6, 13, 26] generate the specific computation codes for each given input.

Table 1: Comparison with state-of-the-art works on GPUs.

Work		Adaptability		Support
		Sparsity Pattern①	Sparsity Level②	SpMM Auto-Implement
Artificial Designs	ASpT [23] Sputnik [16] SparTA [53]	✘	✘	✘
Traditional Auto-tuners	Xin et al. [46] DgSPARSE [9] EC-SpMM [28]	✔	✘	✘
Compiler Technologies	TVM [6]	✘	✘	✘
	AlphaSparse [13] TACO [26]	✔	✘	✘
Intelligent Auto-tuner	GeneralSparse	✔	✔	✔

① **Sparsity Pattern** means the position distribution of non-zero elements.
② **Sparsity Level** means the percentage of zero elements in sparse matrix.

Despite extensive efforts to improve the SpMM performance, limitations remain evident in current research across three aspects of program design methodology (Table 1). Detailed explanation as follows:

*Guangming Tan is the corresponding author: tgm@ict.ac.cn
†These authors contributed equally to this work.

Limitation1 in Sparsity Pattern: The adaptability to matrices with diverse sparsity patterns using different pruning methods of LLMs. Artificial design methods usually use fixed memory access optimization strategies for a specific type of sparsity pattern. ASpT [23] uses column sorting for sparse matrix with localized dense features. Sputnik [16] is proposed for sparse matrix with little variation in row length. SparTA [53] is applied for sparse matrix with structured pruning. However, each method targets specific application scenarios, lacking adaptability to sparsity patterns resulting from various pruning methods (Figure 4).

Limitation2 in Sparsity Level: The adaptability between efficient reduction algorithms and varied pruned sparsity levels at different depth layers of LLMs. For maintaining model accuracy, the weight matrices of different depth layers in LLMs are usually pruned to different sparsity levels (e.g, 70% at the bottom and top layers, 90% at intermediate layers) [2, 38, 49], while matrices of different sparsity levels require corresponding efficient reduction methods in SpMM. In auto-tuners, Xin et al. [46] and EC-SpMM [28] use thread sequential reduction for all sparse matrices. DgSPARSE [9] designs two reduction algorithms, but it also ignores the interaction between reduction algorithms and sparsity levels.

Limitation3 in SpMM Automatic Implementation: The support of automatic code implementations to cover diverse optimizations. Recent works [9, 23–25, 47] implement the SpMM program as the predefined complete program templates, required a significant amount of human time and effort to design. Current compiler technologies works, such as TVM [6] and AlphaSparse [13], do not support SpMM, utilized for dense tensor operations and SpMV respectively. TACO [8] generates the program for sparse tensor algebra operations on CPU, which is not specifically optimized for GPU. Currently, there is still a lack of research for automatically generating high-performance SpMM programs on GPUs.

To address the above three limitations, we abstract the memory access to dividing box strategies for dynamically adapting to diverse sparsity patterns of pruned matrix and form the **memory access space**. It employs a multi-level **reduction space** aligned with GPU hierarchies, enabling efficient handling of varying sparsity levels. Additionally, an **automated kernel generator** generates SpMM kernels for specific sparse matrices, reducing manual effort for implementing the program. These innovations collectively overcome the constraints of existing methods for GPU-based SpMM computation.

In this work, we present GeneralSparse to address the challenges of efficient SpMM for pruned LLMs on GPUs. The key contributions of this work are summarized as follows:

- GeneralSparse proposes the memory access space (§3.2) and reduction space (§3.3), which allows for efficient handling of diverse sparse matrix patterns and sparsity levels.
- GeneralSparse designs a cost model (§3.4) and implements an efficient code generator (§3.5), which saves the time of program development for diverse pruned methods.
- GeneralSparse demonstrates up to 20.82× speedup over existing SpMM libraries on pruned weight matrices of LLMs and the SuiteSparse collection, and achieves up to 2.33× speedup over counterparts on inference time of LLMs.

2 Background and Motivation

2.1 Generative Model Inference

Inference performance hotspot of LLMs. Figure 1a depicts the typical decoder architecture of a single layer in LLMs. Within the decoder layer, there are four primary matrix multiplications (matmuls): QKV Projection, Output Projection, MLP1, and MLP2. The inference performance of LLMs is significantly constrained by these four matmuls. Related works [1] and our experiments reveal that these matmuls account for about 80% of the end-to-end execution time. Although the pruning method (magnitude [20]) has been applied to network weights and matmuls have been replaced with the SpMM computation (implemented via cuSPARSE library), Figure 1b shows SpMM still remains the performance bottleneck **(about 70%)** and has room for further optimization.

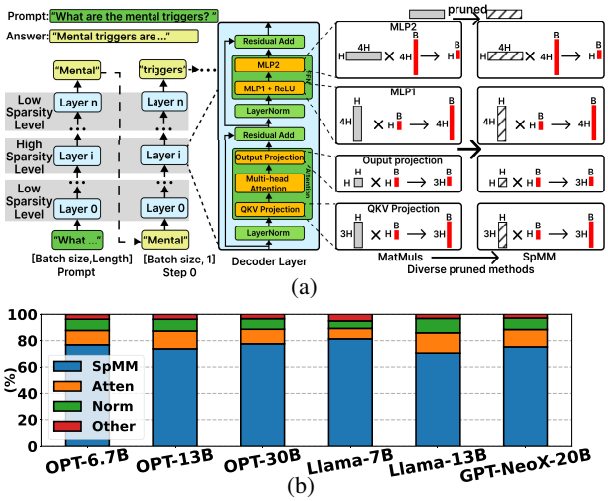


Figure 1: (a) Network architecture of LLMs. The **H** and **B** mean the hidden dimension and inference batch size. (b) The time proportion of inference occupied by different kernels in the pruned model at 8 batch size.

Diverse sparsity patterns by different pruning methods. Many pruning methods have been proposed for LLMs, which are usually classified into structured [5, 7] and unstructured pruning [20, 40]. In practice, unstructured pruning typically retains better accuracy than more restrictive structured pruning [15, 18, 21]. Our approach is applicable to both types, with a primary focus on unstructured pruning. Besides, we give the examples of two well-known unstructured pruning methods, random [3] and magnitude [20]. Random prunes values randomly and results in a uniform non-zero element distribution per row, while magnitude prunes values with smaller absolute

values of matrix weight and leads to an unpredictable and irregular distribution. Sparsity patterns caused by these two pruning methods result in performance fluctuation in Figure 4.

Varied sparsity levels across layers at different depths in LLMs. Current works [2, 38, 49] observe that layers at different depths have different sensitivities to parameter pruning. To maintain accuracy, the sparsity level of pruning varies with the layer depth. For example in Figure 1a, the bottom and top layers show higher sensitivities (low sparsity level), while the intermediate layers have lower sensitivities (high sparsity level) [38]. This variance in sparsity level across layers poses additional challenges for efficient SpMM computation, as different sparsity levels demand corresponding tailored optimization strategies.

2.2 SpMM

SpMM (Sparse Matrix Multiplication) multiplies an $M \times K$ sparse matrix A and an $K \times N$ dense matrix B to output a $M \times N$ dense matrix C (i.e., C = AB). Figure 2 shows the SpMM computation process, where using the CSR format to store sparse matrix. In SpMM, there are two key factors that affect performance on GPUs. ***One aspect*** **is how to distribute three loops ($M/N/K$ dimension) of sparse and dense matrix to GPU process units for parallel memory access.** Especially sparse matrices have diverse non-zero element distributions and a specially designed storage format, which leads to load unbalance and irregular memory access latency. ***The other aspect*** **is how to accumulate (reduce) intermediate results with reduction dependencies of K-dimension.** The number of reduction results varies with different sparsity levels of sparse matrix, which affects the computation efficiency of GPU process units. Many fine-grained analyses [9, 28] on SpMM suggest that diverse optimization strategies for distinct sparse matrices are necessary to achieve high performance on GPUs.

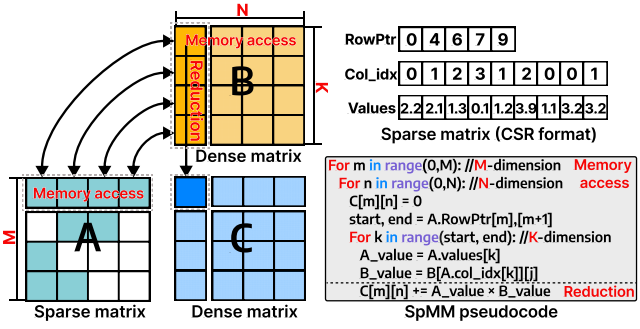


Figure 2: SpMM computation and pseudocode.

2.3 Motivation

The motivation of GeneralSparse comes from two observations, which separately show its necessity and feasibility.

Observation1. Parallel memory access strategies can be abstracted as the process of dividing boxes for diverse sparsity patterns. The parallel memory access strategies are to distribute regions of sparse and dense matrix to parallel process units on GPUs, which can be abstracted as the process of dividing boxes. Sparse and dense matrices are divided consecutively at the block/warp/thread level. Then, dividing strategies of sparse and dense matrices are integrated to form the memory access strategy (e.g., Sputnik [16] in Figure 3). However, existing SpMM methods on GPUs usually use the fixed parallel memory access strategy for all sparse matrices.

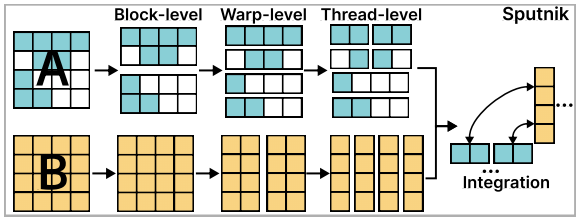


Figure 3: Parallel memory access strategies are abstracted as the process of dividing box at the block/warp/thread-level.

Based on the above observation, the **memory access space** can be constructed, where the sparse matrix is adaptively and fine-grained divided for diverse sparsity patterns, and the dense matrix is regularly divided by column. Finally, different division strategies of sparse and dense matrices are integrated. The memory access space covers enormous and fine-grained box division strategies, which makes it challenging to implement this vast space. As shown in Figure 4, the performance of methods varies across different weight matrices, and new strategies in memory access space achieve higher performance.

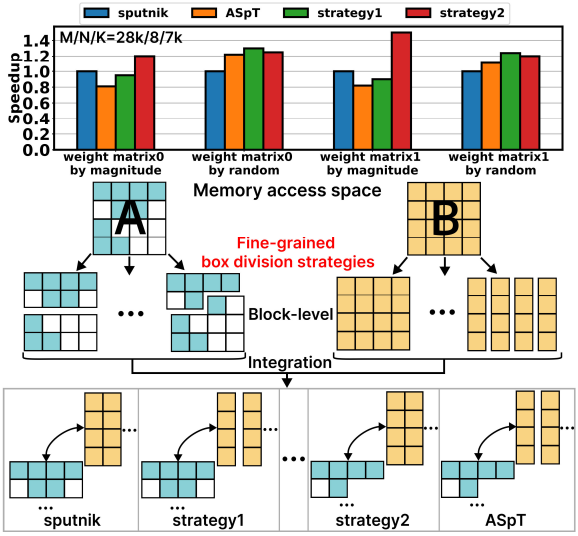


Figure 4: Memory access space is composed of fine-grained box division strategies of the sparse and dense matrix. For simplicity, warp/thread-level is not shown here.

Observation2. Multi-level reduction algorithms can be used in combination for varied sparsity levels in LLMs. The current methods usually use a thread to sequentially reduce the results, while it limits the GPU parallel computation

efficiency, especially in cases of low sparsity levels. As illustrated in Figure 5, we also use the warp-level instruction to reduce results, and the performance of two reduction algorithms varies with sparsity levels, indicating a single reduction algorithm cannot achieve optimal performance in all cases.

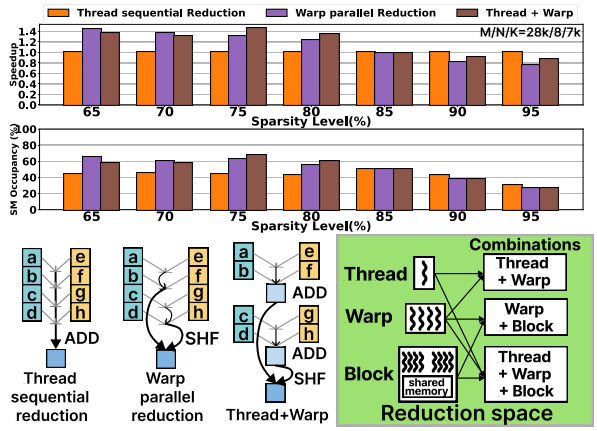


Figure 5: Performance comparison between reduction algorithms, and the reduction space based on GPU hierarchy.

Due to the performance effect between sparsity levels and reduction algorithms, a hierarchical **reduction space** is required to adapt to different matrix sparsity levels based on GPU hierarchy. Furthermore, the reduction algorithms can also be used **in combination of multi-levels thread/warp/block on GPU** (e.g., Thread+Warp in Figure 5), which is challenging to implement these flexible combinations of reduction algorithms. Therefore, there is a need to explore and design efficient reduction strategies that can dynamically select or combine reduction algorithms to fully utilize hardware resources.

3 GeneralSparse Design

3.1 Overview

We propose GeneralSparse, which automatically designs and implements high-performance SpMM programs on GPUs, as illustrated in Figure 6. It consists of four parts: the memory access space, the reduction space, the cost model, and the code generator. The pruned sparse matrix and dense matrix are initially processed within the memory access space. Subsequently, reduction algorithms are selected and applied in the reduction space. Through the cost model for the optimal solution, the code generator generates the SpMM program.

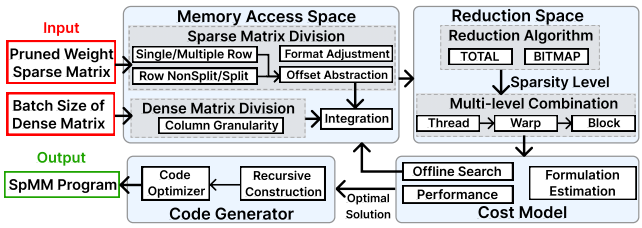


Figure 6: Overview of GeneralSparse.

3.2 Memory Access Space

The memory access space covers fine-grained division strategies of sparse and dense matrix, representing that distributing sparse and dense matrix to process units for parallel memory access. In some cases, we use the concept of **worker** to represent parallel process units (thread/warp/block) on GPUs.

In memory access space, we design an orthogonal division of sparse matrices (row-based and split-based) and a column-based division of dense matrices. Then, we use offset abstraction for sparse matrix division and perform format adjustments. This allows for adaptive handling of diverse sparsity patterns and integration of sparse and dense matrix.

Design rationales for row-based and split-based. For improving the performance, there are two design rationales: reducing memory access latency and improving GPU efficiency. For reducing memory access latency, row-based reduces row index accesses, while split-based reduces non-zero subscript accesses. For GPU parallelism, if non-zeros concentrate in a few columns per row, multi-row row-based processing improves efficiency by enabling simultaneous column access. If non-zero counts vary greatly across rows, split-based achieves better load balancing.

Orthogonal dimension in sparse matrix division (allocation). In sparse matrix allocation, we design two orthogonal dimensions, including row-based and split-based dimensions as depicted in Figure 7a. Row-based dimension is considered from the perspective of GPU processing units, divided into `single/multiple-row`. Split-based dimension is considered from the matrix perspective, divided into `row-nonsplit/split`.

Single/multiple-row dimension means non-zero elements of single/multiple rows in the sparse matrix are processed by a worker. `Single-row` is beneficial when the matrix rows have distinct characteristics and require individual processing. For example, in matrices with highly irregular row-wise non-zero element distributions, processing one row at a time can simplify the data handling and computation process. `Multiple-row` takes advantage of the parallel processing capabilities of GPUs. When there is a certain similarity or correlation among adjacent rows, processing multiple rows together can significantly reduce the overhead of data fetching and processing, thereby improving the overall efficiency.

Row-nonsplit/split dimension means whether a row of the sparse matrix is split and processed by different workers. In `row-nonsplit`, each row of the sparse matrix is kept intact during processing. This ensures that the internal structure of each row is preserved, which can simplify memory access patterns and reduce the complexity of data management. It is useful for matrices with a relatively uniform row-wise structure. The `row-split` means dividing each row of the sparse matrix into smaller segments. This allows for more fine-grained processing, especially for matrices with highly irregular row structures. By splitting the rows, it becomes

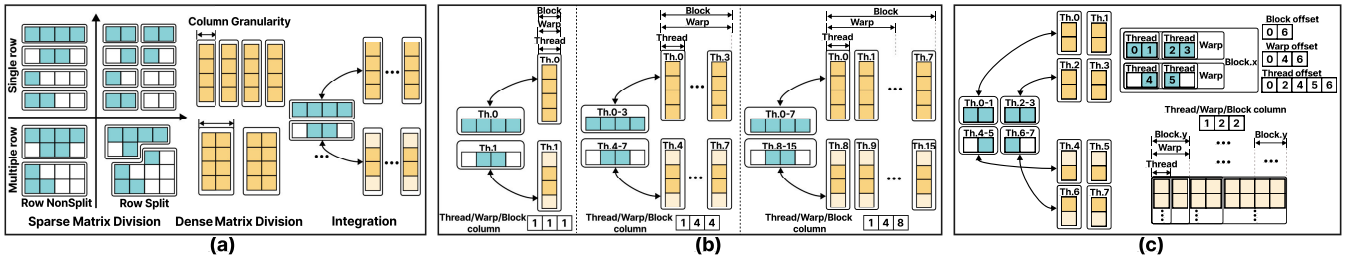


Figure 7: (a) Division of sparse matrix and dense matrix, and integration in SpMM computation. (b) Integration between sparse and dense matrices using thread, warp, and block hierarchies. (Assume each warp/block has 4/8 threads). (c) Offset abstraction for sparse matrix division, along with integration.

possible to better balance the workload among different processing units, resulting in improved performance.

Dense matrix division (allocation). The dense matrix allocation mode pertains to the organization of dense matrices. The dense matrix is distributed to workers with a uniform pattern of data block in column granularity, because computations between columns are independently parallel. An important aspect is the adjustable column granularity of allocation. The number of columns grouped for a particular allocation unit can be changed. For example in Figure 7a, we respectively group 1 and 2 columns into a single allocation unit. By adjusting this grouping of columns, we can optimize the memory access patterns and data transfer operations.

Integration between sparse and dense matrix. Integration of Figure 7a illustrates the column-based relationship between sparse and dense matrices. The arrow indicates the mapping between data blocks of the sparse matrix and those of the dense matrix. In Figure 7b, the dense matrix is divided into 1 column for thread (`thread column = 1`) and each thread processes one row of sparse matrix. When adjacent consecutive threads process different columns of dense matrix (`warp/block column > 1`), threads load the same non-zero elements in sparse matrix but different columns of dense matrix, which forms the `thread-groups` (e.g., Th.0-3 and Th.4-7 in middle of Figure 7b). It achieves data reuse in sparse matrix and coalesced access in dense matrix. When adjacent consecutive warps/blocks process the same non-zero elements in sparse matrix, it also forms `warp/block-groups`.

Offset abstraction of adaptable division in sparse matrix. We use a hierarchical approach with `thread/warp/block offset` for sparse matrix division, which is aligned with `column-based` integration in dense matrix. In Figure 7c, dense matrix is distributed with `Block.y` and two consecutive threads process different columns (e.g., thread-groups of Th.0-1). Sparse matrix is distributed with `Block.x`, cooperated with `Block.y` of dense matrix. Besides, sparse matrix is allocated with thread-groups by single-row and row-split at thread-level, and allocated with warps by single-row and row-nonsplit at warp-level (etc. in block-level). `Thread offset` indicates the starting position of processing non-zeros range by thread-groups (e.g., 0 and 2 of Th.0-1 and Th.2-3). `Warp/Block offset` indicates the starting position of warps/blocks.

Design factors of offset abstraction. When designing the abstract representation of GPU allocation for sparse matrices, two crucial factors need to be taken into account. ***First***, it should comprehensively cover the orthogonal allocation dimensions of the sparse matrix (including single/multiple-rows and row-split/nonsplit) to adapt to different structures and optimize memory access. Regarding covering the orthogonal allocation dimensions, offsets can clearly locate the positions of different rows or row segments, effectively implementing data processing under different strategies. ***Second***, it must align with the integration of the dense matrix. The change of `warp/block column` affects the number of threads in the thread-group, as shown in Figure 7c. So, the `thread offset` is based on thread-groups, not individual threads. `Warp/block` offset is also based on warp/block-groups, if the adjacent warps/blocks process the same non-zero elements in the sparse matrix.

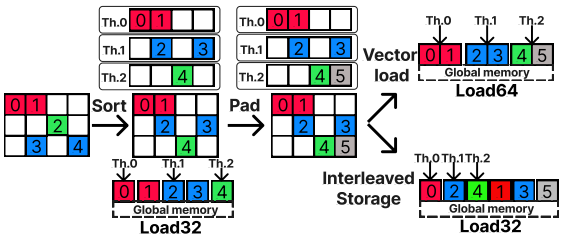


Figure 8: Format adjustments of sparse matrix with thread allocation for access.

Format adjustment is centered around optimizing the memory access layouts of sparse matrices, which encompasses several crucial operations in Figure 8. `Sort` is employed to reorder the elements, leading to more regular and predictable memory access patterns. `Pad` ensures proper data alignment, especially considering the alignment requirements of the hardware. `Interleaved storage` stores data in a way that enables multiple elements to be loaded in a single coalesced memory access. Besides, **vector instruction** is utilized for memory access. For dense matrices, they allow simultaneous loading of multiple columns, maximizing memory bandwidth utilization and enhancing data throughput. For sparse matrices, adjacent threads can also load data in a coalesced manner, minimizing redundant memory accesses in Figure 8.

Workflow of memory access space. Based on our abstraction of the memory access, ***first***, the columns of dense matrix are divided three times (`block/warp/thread column`), determining the threads number in `thread-groups` of sparse matrix. ***Second***, the non-zero elements of sparse matrix are divided three times. The non-zero elements are divided into different blocks (`block offset`). The non-zero elements within the blocks are further divided into warps (`warp offset`) and threads (`thread offset`). ***Third***, format adjustment and vector instructions are employed to fine-tune the memory access.

3.3 Reduction Space

Reduction space represents the reduction (accumulation) algorithms of intermediate results. We design reduction space from two aspects. ***First***, **reduction algorithms can be used in combination, with consideration of the pruning sparsity levels and the GPU hierarchy.** The thread-level reduction can use fine-grained parallelism and is suitable for higher sparsity levels, while warp/block-levels can handle more data and suitable for lower sparsity levels. The reduction algorithms can be used in combination for diverse pruning sparsity levels. ***Second***, **reduction algorithms need to be aligned with the memory access space, especially irregular sparse matrix allocation.** The memory access space determines data organization and access in memory, so the allocation of sparse and dense matrices affects the reduction method choice.

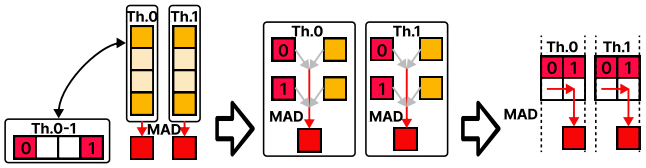


Figure 9: Simplified illustration of reduction methods.

Simplify figure expression of the reduction algorithm. To describe reduction algorithms more clearly, we show the example in Figure 9. It depicts that two threads (Th.0 and Th.1) use Multiply-Add (MAD) operations to reduce the results of two adjacent columns corresponding with the same row. The elements of the dense matrix are removed for clarity.

Reduction algorithms are designed from GPU hierarchy, and aligned with single/multiple-row dimension. We design reduction algorithms from three-levels of GPU, and propose `TOTAL` and `BITMAP` algorithms which are aligned with `single/multiple-row` dimension. For single-row dimension, each row can be independently processed by a thread or a small set of threads. So it leads to a relatively simple and faster `TOTAL` algorithm. For multiple-row dimension, it enables the distribution of non-zero elements across rows, thereby achieving better load balancing among the processing units. However, it incurs additional overhead, because `BITMAP` algorithm must handle reductions that span across rows, necessitating the determination of element row membership.

TOTAL Reduction Algorithm. We introduce the `TOTAL` algorithm from three-levels of GPU corresponding with single-row dimension. Figure 10(a) shows that `THREAD_TOTAL` uses the MAD operation and registers to sequentially reduce the results of non-zero elements within the same row. Figure 10(c) shows that `WARP_TOTAL` uses the warp-level shuffle instruction and registers to reduce results processed by a warp. Figure 10(e) shows that `BLOCK_TOTAL` uses MUL and ADD instructions on block-level shared memory to reduce results.

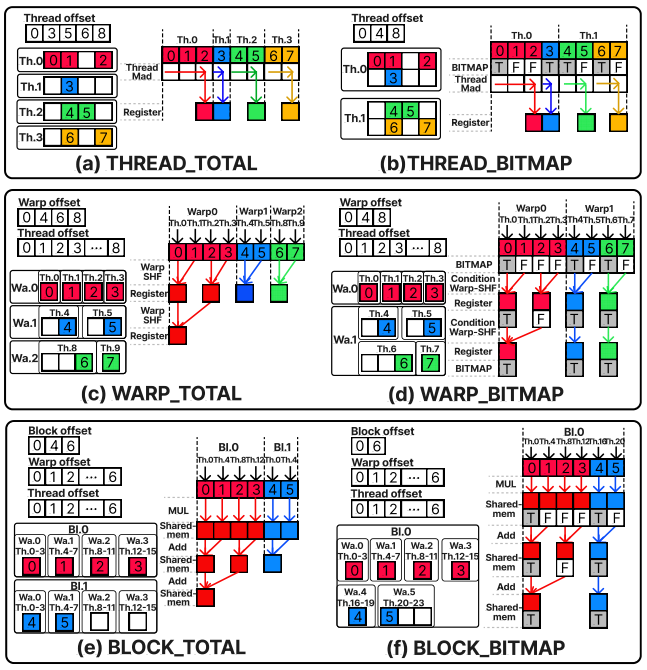


Figure 10: TOTAL and BITMAP reduction example. Thread to warp/block level are suitable for sparsity from high to low.

BITMAP Reduction Algorithm. We propose the `BITMAP` algorithm from three-levels of GPU corresponding with multiple-row dimension. It uses the data structure of a BITMAP to judge whether the non-zero elements span rows. Each binary bit of BITMAP represents a non-zero element, `T` represents the starting position of the row, and `F` represents the positions within the row. Figure 10(b) shows that `THREAD_BITMAP` uses the BITMAP to determine whether a thread continues accumulating the current result. If the next step crosses rows, the result will be written back directly. In warp/block level (Figure 10(d,f)), multiple threads perform reduction in parallel, detailed in Algorithm 1.

Multi-level combination of reduction algorithms can be used on GPU for diverse sparsity levels, which are aligned with row-split/nonsplit dimension. The reduction algorithms of three-levels can be used in combination from thread to warp/block. In row-nonsplit dimension, only a single processing pass is required to reduce all values within a row. This is because the entire row is processed intact, and the reduction can be finished straightforwardly. Conversely, in row-split dimension, the reduction values of the row are divided into several segments, so an extra step of reduction

Algorithm 1 Parallel BITMAP Reduction Algorithm

Require: Array *Result* of length L, $BITMAP$ of length L.
$n = \lfloor \log_2(L) \rfloor$, $step = 1$, $BITMAP_{init} = BITMAP$
for $i = 0$ to n **do**
  $step = step \times 2$
  for $j = 0$ to $L - 1 - step$ **in parallel do**
    if $BITMAP[j + step] = F$ **then**
      $Result[j] = Result[j] + Result[j + step]$
      $BITMAP[j + step] = BITMAP[j] \vee BITMAP[j + step]$
    end if
  end for
end for
for $i = 0$ to $L - 1$ **in parallel do**
  if $BITMAP_{init}[i] = T$ **then**
    return $Result[i]$
  end if
end for

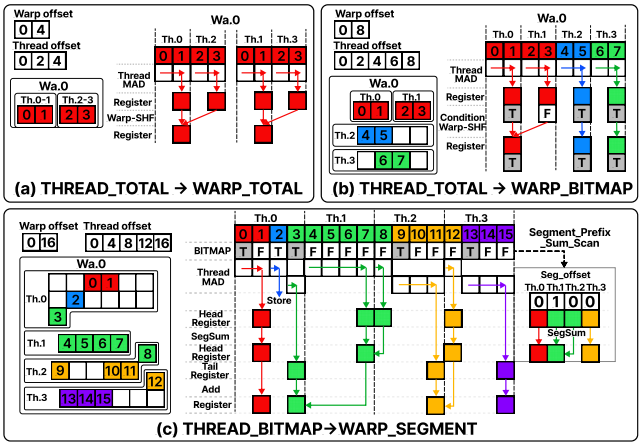


Figure 11: Multi-level combination examples of reduction.

becomes necessary to consolidate these segments. For example, after an initial reduction at a low-level (e.g., thread-level), further reductions at a high-level (e.g., warp-level) are needed.

Low-level TOTAL to High-level TOTAL. In Figure 11(a), it shows the example of `THREAD_TOTAL` and `WARP_TOTAL`. Thread-level allocation is in `single-row` and `row-split` dimension, which corresponds with the `THREAD_TOTAL` to reduce results, and needs further warp-level reduction. Warp-level allocation is in `single-row` and `row-nonsplit` dimension, which corresponds with `WARP_TOTAL`, and does not need further block-level reduction.

Low-level TOTAL to High-level BITMAP. In Figure 11(b), it shows the example of `THREAD_TOTAL` and `WARP_BITMAP`. Thread-level allocation is in `single-row` and `row-split dimension`, which corresponds with `THREAD_TOTAL` to reduce results, and needs further warp-level reduction. Warp-level allocation is in `multiple-row` and `row-nonsplit`, which corresponds with the `WARP_BITMAP`, and does not need further block-level reduction.

Low-level BITMAP to High-level SEGMENT. `SEGMENT` algorithm (Algorithm 2) is proposed to process the remaining intermediate results of low-level `BITMAP`. In Figure 11(c), `THREAD_BITMAP` produces two intermediate results: the remaining result of the non-zero elements of the previous row and the head part of the non-zero elements of the next row, which need to further reduced by `WARP_SEGMENT`. `WARP_SEGMENT` algorithm uses `Seg_offset` to perform reduction in head part, then adds the head and tail results together.

Algorithm 2 Parallel SEGMENT Reduction Algorithm

Require: Array $HeadRegister$ of length T, Array $TailRegister$ of length T, $BITMAP$ of length L.
$n = L/T$
Seg_offset = Segment_Prefix_Sum_Scan($BITMAP$)
for $i = 0$ to $T - 1$ **in parallel do**
  $HeadRegister[i]$ = SegSum($HeadRegister$[i:i+Seg_offset[i]])
  $Result[i] = HeadRegister[i] + TailRegister[i]$
end for
return *$Result$

Function Segment_Prefix_Sum_Scan(*$BITMAP$):
  malloc(*Seg_offset, T)
  for $i = 0$ to $T - 1$ **in parallel do**
    Seg_offset[i] = False
    for $j = 0$ to $n - 1$ **do**
      Seg_offset[i] = Seg_offset[i] $\vee$ BITMAP[$i \times n + j$]
    end for
    Seg_offset[i] = 1 - Seg_offset[i]
  end for
  Cumu_sum(*Seg_offset) // Accumulate the positions with
  // consecutive values of 1 to the first position
  return *Seg_offset
End Function

Rule-based selections of reduction algorithms considering sparsity levels and sparse matrix allocation. The selection process is structured into three hierarchical layers in Figure 12 and each layer has the judge of `single-row` and `row-split` dimension of sparse matrix allocation. The `*offset range` is calculated by `*offset[i+1] - *offset[i]`. When the weight matrix is pruned at the high sparsity level, we set the `thread offset range` ≥ 2, so a thread reduction algorithm is applied. Conversely, when the pruned weight matrix at the low sparsity level, we set the `thread offset range` < 2, so it skips to warp/block-level, and warp/block reduction algorithms are utilized.

3.4 Cost Model

Workflow of GeneralSparse. Based on our abstraction of the memory access and reduction space, ***first***, the sparse matrix and dense matrix are consecutively divided and integrated at block/warp/thread levels for distinct sparsity patterns. ***Second***, the reduction algorithm is selected based on the rules, considered from sparsity levels. ***Third***, through offline traversal search and the cost model for performance estimation, the optimal solution is selected.

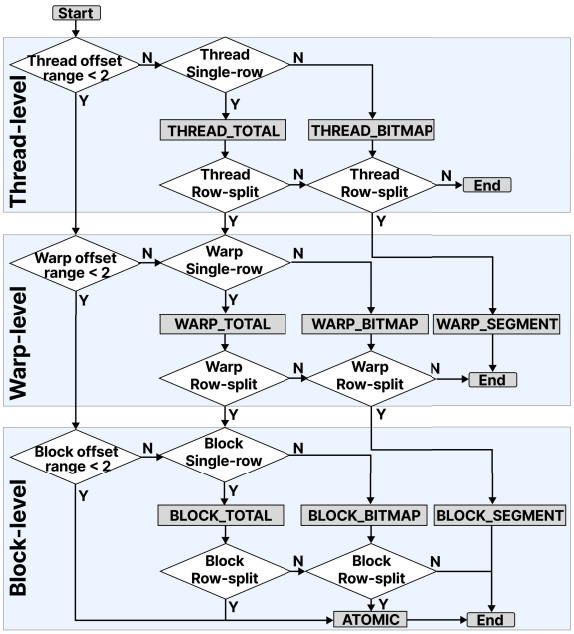


Figure 12: Rule-based selections of reduction algorithms from GPU structures. The `offset` is set by sparsity levels.

Problem definition. Based on the workflow of GeneralSparse, the SpMM computation on GPUs can be expressed as a combinatorial problem of divisions (partitions). Consider NNZ non-zero elements of $M \times K$ sparse matrix and N columns of dense matrix that are divided into several groups. We summarize a variety of division strategies and each division (memory access) strategy incurs a specific cost c. Each division strategy is with its post-processing (reduction) cost d. The objective is to select a combination of division and post-processing strategies that minimizes the total cost of division and post-processing costs $c+d$.

Cost estimation for division and reduction strategies. We collect the sparse characteristics of the sparse matrix (e.g., the average number of non-zeros per row), the column number of dense matrix and etc, as input, and fit the cost estimation using linear regression. The cost functions include the following aspects. `Division (memory access) cost` (c_{ij}): Estimated based on workload distribution between parallel computation units. `Post-processing (reduction) cost` (d_{ij}): Estimated based on the number and type of arithmetic operations, including read and write operations of reduction results. The goal is to minimize the total cost:

$$\min_{i\in I, j\in J} \quad x_{ij}(c_{ij}+d_{ij}) \tag{1}$$

subject to:

$$x_{ij} = 0, \quad \text{if exceeding hardware limitations} \tag{2}$$

Offline search solution for optimal strategy. We solve this problem offline by a traversal search. ***First***, we generate the valid strategy combinations for dense and sparse matrices, and calculate costs for each combination, factoring in division and post-processing costs. ***Second***, identify the combination with the minimum cost as the optimal strategy. The offline search for the cost is lightweight compared with long-time running inference services, because the optimal strategy can be reused in the network of LLM.

Table 2: Variable Definitions

Variable	Type	Description
x_{ij}	Binary	1 if the division strategy i of dense matrix and j of sparse matrix is selected; 0 otherwise.
c_{ij}	Real	Division cost associated with the ij-th division strategy.
d_{ij}	Real	Post-processing (reduction) cost associated with the ij-th division strategy.

Cost model explanation. For designing, our cost model is a linear regression model, to estimate the running performance of a specific matrix under a specific combination of strategies. For training, we use the characteristics of the sparse matrix (e.g., row numbers, column numbers, average non-zero elements per row and etc.) as the input for linear regression. We then use this model to fit the actual running performance of the sparse matrices in the training-set and the accuracy rate is approximately 90%. For dependency, there are different linear regression formulas for each specific memory access/reduction algorithm to estimate the actual running performance. Therefore, each formula of cost model depends on the specific memory access/reduction algorithm. Based on the estimated cost, we select the optimal memory access strategy and reduction combination for each sparse matrix.

3.5 Code Generator

The code-generator needs to generate executable code based on the memory access and reduction space. The kernel implementation includes recursive construction of the parallel loop and code optimizer.

Recursive construction algorithm. The SpMM kernel is decomposed into multiple levels of parallelism, recursively from block to warp and thread as shown in Figure 13. At the highest level, the SpMM kernel initiates the block-level parallelism and performs block reduction algorithms (if needed). Then move down to the warp and thread level. Each thread operates on a division of the data. The `Thread Action` involves vector loading of elements from the sparse matrix and dense matrix, reusing sparse data of thread-groups. Finally, the thread reduction is performed using either the `THREAD_TOTAL/BITMAP` algorithm, depending on the sparse allocation.

Code Optimizer. An important consideration of code optimizer is the memory access of `offset`, which is used to determine the address range of non-zero elements in sparse matrix. However, `offset` may exhibit certain computational patterns in Table 3, so we can replace `offset` access with computational code. For instance, if the values follow a linear

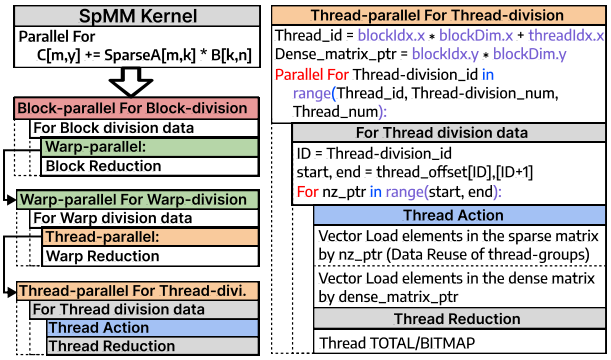


Figure 13: Kernel skeleton of code generator.

progression (fixed intervals n), instead of directly accessing the values, a mathematical formula ($n \times ID$) can be used to calculate the address. This approach reduces the memory access overhead and can enhance the overall efficiency of the SpMM implementation.

Table 3: Patterns of the offset values, where the memory access can be replaced with the computational code.

Method	Description	Offset Example
Linear	Compress indices with linear patterns into expressions	[0,5,10,...] → 5×ID ①
Branch	Compress indices with only k values into if statements	[0,5,8] → 0 if ID=0; 5 if ID=1; 8 if ID=2
Cyclicity	Periodically retain one complete cycle and the cycle size	[0,1,6,7,...] → 6×(ID/2)+ID%2
Quasilinear	Fit a linear function to transform data reads into linear calculations plus residuals	[3,8,13,...] → 5×ID +3

① start, end = thread_offset[ID],[ID+1] → start, end = 5×ID, 5×(ID+1) in Figure 13.

4 Evaluation

4.1 Experimental Setup

We evaluate the performance of GeneralSparse from two aspects: kernel-level and model-level evaluation. Mixed precisions (FP16/32) are supported. The platform is the NVIDIA Tesla A100, based on the Ampere architecture. The CUDA environment is version 12.1.

4.2 Kernel Performance

Dataset of matrices. The sparse matrix datasets are from two aspects. ***First***, the pruned weight matrix of LLM is across different shapes, corresponding to the matmuls within the OPT-30B and OPT-66B models [52]. ***Second***, to demonstrate that our method is applicable to various sparsity patterns, we evaluate 1168 matrices from SuiteSparse collection [11], which source from real-world problems in various scenarios, including scientific computing, graph processing, etc.

Baselines. We compare GeneralSparse with various methods targeting SpMM optimization:

(1) Library: (i) CuSPARSE (v12.1) [30], which inputs sparse matrices in CSR format and offers three parameter choices, with the best result chosen.

(2) Artificial designs and auto-tuners: (i) ASpT [23], which uses column sorting and adaptive tiling to partition sparse matrices. (ii) Sputnik [16], a well-designed sparse library for matrices in DL. (iii) DgSPARSE [9], as the auto-tuner, offers eight algorithm choices, with the best result chosen. (iv) Flash-LLM [45], which uses tensor core to accelerate sparse weight matrices in DL. (v) SparTA [53], which supports structured sparsity on tensor core. (vi) TC-GNN [44], which fits the sparse GNN workload on dense TCs. (vii) DTC-SpMM [14], which is tailored to harness TCs.

(3) Sparse compiler technologies: (i) TACO [26], which supports the code generation of SpMM with the default schedule. (ii) SparseTIR [48], which provides a sparse tensor compilation abstraction.

Furthermore, SpMM in LLMs is the half-precision computation and only partial methods support half-precision. So on pruned matrices of LLMs, SpMM is compared on these partial methods. For the remaining methods, the comparison is conducted on SuiteSparse matrix using single-precision.

Achieved performance on pruned weight matrix. To verify the performance superiority of our method under different sparsity levels and pruning methods, we employ two representative pruning methods (Magnitude and Random) and two sparsity levels (70/90%). Figure 14 presents the kernel performance (in TFLOPs) of GeneralSparse compared to the state-of-the-art methods. GeneralSparse consistently outperforms other methods.

(1) Notably, GeneralSparse achieves the average speedup of 17.15/19.14/20.82× at 8/32/64 batch size over cuSPARSE.

(2) GeneralSparse achieves performance improvements of average 1.84/1.57/1.30× at 8 batch size on A100 over Sputnik, sparTA and Flash-LLM (2.24/1.37/1.31× at 32 and 3.37/1.27/1.38× at 64 batch size). Other methods do not support half-precision computation on GPU.

(3) GeneralSparse achieves the average speedup of 1.21/1.34 /1.33× at 8/32/64 batch size over SparseTIR. TACO does not support half-precision on GPU.

Speedup on pruned methods and sparsity levels. Table 4 shows GeneralSparse outperforms other methods in different pruned methods and sparsity levels. The results also show that sparTA outperforms sputnik in random pruning (70%), while sputnik outperforms sparTA in magnitude pruning (90%). This is because random pruning leads to the low variance of the non-zero elements number per row and 70% sparsity level is relatively dense, which is more suitable for SparTA based on tensor-core operations. While GeneralSparse consistently outperforms other methods, because other methods are unable to adaptively adjust to different sparse pruning patterns.

Speedup on unstructured and structured pruned weight matrix. As pruning methods are usually divided into two

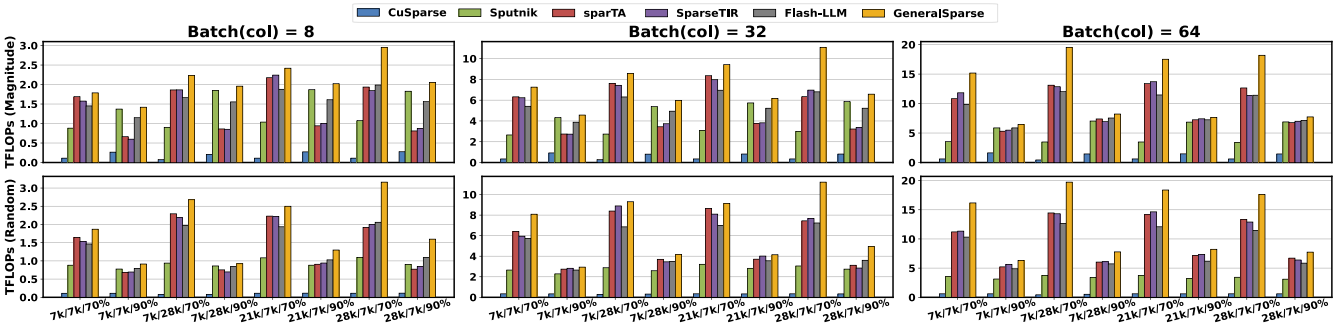


Figure 14: Kernel performance on weight matrices of different shapes using Magnitude and Random pruning.

Table 4: The average speedup of GeneralSparse over other methods on the weight matrices in Figure 14.

Methods	Pruned methods	col=8		col=32	
		70%	90%	70%	90%
CuSPARSE	Magnitude	23.71×	7.40×	27.99×	7.01×
	Random	26.13×	11.35×	29.29×	12.31×
Sputnik	Magnitude	2.39×	1.07×	3.15×	1.09×
	Random	2.54×	1.37×	3.20×	1.54×
SparTA	Magnitude	1.22×	2.27×	1.28×	1.77×
	Random	1.27×	1.51×	1.23×	1.22×
Flash-LLM	Magnitude	1.33×	1.26×	1.42×	1.21×
	Random	1.36×	1.24×	1.41×	1.21×

categories [5, 7]: structured pruning and unstructured pruning. We use different sizes of the pruning area to demonstrate the difference between the unstructured pruning method and the structured pruning method. The size of the pruned area in structured pruning is larger compared with unstructured pruning. Table 5 presents the kernel performance of GeneralSparse compared to the other methods.

The results indicate that GeneralSparse outperforms the cuSPARSE and Sputnik methods under various pruning granularities. However, it is inferior to SparTA, Flash-LLM, TC-GNN and DTC-SpMM in the case of a pruning granularity of (8, 8) and 70%. This is because when the pruning granularity is relatively large and relatively dense, it is more favorable for the structured computation of tensor-core. This indicates that our method is more suitable for unstructured pruning, but it also has a speedup for higher sparsity in structured pruning.

Achieved performance on SuiteSparse matrix. To validate that our method is applicable to a wider variety of sparse matrix characteristics, we conduct tests on the SuiteSparse matrix collection and use single-precision for floating-point values. Figure 15 illustrates the performance of 1168 matrices across various libraries and systems. The achieved performance is represented in a scatter plot, where the horizontal axis denotes the number of non-zero elements in the matrices, and the vertical axis measures performance in TFLOPS. The results demonstrate that GeneralSparse consistently outperforms other methods across the majority of the matrices analyzed. We can observe that:

(1) Notably, GeneralSparse achieves the average speedup of 6.39/4.38/7.46× over cuSPARSE at 8/32/64 batch size.

Table 5: The average speedup of GeneralSparse over other methods using random pruning on weight matrix.

Methods	Pruned size	col=8		col=32	
		70%	90%	70%	90%
CuSPARSE	(1,1)	26.13×	11.35×	29.29×	12.31×
	(4,4)	27.08×	13.33×	30.35×	12.37×
	(8,8)	27.10×	14.01×	31.37×	12.45×
Sputnik	(1,1)	2.54×	1.37×	3.20×	1.54×
	(4,4)	2.60×	1.45×	3.27×	1.56×
	(8,8)	2.61×	1.47×	3.28×	1.57×
SparTA	(1,1)	1.27×	1.51×	1.23×	1.22×
	(4,4)	1.12×	1.26×	1.11×	1.18×
	(8,8)	0.97×	1.02×	0.89×	0.93×
Flash-LLM	(1,1)	1.36×	1.24×	1.41×	1.21×
	(4,4)	1.22×	1.20×	1.31×	1.12×
	(8,8)	1.05×	1.15×	0.98×	1.04×
TC-GNN	(1,1)	1.41×	1.32×	1.46×	1.22×
	(4,4)	1.32×	1.25×	1.35×	1.16×
	(8,8)	1.07×	1.19×	0.98×	1.01×
DTC-SpMM	(1,1)	1.23×	1.20×	1.21×	1.32×
	(4,4)	1.11×	1.18×	1.09×	1.10×
	(8,8)	0.95×	1.02×	0.88×	0.95×

(2) GeneralSparse achieves performance improvements of average 2.32/1.37/N.A.× on A100 over Sputnik, DgSPARSE, and ASpT at 8 (1.22/1.20/7.69× at 32 and 1.29/1.23/2.15× at 64) batch size.

(3) GeneralSparse achieves the average speedup of 10.60/4.97/2.73× on A100 over TACO at 8/32/64 batch size. A primary factor is that TACO's design is not tailored for GPUs.

Speedup on SuiteSparse matrix. To illustrate the comparison between methods, we present the speedup ratio for the matrices in Table 6. The results demonstrate that there is a speedup effect compared to the other three methods across different columns for most matrices and experiences slowdown only for a small fraction of the matrices. This is because the current optimization method of the cost model we are using fails to cover finer-grained strategy selections. The best speedup effect is achieved where col = 8. This is because when col is relatively large, the proportion of computation increases compared with memory access, so the impact of memory access and reduction adjustments on GPU decreases.

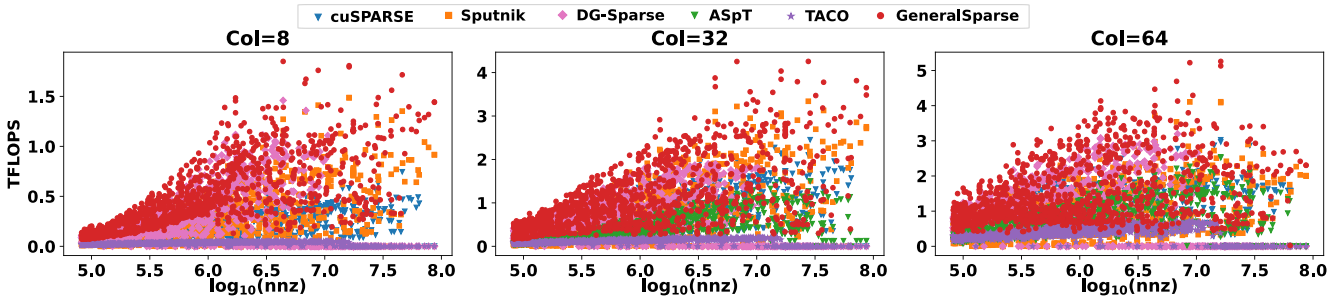


Figure 15: Kernel Performance on SuiteSparse matrix.

Table 6: The speedup of GeneralSparse over other methods. The percentage represents the portion of matrices from suitesparse matrices. (N.A., ASpT is not applicable to col=8).

Col	Speedup	cuSPARSE	Sputnik	DgSparse	ASpT
8	>1.8×	76.9%	54.3%	19.1%	N.A.
	1.4-1.8×	10.6%	27.4%	16.4%	
	1.0-1.4×	12.0%	15.4%	61.3%	
	0.8-1.0×	0.5%	2.9%	3.2%	
	mean	**6.39**×	**2.32**×	**1.37**×	N.A.
32	>1.8×	4.2%	36.7%	15.2%	64.8%
	1.4-1.8×	22.5%	17.9%	22.5%	16.9%
	1.0-1.4×	65.4%	40.0%	49.5%	17.7%
	0.8-1.0×	7.9%	5.4%	6.8%	0.6%
	mean	**4.38**×	**1.22**×	**1.20**×	**7.69**×
64	>1.8×	0.0%	39.7%	10.5%	23.9%
	1.4-1.8×	12.0%	27.2%	13.2%	40.7%
	1.0-1.4×	82.9%	30.2%	72.1%	35.0%
	0.8-1.0×	5.1%	2.8%	4.3%	0.3%
	mean	**7.46**×	**1.29**×	**1.23**×	**2.15**×

4.3 Kernel Analysis

The kernel analysis primarily focuses on comparing GeneralSparse with two representative methods, Sputnik [16] and SparTA [53], to investigate the root causes of performance improvement from various perspectives. Additionally, ablation experiments are performed to evaluate our proposed method.

GeneralSparse achieves speedup compared to Sputnik by improving SM utilization. The profiling metrics for the matrix are presented in Figure 16. Notably, *Active warps per SM* of each matrix increases significantly. Sputnik uses a fixed GPU allocation to assign the row of sparse matrix to different thread blocks, leading to load unbalance. SparTA's SM utilization is not listed because it uses tensor cores, making the comparison meaningless. GeneralSparse uses *offset* abstraction to achieve flexible GPU allocation of memory access and improve the parallelism. In addition, reduction methods, such as warp-level reduction, also improve the SM utilization.

GeneralSparse achieves speedup compared to SparTA by improving memory access efficiency. The profiling metrics for the matrix are presented in Figure 17. Notably, the *Memory Bandwidth* and *L1/TEX Cache Throughput* of each matrix increases significantly. The distinction in memory utilization is reflected in the reduction methods. SparTA converts the matrix into a format that is suitable for processing by

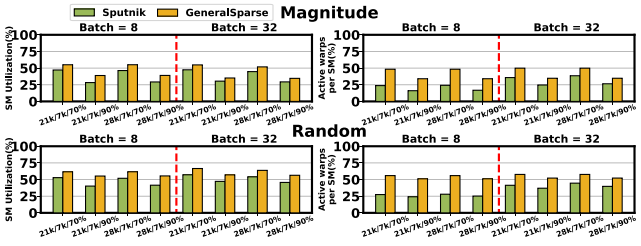


Figure 16: Profiling metrics compared with Sputnik.

the tensor-core and further performs reduction sequentially. GeneralSparse performs multi-level reduction on the GPU in the reduction method, alleviating the pressure of memory write-back to global memory. It saves the number of redundant write-back operations, improves L1 cache utilization, and thus increases memory bandwidth. Although the computing power of the tensor-core is stronger than that of the cuda-core, the performance bottleneck is limited by the memory access rather than the computing power.

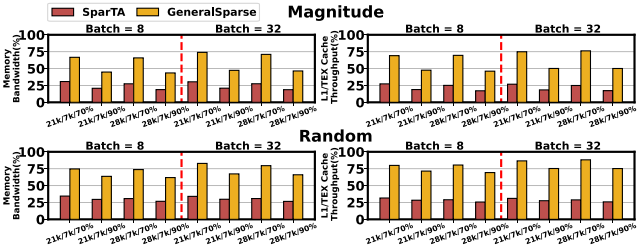


Figure 17: Profiling metrics compared with SparTA.

Performance gain breakdown. We breakdown the performance gain in Figure 18 brought by GeneralSparse on suitesparse matrix. Due to the diverse sparse characteristics of matrices, the key points for improving performance vary on different matrices. *GeneralSparse (base)* means using the fixed GPU allocation (one thread block per row of sparse matrix) and thread sequential reduction. *Op1* represents the flexible GPU allocation with format adjustments. *Op2* represents using the multi-level reduction methods. *Op1* mainly improves the performance on *magnitude* (*90%*) matrices, because they have a large variance in non-zero elements per row. *Op2* mainly improves the performance on *70%* matrices, because they have a relatively large number of non-zero elements per row. *GeneralSparse (base + op1 + op2)* achieves the best performance among them.

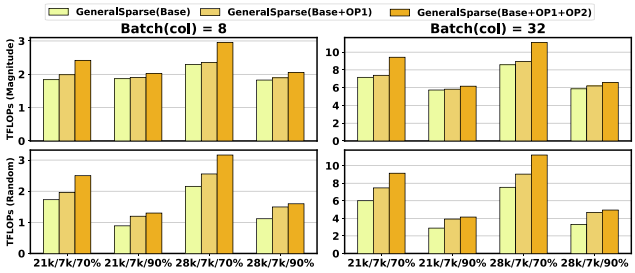


Figure 18: Performance gain breakdown of GeneralSparse.

Achieved performance on additional GPU hardware. We also test the performance improvements of the pruned weight matrices at 8 batch size on NVIDIA V100 in Figure 19.

(1) Notably, GeneralSparse achieves the average speedup of 16.08× at 8 batch size over cuSPARSE.

(2) GeneralSparse achieves performance improvements of average 1.73/1.55/1.28× at 8 batch size on V100 over Sputnik, sparTA and Flash-LLM.

(3) GeneralSparse achieves the average speedup of 15.98× at 8 batch size over SparseTIR.

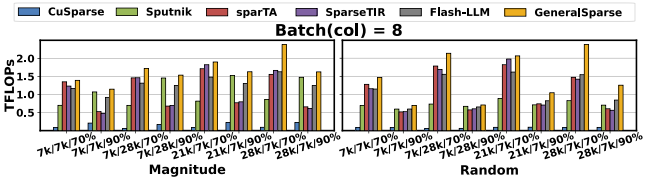


Figure 19: Kernel performance on weight matrices using Magnitude and Random on NVIDIA V100.

4.4 Model Evaluation

Settings and Baselines. We integrate the kernels, generated by GeneralSparse, into FasterTransformer [31] by providing a set of C++ APIs, which enables high-efficiency distributed inference with sparsified weight matrices. GeneralSparse can also be easily integrated into other deep learning frameworks through library calls with APIs. We compare GeneralSparse with various methods by end-to-end performance:

(1) Dense computation: (i) CUBLAS [33], which is included as a baseline to compare the performance gains to standard dense implementations of LLM inference.

(2) Sparse counterparts: (i) CuSPARSE, which is the sparse library. (ii) Flash-LLM, which is also integrated into FasterTransformer.

Dataset of models. We benchmark the end-to-end inference latency on OPT-30B/66B [52] and Llama-7B/13B/65B [41], Llama-3.1-8B/70B [19]. For all experiments, the input prompt sequence length is 64, and the output/generated sequence length is 512. We utilize the magnitude pruning [29] method, and use 70% sparsity level at the bottom and top layers, and 90% sparsity level at the intermediate layers.

To demonstrate that the pruned model maintains comparable performance to the original model, we evaluate the accuracy of the pruned versions of OPT-30B/66B [52] on the BoolQ task of SuperGLUE [42] by lm-evaluation-harness benchmark [17]. Specifically, the accuracy for OPT-30B decreases from 69.69% to 67.20%, while OPT-60B's accuracy declines from 70.46% to 68.01%.

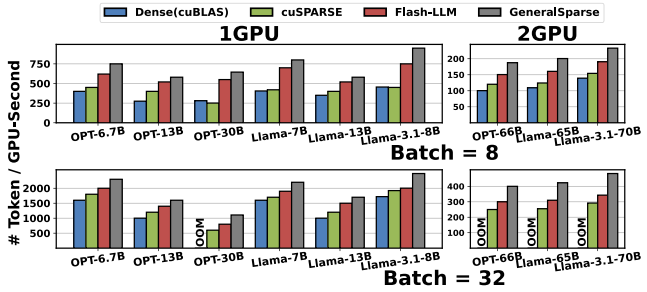


Figure 20: End-to-end model throughput.

Metric. We use the metric of `tokens per GPU-second` to represent normalized inference throughput, accounting for both inference time and hardware cost (i.e., the number of GPUs used), which is calculated by the formula: $N_{token}/\sum_{i=1}^{N_{gpu}} T_i$. N_{token} means the number of tokens generated, whereas N_{gpu} and T_i mean the GPU number and the time spent on the i'th GPU for execution. (OOM, out-of-memory).

Speedups on models. OPT-60B, Llama-65B and Llama-3.1-70B models are evaluated on two A100 GPUs in a tensor parallel manner. Other smaller models are evaluated on a single GPU. From the results, we observe that:

(1) Notably, GeneralSparse achieves the average speedup of 2.33/1.58× over Dense(cuBLAS) at 8/32 batch size.

(2) GeneralSparse achieves performance improvements of average 1.50/1.20× at 8 batch size, 1.48/1.23× at 32 batch size over cuSPARSE/FlashLLM.

OPT-30B, OPT-66B and Llama-65B models exceed the GPU memory limit (OOM, out-of-memory) when batch size is 32. However, the weights after sparse pruning reduce the memory usage, enabling the normal inference process. Besides, under the OPT-30B model (batch size = 8), the inference time of cuSPARSE is even worse than that of the dense implementation, because in some sparse patterns the performance of cuSPARSE is worse than its dense implementation.

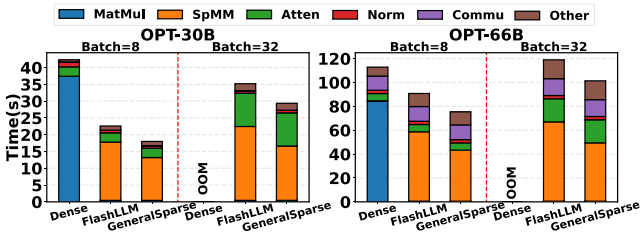


Figure 21: Time breakdown compared with other methods.

4.5 Model Analysis

OPT-30B. For figuring out why GeneralSparse can achieve better performance, we conduct the end-to-end breakdowns using the NSight System [32]. The inference time is shown in Figure 21. GeneralSparse can achieve lower inference latency mainly because we replace the dense MatMul with the more efficient SpMM. Besides, the time of MatMul, SpMM, and attention all increase with the increase in batch size. Compared with cuSPARSE and Flash-LLM, however, due to the performance improvement of SpMM, we decrease the inference time of the end-to-end model.

OPT-66B. To verify the proportion of our matrix time on multiple GPUs, we further analyze the inference time of OPT-60B across 2-GPUs. The OPT-60B runs on two GPUs in a tensor parallel manner, where the weight matrices are split and distributed to 2 GPUs. We conduct the time breakdown of end-to-end inference for OPT-66B as shown in Figure 21. Compared with a single GPU, there is additional communication overhead, but the performance bottleneck is still the MatMul part. It also verifies that our kernel speedup and tensor parallel can be effectively integrated. In detail, the acceleratio ratio in the prefill and decoding stages is 1.21/1.45 times and 1.52/1.63 times higher than FlashLLM/Dense at 8 batch size.

4.6 Search Analysis

Offline search analysis. The number of search iterations depends on the number of valid strategy combinations and the matrix characteristics. The search ranges about 30-50 iterations and 10-30 seconds in our search, achieving 90% of the best performance (hundreds search and about 15 minutes). For performance impact of different strategies, the difference in performance between different strategies for different matrices varies, with some matrices having a performance gap of three times that of different strategies, while others fluctuate by about ten percent.

Code generation and compilation time cost. For code generation time cost, it takes about a few seconds on small matrices, and one or two minutes on large matrices. During the offline search, the generated code has already been compiled for each matrix. At runtime, one can directly call the customized library for each matrix, similar to calling a sparse matrix multiplication library. We measure the compilation time for each matrix, which is only a few seconds. For the entire model compilation time, it takes only a few minutes for smaller models like Llama-7B, and for larger models such as Llama-65B, it takes around ten-odd minutes.

5 Related Work and Discussion

Our work draws inspiration from and contributes to several key research directions:

SpMM optimization. SpMM is a fundamental operation in sparse computation, with prior works such as cuSPARSE [34] and TACO [26] providing essential support for sparse matrix operations. However, these approaches often lack flexibility and fail to fully leverage GPU architectures for diverse sparsity patterns. Recent methods, such as Sputnik [16] and ASpT [23], have focused on optimizing SpMM for specific sparsity configurations, while SparTA [53] and Flash-LLM [45] utilize GPU tensor cores for structured sparsity acceleration. Despite their improvements, these methods often remain limited in adaptability to diverse sparsity patterns. GeneralSparse addresses these limitations by introducing an adaptable solution and automated code generation.

Pruning techniques for neural networks. Pruning methods are usually classified into unstructured pruning [20, 40] to structured sparsity [5, 7], which have been widely adopted to reduce the computational and memory demands of neural networks. In practice, unstructured pruning achieves higher accuracy retention [15, 18, 21] compared with structured pruning. GeneralSparse is designed to handle both categories, providing consistent performance gains across varying sparsity levels and patterns.

Large language model acceleration. Accelerating LLMs has become a critical research focus due to their growing computational demands. Existing efforts, such as FasterTransformer [31], DeepSpeed [36], and FlashAttention [10], have optimized various components of LLM inference, including attention mechanisms and scheduling strategies. By integrating seamlessly with existing LLM acceleration frameworks, GeneralSparse enhances the overall efficiency of sparsified LLM inference.

Discussion. This work represents a convergence of advancements in sparse computation, pruning methods, and GPU optimization, delivering a **vertical solution** that aligns with ongoing developments in LLM acceleration and sparse computation. By bridging the gap between adaptability and performance, GeneralSparse sets a new benchmark for high-performance sparse computation on GPUs.

6 Conclusion

We present GeneralSparse, a novel solution for designing SpMM programs on GPUs, addressing key challenges in performance bottlenecks in LLM inference considering diverse pruning patterns and sparsity levels. By introducing a unified abstraction of memory access and reduction spaces, flexible GPU allocation, and automated kernel generation. Extensive evaluations demonstrate its adaptability across pruned weight matrices and real-world datasets. Our code is available on https://github.com/Wangyaoyuu/GeneralSparse.

Acknowledgments

The authors sincerely thank our shepherd Somali Chaterji and anonymous reviewers for their insightful suggestions. The work is supported by National Natural Science Foundation of China, under Grant No. 62032023, T2125013, 62172391.

References

[1] Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav Gulavani, Alexey Tumanov, and Ramachandran Ramjee. Taming {Throughput-Latency} tradeoff in {LLM} inference with {Sarathi-Serve}. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 117–134, 2024.

[2] Guangji Bai, Yijiang Li, Chen Ling, Kibaek Kim, and Liang Zhao. Sparsellm: Towards global pruning of pre-trained language models. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024.

[3] Davis Blalock, Jose Javier Gonzalez Ortiz, Jonathan Frankle, and John Guttag. What is the state of neural network pruning? In I. Dhillon, D. Papailiopoulos, and V. Sze, editors, *Proceedings of Machine Learning and Systems*, volume 2, pages 129–146, 2020.

[4] Tom B Brown. Language models are few-shot learners. In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901. Curran Associates, Inc., 2020.

[5] Roberto L Castro, Andrei Ivanov, Diego Andrade, Tal Ben-Nun, Basilio B Fraguela, and Torsten Hoefler. Venom: A vectorized n: M format for unleashing the power of sparse tensor cores. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14, 2023.

[6] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. {TVM}: An automated {End-to-End} optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 578–594, 2018.

[7] Zhaodong Chen, Zheng Qu, Liu Liu, Yufei Ding, and Yuan Xie. Efficient tensor core-based gpu kernels for structured sparsity under reduced precision. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14, 2021.

[8] Stephen Chou, Fredrik Kjolstad, and Saman Amarasinghe. Format abstraction for sparse tensor algebra compilers. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):1–30, 2018.

[9] Guohao Dai, Guyue Huang, Shang Yang, Zhongming Yu, Hengrui Zhang, Yufei Ding, Yuan Xie, Huazhong Yang, and Yu Wang. Heuristic adaptability to input dynamics for spmm on gpus. In *Proceedings of the 59th ACM/IEEE Design Automation Conference*, pages 595–600, 2022.

[10] Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in Neural Information Processing Systems*, 35:16344–16359, 2022.

[11] Timothy A Davis and Yifan Hu. The university of florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)*, 38(1):1–25, 2011.

[12] Tianyu Ding, Tianyi Chen, Haidong Zhu, Jiachen Jiang, Yiqi Zhong, Jinxin Zhou, Guangzhi Wang, Zhihui Zhu, Ilya Zharkov, and Luming Liang. The efficiency spectrum of large language models: An algorithmic survey. *arXiv preprint arXiv:2312.00678*, 2023.

[13] Zhen Du, Jiajia Li, Yinshan Wang, Xueqi Li, Guangming Tan, and Ninghui Sun. Alphasparse: Generating high performance spmv codes directly from sparse matrices. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15. IEEE, 2022.

[14] Ruibo Fan, Wei Wang, and Xiaowen Chu. Dtc-spmm: Bridging the gap in accelerating general sparse matrix multiplication with tensor cores. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pages 253–267, 2024.

[15] Elias Frantar and Dan Alistarh. Sparsegpt: Massive language models can be accurately pruned in one-shot. In *International Conference on Machine Learning*, pages 10323–10337. PMLR, 2023.

[16] Trevor Gale, Matei Zaharia, Cliff Young, and Erich Elsen. Sparse gpu kernels for deep learning. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14. IEEE, 2020.

[17] Leo Gao, Jonathan Tow, Baber Abbasi, Stella Biderman, Sid Black, Anthony DiPofi, Charles Foster, Laurence Golding, Jeffrey Hsu, Alain Le Noac'h, Haonan Li, Kyle McDonell, Niklas Muennighoff, Chris Ociepa, Jason Phang, Laria Reynolds, Hailey Schoelkopf, Aviya Skowron, Lintang Sutawika, Eric Tang, Anish Thite, Ben Wang, Kevin Wang, and Andy Zou. A framework for few-shot language model evaluation, 07 2024.

[18] Aidan N Gomez, Ivan Zhang, Siddhartha Rao Kamalakara, Divyam Madaan, Kevin Swersky, Yarin Gal, and Geoffrey E Hinton. Learning sparse networks using targeted dropout. *arXiv preprint arXiv:1905.13678*, 2019.

[19] Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

[20] Song Han, Jeff Pool, John Tran, and William J. Dally. Learning both weights and connections for efficient neural networks, 2015.

[21] Torsten Hoefler, Dan Alistarh, Tal Ben-Nun, Nikoli Dryden, and Alexandra Peste. Sparsity in deep learning: Pruning and growth for efficient inference and training in neural networks. *Journal of Machine Learning Research*, 22(241):1–124, 2021.

[22] Changwan Hong, Aravind Sukumaran-Rajam, Bortik Bandyopadhyay, Jinsung Kim, Süreyya Emre Kurt, Israt Nisa, Shivani Sabhlok, Ümit V Çatalyürek, Srinivasan Parthasarathy, and P Sadayappan. Efficient sparse-matrix multi-vector product on gpus. In *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*, pages 66–79, 2018.

[23] Changwan Hong, Aravind Sukumaran-Rajam, Israt Nisa, Kunal Singh, and P Sadayappan. Adaptive sparse tiling for sparse matrix multiplication. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, pages 300–314, 2019.

[24] Guyue Huang, Guohao Dai, Yu Wang, and Huazhong Yang. Ge-spmm: General-purpose sparse matrix-matrix multiplication on gpus for graph neural networks. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12. IEEE, 2020.

[25] Peng Jiang, Changwan Hong, and Gagan Agrawal. A novel data transformation and execution strategy for accelerating sparse matrix multiplication on gpus. In *Proceedings of the 25th ACM SIGPLAN symposium on principles and practice of parallel programming*, pages 376–388, 2020.

[26] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. The tensor algebra compiler. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):1–29, 2017.

[27] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*, 2023.

[28] Junqing Lin, Honghe Zhang, Xiaolong Shi, Jingwei Sun, Xianzhi Yu, Jun Yao, and Guangzhong Sun. Ec-spmm: Efficient compilation of spmm kernel on gpus. In *Proceedings of the 52nd International Conference on Parallel Processing*, ICPP '23, page 21–30, New York, NY, USA, 2023. Association for Computing Machinery.

[29] Pavlo Molchanov, Arun Mallya, Stephen Tyree, Iuri Frosio, and Jan Kautz. Importance estimation for neural network pruning. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 11264–11272, 2019.

[30] Maxim Naumov, L Chien, Philippe Vandermersch, and Ujval Kapasi. Cusparse library. In *GPU Technology Conference*, 2010.

[31] NVIDIA. Fastertransformer. https://github.com/NVIDIA/FasterTransformer, 2022.

[32] NVIDIA. Nsight system. https://developer.nvidia.com/nsight-systems, 2023.

[33] NVIDIA. Cublas docs. https://docs.nvidia.com/cuda/cublas, 2024.

[34] NVIDIA. Cusparse docs. https://docs.nvidia.com/cuda/cusparse, 2024.

[35] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

[36] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 3505–3506, 2020.

[37] Abudurexiti Reheman, Tao Zhou, Yingfeng Luo, Di Yang, Tong Xiao, and Jingbo Zhu. Prompting neural machine translation with translation memories. In *Proceedings of the AAAI Conference on Artificial Intelligence*, pages 13519–13527, 2023.

[38] Hang Shao, Bei Liu, and Yanmin Qian. One-shot sensitivity-aware mixed sparsity pruning for large language models. In *ICASSP 2024-2024 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 11296–11300. IEEE, 2024.

[39] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

[40] Mingjie Sun, Zhuang Liu, Anna Bair, and J Zico Kolter. A simple and effective pruning approach for large language models. *arXiv preprint arXiv:2306.11695*, 2023.

[41] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.

[42] Alex Wang, Yada Pruksachatkun, Nikita Nangia, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel Bowman. Superglue: A stickier benchmark for general-purpose language understanding systems. *Advances in neural information processing systems*, 32, 2019.

[43] Wenxiao Wang, Wei Chen, Yicong Luo, Yongliu Long, Zhengkai Lin, Liye Zhang, Binbin Lin, Deng Cai, and Xiaofei He. Model compression and efficient inference for large language models: A survey. *arXiv preprint arXiv:2402.09748*, 2024.

[44] Yuke Wang, Boyuan Feng, Zheng Wang, Guyue Huang, and Yufei Ding. {TC-GNN}: Bridging sparse {GNN} computation and dense tensor cores on {GPUs}. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 149–164, 2023.

[45] Haojun Xia, Zhen Zheng, Yuchao Li, Donglin Zhuang, Zhongzhu Zhou, Xiafei Qiu, Yong Li, Wei Lin, and Shuaiwen Leon Song. Flash-llm: Enabling cost-effective and highly-efficient large generative model inference with unstructured sparsity. *Proceedings of the VLDB Endowment*, 17(2):211–224, 2023.

[46] Jie Xin, Xianqi Ye, Long Zheng, Qinggang Wang, Yu Huang, Pengcheng Yao, Linchen Yu, Xiaofei Liao, and Hai Jin. Fast sparse deep neural network inference with flexible spmm optimization space exploration. In *2021 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7. IEEE, 2021.

[47] Carl Yang, Aydın Buluç, and John D Owens. Design principles for sparse matrix multiplication on the gpu. In *European Conference on Parallel Processing*, pages 672–687. Springer, 2018.

[48] Zihao Ye, Ruihang Lai, Junru Shao, Tianqi Chen, and Luis Ceze. Sparsetir: Composable abstractions for sparse compilation in deep learning. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pages 660–678, 2023.

[49] Lu Yin, You Wu, Zhenyu Zhang, Cheng-Yu Hsieh, Yaqing Wang, Yiling Jia, Gen Li, Ajay Jaiswal, Mykola Pechenizkiy, Yi Liang, et al. Outlier weighed layerwise sparsity (owl): A missing secret sauce for pruning llms to high sparsity. *arXiv preprint arXiv:2310.05175*, 2023.

[50] A. M. Zaki, M. I. Khalil, and H. M. Abbas. Deep architectures for abstractive text summarization in multiple languages. In *2019 14th International Conference on Computer Engineering and Systems (ICCES)*, pages 22–27, 2019.

[51] Amr M. Zaki, Mahmoud I. Khalil, and Hazem M. Abbas. Amharic abstractive text summarization, 2020.

[52] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, et al. Opt: Open pre-trained transformer language models. *arXiv preprint arXiv:2205.01068*, 2022.

[53] Ningxin Zheng, Bin Lin, Quanlu Zhang, Lingxiao Ma, Yuqing Yang, Fan Yang, Yang Wang, Mao Yang, and Lidong Zhou. {SparTA}:{Deep-Learning} model sparsity via {Tensor-with-Sparsity-Attribute}. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 213–232, 2022.

HyCache: Hybrid Caching for Accelerating DNN Input Preprocessing Pipelines

Keshav Vinayak Jha *
keshavvinayakjha@gmail.com
Independent Researcher
Bengaluru, India

Shweta Pandey
shwetapandey@iisc.ac.in
Indian Institute of Science
Bengaluru, India

Murali Annavaram
annavara@usc.edu
University of Southern California
Los Angeles, CA, USA

Arkaprava Basu
arkapravab@iisc.ac.in
Indian Institute of Science
Bengaluru, India

Abstract

End-to-end deep neural networks' (DNNs) training performance depends not only on the time spent in training the model weights but also on the time spent in loading and preprocessing the training data. Recent advances in GPU hardware have made training substantially faster. As a result, the bottleneck has shifted to the CPU-based input pipeline. This pipeline must fetch and transform each sample through multiple stages before it can be consumed by the GPU.

Prior works accelerate preprocessing by caching intermediate results across epochs, but suffer from several key limitations: (1) They cache either in memory or in storage, but are unable to leverage both together. (2) They can cache the output of a stage only if it can entirely fit in the cache, which is a severe limitation for larger datasets. (3) They can cache the output of *only* one of the stages which could be suboptimal.

We thus introduce Hybrid Cache (HyCache), a runtime that enables the caching of subsets of preprocessed data from multiple intermediate steps on *both* memory and storage. HyCache possesses the ability to partially cache the outputs of a stage across both memory and storage. HyCache deploys integer linear programming (ILP) to automatically determine the best caching strategies across the memory and the storage by finding an optimal trade-off between recomputation and caching. Importantly, it does so *without* any manual intervention. HyCache outperforms state-of-the-art prior approaches, delivering a raw pipeline throughput improvement ranging in speedups from 1.11× to 10.1× over a variety of pipelines.

1 Introduction

Deep Neural Networks (DNNs) have emerged as a transformative force in the field of machine learning, revolutionising how we process, understand, and make decisions from complex data. Their importance lies in their ability to learn intricate patterns, hierarchies, and representations automatically from data. Their success has driven advances across diverse domains such as computer vision [17, 25, 41, 48], natural language processing [15, 49], healthcare [3], and autonomous systems [22].

DNN training typically proceeds over multiple epochs, each representing a complete pass over the training dataset. To achieve high accuracy and generalisation, the model often requires multiple such passes over large-scale, high-quality datasets [11, 50]. However, these raw datasets—e.g., JPEG images or text files—are not directly consumable by the model and must first be transformed into structured tensors.

The **input preprocessing pipeline** handles the transformation of the raw dataset into structured tensors, which are finally consumed by the training pipeline. The pipeline performs a series of operations, such as image decoding, colour-space conversion, normalisation, data-type casting, and random augmentations, to prepare data batches for the accelerator [36]. These steps are essential for maintaining model accuracy and training stability. Their exact composition varies by model and task. As training progresses through epochs, each sample repeatedly passes through this pipeline, making it a critical component of end-to-end training performance. Generally, input preprocessing is managed by the CPU while the GPU performs training.

Today, accelerators like GPUs have dramatically improved the throughput of model training [40]. Meanwhile, dataset sizes continue to grow exponentially [20]. This places a significant pressure on the CPU-based preprocessing pipeline. Consequently, preprocessing is now often the critical path, stalling training and wasting valuable GPU cycles [23, 33].

Studies show that preprocessing delays can dominate training time: up to 65% of epoch time can be spent in the preprocessing pipeline [33]. Google reported that 62% of training pipelines stall for at least 1 ms *per epoch* due to input pipeline delays, and 16% stall for more than 100 ms [26] per epoch. At any given time, up to 10% of jobs wait on preprocessing and end-to-end training can spend as much as 65% of epoch time on data preprocessing alone [33].

Alleviating input preprocessing pipeline bottlenecks: A widely used strategy to mitigate preprocessing pipeline stalls is to cache the output of expensive preprocessing stages so that

*Work done as a student at the Indian Institute of Science.

they can be reused across epochs. Existing frameworks like *tf.data* [34] offer built-in support for in-memory and on-disk caching. However, these mechanisms rely on users to manually specify which pipeline stages to cache.

Recent systems have proposed automated and profile-guided caching strategies [14, 20, 26, 30]. These systems address a key limitation of OS-level caches—such as LRU—which perform poorly under the random access patterns caused by input shuffling across epochs [33]. Instead, they use offline profiling to identify preprocessing steps that are both compute-intensive and have a small data size footprint, making them ideal candidates for caching. For example, PRESTO [20] and Cachew [14] selectively cache such steps to reduce preprocessing latency.

However, these techniques face an important tradeoff: intermediate outputs, such as decoded RGB images, can be significantly larger than the raw dataset. As a result, even if a step is computationally expensive, caching its output may be infeasible due to memory or storage budget. Thus, any effective caching strategy must balance the cost of recomputation against the space required to store intermediate results.

Limitations of prior caching approaches: While prior works demonstrate the value of caching in DNN input pipelines, they remain limited in flexibility and adaptability. Most existing techniques suffer from four key shortcomings:

(1) They adopt an all-or-nothing approach—a pipeline stage is cached only if its entire output fits within the available memory or storage. If not, caching is disabled entirely, even though partial caching could still reduce recomputation costs.

(2) They treat memory and storage as disjoint caching tiers, using either memory or storage for caching but never coordinating the two. This separation prevents optimal combined use of system resources.

(3) If both memory and storage are used for caching, prior work restricts caching to the same pipeline step in both memory and storage. This limits the flexibility to exploit the differing performance characteristics of each tier.

(4) These works lack any mechanism to estimate or reserve working memory, which is needed for tasks like batching, data queuing, or threading. As a result, they risk overcommitting memory when allocating space for the cache, potentially causing stalls or out-of-memory errors.

These design simplifications ease implementation, but they miss significant opportunities for improving cache efficiency, particularly in modern training environments that provide sufficient DRAM and storage bandwidth.

Hybrid Caching (HyCache): We propose HyCache, which is designed to overcome the limitations of existing caching approaches by enabling partial, exclusive and coordinated caching across both memory and storage. It enables:

(1) HyCache supports **partial caching** of intermediate outputs. Rather than requiring an entire stage's output to fit in memory or storage, it allows arbitrary subsets of tensors to be cached. This enables reuse even under tight resource budgets.

(2) HyCache ensures **exclusive caching** across memory and storage. The intermediate outputs cached in both do not overlap. This avoids duplication, maximises overall capacity utilisation, and simplifies cache management.

(3) HyCache uses a **coordinated caching policy** to decide not only which tensors to cache, but also from which pipeline step and whether in memory or storage. It may place outputs of different preprocessing steps in memory and storage depending on their compute intensity and size.

(4) HyCache automatically estimates the **working memory requirements** of the input pipeline—accounting for batching, threading, and buffer management—and only allocates the remaining DRAM for caching. This prevents out-of-memory errors without burdening the user to tune cache sizes manually.

To realise its hybrid caching strategy, HyCache first profiles the input preprocessing pipeline to estimate the compute cost saved by caching in memory or storage and the output size of each step. It then formulates an integer linear program (ILP) to determine the optimal caching plan under given memory and storage budgets. The ILP balances the cost of recomputation against the latency and footprint of storing intermediate outputs, selecting both which steps to cache and where to cache them. The selected tensors are then placed into memory or storage according to this plan.

At runtime, HyCache supports seamless ingestion of intermediate tensors regardless of their origin—some may be fetched from DRAM, others from SSD, and the rest recomputed as needed. It guarantees correctness and ordering across these mixed sources, enabling fine-grained reuse without interrupting pipeline execution.

To avoid memory overcommitment, HyCache estimates the working memory required for pipeline operation—accounting for batching, threading, and buffer usage—and reserves only the remaining memory for caching. This prevents OOM errors and eliminates the need for manual tuning. Finally, HyCache exposes a simple, user-facing API that abstracts the cache management logic, enabling drop-in integration into existing training pipelines.

Summary: HyCache (1) Enables partial and coordinated caching across memory and storage, allowing different stages to be cached in different tiers. (2) Uses profiling and ILP to select what to cache and where, optimising for latency and resource constraints. (3) Supports ingestion of mixed cached and uncached tensors while preserving pipeline correctness. (4) Automatically adjusts cache sizes based on working memory needs and exposes simple APIs for easy integration.

2 Background

The input preprocessing pipeline can be characterised as a three-stage extract, transform, load (ETL) process. It applies transformations to the input data, such as permuting and filtering data to extract a subset of the relevant features and create input tensors to be used for training. Since the input data is usually stored on a CPU-attached storage device,

the transformation operations are performed on the CPU. Depending on the type of transformation, they may require significant computation. It may also generate outputs that exceed the input size. Finally, the third stage loads the data onto the GPU for training. Models are iteratively trained for several epochs for accuracy. Due to the large size of the dataset generated from transformations, it is not possible to keep all processed data in GPU memory for training. Thus, the preprocessing is repeated to generate tensors for each epoch.

The input pipeline is typically decoupled from the training on the GPU. Ideally, the input pipeline should continuously feed data to the GPU. Decoupling works effectively when the GPU computation time per batch equals or exceeds the latency to ETL the input data batch. With the GPU computational throughput increasing over time, the training time has decreased, making input pipelines a potential bottleneck. Prior works [26, 33] have shown that DNN training is now often I/O bound, bottlenecked by fetching the data from storage, or CPU-bound, bottlenecked by computation on the data items.

To provide further insights into the ETL process, consider the raw input for an image classification model. The raw input might be a protocol buffer containing a JPEG-encoded image, and the input pipeline must convert the raw input into a dense three-dimensional array of floating-point values corresponding to the RGB values of each pixel. During this process, the input pipeline needs to extract and decode the JPEG, then apply extra transformations like affine transformations and changes in colour space to augment the training data. Some of the input transformations lead to a significant increase in data size that exceeds available space even in large server-class CPU configurations. Further, given the slow access time to the storage devices, it is not always beneficial to cache the transformed tensors; recomputing them may be faster than accessing the cached tensors.

Preprocessing steps can be classified into two: ① **Offline (deterministic) steps**: always produce the same output for a given input, like JPEG decode, fixed resize, channel-wise normalisation. These steps can be computed once and cached across epochs. ② **Online (stochastic) steps**: which introduces randomness in each epoch, like random crops/flips, colour jitter. Hence, caching them either wastes space or defeats their randomness. We focus on caching offline steps, where outputs are repeatable and caching is feasible.

3 Opportunities for improving caching

In this section, we quantify the performance headroom for DNN input preprocessing pipelines and identify key opportunities for a partial, exclusive and coordinated caching strategy across memory and storage.

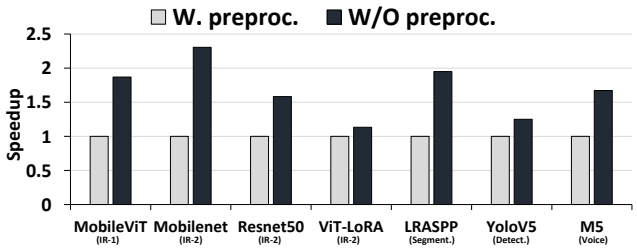


Figure 1: Headroom analysis for preprocessing pipelines.

3.1 Measuring headroom for improvement

We quantify the performance headroom available in end-to-end DNN training pipelines by eliminating preprocessing stalls. To do so, we measure two latencies: the baseline latency where input is fetched and preprocessed on demand before training, and the ideal latency with no preprocessing stalls.

To emulate the no-stall ideal scenario, we preload ~16K fully preprocessed tensors into CPU DRAM and measure the training time over these in-memory batches. This setup effectively removes all preprocessing stalls. [1] We then compare this against the baseline latency, where raw data is fetched and preprocessed for every batch. The difference between the two latencies quantifies the upper bound on speedup achievable by eliminating preprocessing stalls.

Figure 1 shows this headroom across various models (see Table 3 for model-pipeline details). Each model has two bars: one for the default case with preprocessing (W. preproc.') and one for the no-stall case (W/O preproc.'). The ratio between them reflects the speedup from removing preprocessing delays, ranging from 1.13× to 2.3× depending on the model.

3.2 Analyzing key limitations of prior works

All-or-nothing caching: Frameworks like TensorFlow offer APIs (e.g., `tf.data.Dataset.cache`) to cache the output of a specific pipeline step—provided the entire output fits in memory or storage. Users must manually select which step to cache, balancing the cost of recomputation against retrieval. Recent systems automate this decision by profiling the pipeline to identify a single "optimal" step for caching [14, 20].

While simple, this approach is sub-optimal. Many preprocessing steps, such as JPEG decoding, significantly inflate data volume [33], making their outputs too large to fully cache in. These steps are thus excluded from caching, even when partially caching their outputs could improve performance. An ideal caching framework should support *partial caching*—storing only a subset of intermediate tensors.

Caching either in memory or storage: Prior works focus on caching either in memory [26, 33] or in storage [20], but do not coordinate both. This separation can lead to inefficiencies.

[1] This emulation is valid because per-batch preprocessing and training latencies remain the same across samples, so the DRAM-resident subset accurately reflects a no-stall pipeline.

For example, data cached in storage may still be fetched into DRAM via the OS page cache, duplicating cached content across both. To fully utilise system resources, caching decisions should be coordinated across memory and storage, with exclusive contents in each.

Difference in memory and storage latency: Memory and storage differ significantly in both latency and capacity. A preprocessing step that is expensive to recompute may benefit from being cached in memory but not in storage, where access latency could outweigh recomputation. Prior works cache the same pipeline step in both tiers, ignoring the difference in latencies of memory and storage. A better approach would select different steps for memory and storage based on a comparison of recompute and access costs.

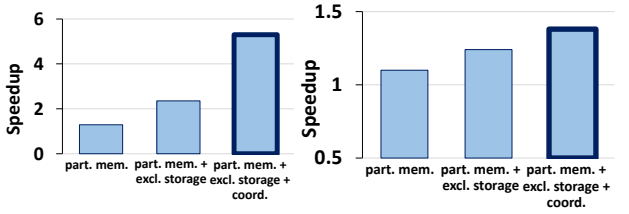
Variable working memory requirements: Different pipelines demand different amounts of working memory for batching, buffer management, and parallel fetching. In our study across six pipelines, we observed working memory usage ranging from 16–77 GB. This requirement also varies with dataset characteristics and the number of fetcher processes. Manually estimating this budget is error-prone; overprovisioning leads to poor cache utilisation, while underprovisioning risks out-of-memory failures. However, prior works require users to manually specify cache size, without accounting for dynamic working memory needs, limiting usability in real-world deployments.

Summary. An ideal caching framework should address these limitations by: ① Supporting partial caching of intermediate outputs instead of requiring all-or-nothing decisions; ② Coordinating memory and storage while avoiding redundant copies; ③ Allowing different preprocessing steps to be cached in memory and storage based on latency–cost trade-offs; ④ Automatically sizing the cache after reserving sufficient working memory for pipeline execution.

3.3 Quantifying opportunities for better caching strategies

We evaluated three different caching schemes on two representative DNN training pipelines (voice recognition and object detection) to quantify the potential benefits of a coordinated memory and storage caching strategy. We used the C2 configuration as described in Table 5. The first strategy allows partial caching of the output of a pipeline step only in memory (`'part. mem.'` in short). The optimal step to cache for a given available memory capacity is decided by an ILP solver (detailed Section 5), which considers the fraction of the output of the steps that can be cached in the available memory.

The second strategy additionally caches data in storage (here, SSD), denoted by `'part. mem + excl. storage'` in short. This policy ensures that the contents of the cache in memory and SSD are *exclusive*. However, the output of the *same* intermediate step is cached partly in memory until the available memory is fully utilised. The remaining tensors are cached in storage until storage capacity is exhausted. The remaining tensors are recomputed.



(a) **Voice recognition pipeline** (b) **Object detection pipeline**

Figure 2: Speedup of voice recognition and object detection pipeline across different caching policies (speedup is relative to the baseline).

Finally, the third configuration improves it further by allowing the caching of outputs of *different steps* in memory and storage (`part. mem. + excl. storage + coord.`). This coordinated strategy uses an ILP and access latency profiling to decide which step to cache in memory and storage.

Figure 2 shows the performances of these different caching policies for two representative DNN input pipelines: voice recognition and object detection. The performance of the caching policies is measured as the speedup over in-memory caching that disallows partial caching, as in prior work. We observe that enabling partial caching in memory improves performance by 28% and 10% for the voice and detection pipelines, respectively. Allowing partial caching cumulatively across memory and storage, such that cache contents in memory and storage are mutually exclusive, improves performance further to 2.35× and 24%. Finally, allowing different steps to be cached in memory and storage improves performance by 5.3× and 38%. These experiments suggest that a caching strategy that allows partial caching of multiple steps across memory and storage can significantly improve the performance of the input preprocessing pipelines.

3.4 Caching vs. re-computing trade-offs

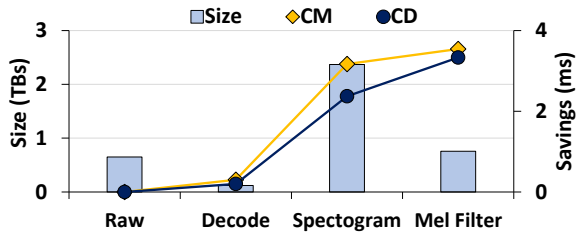
The next question is which pipeline steps to cache in memory and which to cache in storage. The decision depends on three factors: ① the available capacity in each tier, ② the latency of fetching data from that tier, and ③ how that latency compares with the cost of recomputation. To evaluate this trade-off, we run an empirical experiment on a small subset of the dataset.

We define two metrics : C_{M_i} and C_{D_i}, shown in equation 1.

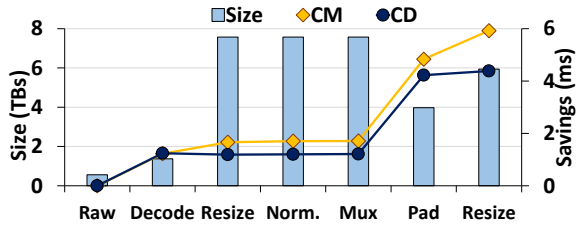
$$C_{M_i} = t_{M_{raw}} - t_{M_i}, \quad C_{D_i} = t_{D_{raw}} - t_{D_i} \tag{1}$$

$t_{M_{raw}}$ and $t_{D_{raw}}$ are the times (in ms) to fetch and preprocess the *raw* tensor from the memory and storage, respectively. [2] Similarly, t_{M_i} and t_{D_i} are the times (in ms) to fetch and preprocess the tensor cached after step i in memory and

[2] "Raw" refers to the original dataset sample without any preprocessing.



(a) **Voice pipeline**



(b) **Detection pipeline**

Figure 3: Savings and Storage vs Pipeline steps. The left y-axis shows the intermediate tensor sizes. The right y-axis is the unit savings(ms) per tensor in memory(C_{M_i}) or storage (C_{D_i}).

storage. Hence, C_{M_i} and C_{D_i} are the times *saved* by employing a caching strategy in memory and storage, respectively.

Figure 3 shows the trends of C_{Mi} and C_{Di} metrics across the preprocessing steps of voice recognition and object detection pipelines (subgraphs). Here, we report the measurement for a *single* tensor without loss of generality. The x-axis lists the preprocessing steps of these pipelines. The height of each bar captures the tensor output size of the given step (lower is better, primary y-axis). The two lines represent the (cumulative) latency savings *per tensor* in ms (secondary y-axis) from caching in memory and storage.

Note C_{M_i} is generally higher than C_{D_i}. As expected, the time savings from caching in memory are higher than caching in storage. The step with the lowest memory or storage footprint, i.e., the height of the bars, can potentially be cached well within a limited budget. However, one should also ensure that caching maximises the time saved on recomputation, i.e., where the saving curve for C_{M_i} (or C_{D_i}) rises steeply. For example, in detection shown in Figure 3(b), 'Pad' generates smaller tensors than 'Mux', yet its compute time is higher. Caching Pad, therefore, needs less capacity but yields substantial time savings. For the voice pipeline shown in Figure 3(a), 'Spectrogram' gives the largest time savings but also the largest tensors. 'Mel Filter' offers nearly the same savings while requiring far less space, making it a better candidate. Here, the most beneficial step to cache depends on the interaction of multiple factors, including available cache size, time to fetch data from the cache, and computation time.

3.5 Offline and online preprocessing steps

Pipelines	Online time %	#online steps
Recognition on ImageNet	7%	1
CubePP	18%	1
Voice recognition, object detection, image recognition and segmentation	0%	None

Table 1: Time spent on online steps across different preprocessing pipelines.

Thus far, we have discussed the steps to cache and how to cache them. An important note is that we cache **only offline steps**. Offline steps are deterministic and idempotent, so their outputs can be cached without altering training. Online steps add stochastic augmentations; caching them would eliminate this randomness and harm augmentation quality. As shown in Table 1, among the six pipelines we study, only image recognition on ImageNet [13] and CubePP [12] contain a single online step, and even then offline steps still account for 82% to 100% of the total preprocessing time.

Step	Action	Type	% time
Raw	Read JPEG	Offline	48%
Decode	JPEG to RGB tensor	Offline	9%
Resize	Scale to 224×224	Offline	26%
Normalise	Convert, channel normalise	Offline	10%
Flip	Random horizontal flip	**Online**	7%

Table 2: Preprocessing steps of image recognition pipeline.

Breakdown of image recognition pipeline: The pipeline comprises four offline steps—raw read, decode, resize, and normalise—and one online step, flip. The offline steps consume 93% of preprocessing time, making them ideal caching targets. The lone online step is left uncached to preserve randomness. As offline steps dominate the runtime, an effective caching policy can concentrate solely on them. This approach preserves data-augmentation randomness—and thus training quality—while still delivering near-maximal caching benefit.

4 HyCache: Key Ideas and Design Principles

Guided by the limitations outlined in Section 3.2, HyCache is designed to accelerate the input preprocessing pipeline by maximising memory and storage utilisation while minimising user effort. The design has following four key contributions:
1. Partial caching of intermediate outputs: HyCache supports partial caching, allowing any fraction of a pipeline stage's outputs to be cached in memory or storage. This relaxes the all-or-nothing constraint of prior systems and enables effective reuse even under limited capacity. Multiple stages can also be partially cached depending on resource availability.

2. Exclusive caching across memory and storage: To avoid duplication, HyCache guarantees that the memory and storage caches contain mutually exclusive data. Each cached tensor is placed in exactly one tier, ensuring efficient and non-redundant use of available hardware.

3. Coordinated tier-aware caching decisions: HyCache uses a profile-guided ILP formulation to determine which steps are best cached in memory versus storage, based on recomputation cost, output size, and access latency. Different steps can be cached in each tier, and the system automatically constructs an optimized pipeline that ingests tensors from memory, storage, or raw data as needed.

4. Automatic estimation of working memory: HyCache analyzes the runtime memory demands of preprocessing—such as batching, buffering, and fetcher overhead—and reserves only the remaining capacity for caching. This eliminates the need for manual tuning while ensuring stability and high resource utilisation. To create an automated framework HyCache does:

(1) Cost-Benefit profiling: Each step in the input pipeline is profiled with just a few sample batches to evaluate the relative merits of caching its tensors in memory and storage.

(2) Identifying the optimal steps to cache: The framework then identifies the optimal step for caching, guided by the memory and storage budgets and the computed cost-benefit ratios using an ILP solver. This approach allows HyCache to scale across different hardware configurations.

(3) Optimised pipeline generation: HyCache finally generates an optimised pipeline that is capable of sourcing tensors from various levels of the input pipeline, utilising both memory and storage resources efficiently.

(4) Working memory size estimation: It calculates the amount of working memory required for executing preprocessing tasks and reserves the rest for caching tensors.

5 Design and Implementation of HyCache

In this section, we describe the implementation details of HyCache. The input to HyCache consists of: (1) a user-defined preprocessing pipeline, (2) the raw dataset of N training samples, (3) a memory budget M, and (4) a storage budget D. Users may also annotate specific steps in the pipeline as "online-only," indicating they should not be cached.

Given these inputs, HyCache produces an optimised pipeline that fetches data from memory, storage, or raw input depending on availability. It also configures and populates memory and storage caches. Figure 4 provides an overview of the components involved, which we describe below.

Annotator and Step Filter: The first step in HyCache is identifying which pipeline stages are worth considering for caching. HyCache analyses the preprocessing pipeline to determine the input/output dimensions of each step. When a sequence of consecutive steps—e.g., S_k, S_{k+1}, S_{k+2}—produces intermediate tensors of identical size, only the final step (S_{k+2}) is considered for caching. The rationale is that since intermediate outputs do not change in size across these steps, caching earlier steps would offer no additional space-related tradeoffs.

To implement this logic, HyCache executes each step in isolation over a sample training batch and records the size of the output tensor. It then identifies sequences of steps with identical output sizes, retaining only the last step of each sequence. These are annotated as `filter_steps`, and they comprise the set of steps to be evaluated for caching. In Figure 4, these steps are highlighted in orange (e.g., `filtered_steps` $= S_2, S_3, S_5$). For example, if S_1 and S_2 produce outputs of the same size, only S_2 is retained for evaluation.

Profiler The profiler evaluates the cost–benefit tradeoff of caching each `filter_step` in memory or storage. For each step $S_i \in$ `filtered_steps`, the profiler computes the expected benefit of caching its output in memory (C_{M_i}) or storage (C_{D_i}), as defined in Equation 1. It also records the average output tensor size Sz_i. While output sizes are typically deterministic, some steps may depend on input characteristics, so Sz_i is averaged over samples.

Profiling is conducted during the first epoch and proceeds until 1% of the dataset is processed. This threshold has been shown to provide a representative view of the overall dataset [26]. Since profiling spans multiple batches, HyCache averages the measurements to account for batch-to-batch variability. This process incurs minimal overhead (Section 7.5).

Tuning number of fetchers Pipeline throughput improves with additional fetchers, but at the cost of higher memory usage. Each fetcher is a separate process responsible for loading and preprocessing data, and increasing the number of fetchers reduces latency but consumes more working memory, thereby reducing the cacheable memory capacity.

To strike the right balance, HyCache performs a binary search over the number of fetchers during the profiling phase. It measures the throughput of preprocessed batch creation for each configuration. The search upper bound is the number of available CPU cores, and the search converges at the point where adding fetchers no longer improves throughput. It identifies the optimal number of fetchers F_i for the pipeline, ensuring efficient data ingestion without overcommitting memory.

ILP solver: At the core of HyCache is an ILP solver, which identifies the optimal steps for caching in the memory and storage based on the profiling. The solver uses the cost savings, C_{Mi} and C_{Di}, and the tensor sizes after each filter_step (Sz_i).The goal is to identify the optimal caching steps in memory (S_{optM}) and storage (S_{optD}), as well as the number of tensors to cache for these steps (N_{optM} and N_{optD}).

The optimal caching policy is formulated using Equations 2 to 5. S_{optM}, S_{optD} is the solution that maximizes the sum in Equation 5, using the constraints in Equations 2 through 4.

$$\sum N_{M_i} Sz_i \le M, \quad \sum N_{D_i} Sz_i \le D \tag{2}$$

$$N_{optM} = \sum N_{M_i}, \quad N_{optD} = \sum N_{D_i} \tag{3}$$

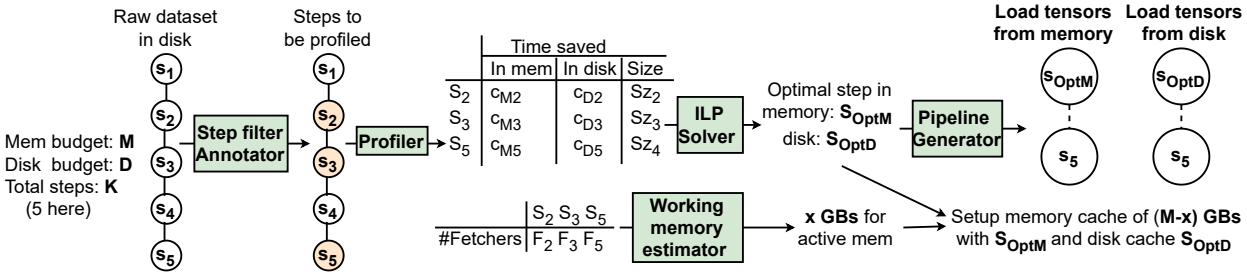


Figure 4: Overview of HyCache.

$$N_{optM} + N_{optD} \leq N \tag{4}$$

$$maximize \sum N_{M_i} C_{M_i} + \sum N_{D_i} C_{D_i} \tag{5}$$

Equation 2 ensures that the cumulative size of tensors at any step remains within the memory and storage budget. Equation 3 defines N_{optM} and N_{optD}, as mentioned previously. Equation 4 constrains the total number of tensors cached jointly in the memory and storage, never to exceed the number of elements in the dataset. It also allows for *partial* caching in both storage and memory, as the number of total tensors cached may be less than or equal to the total elements (N).

Jointly optimising the cache stores of the memory and storage allows convergence into a solution that accounts for the access latency of SSD while deciding what to cache in memory, i.e., a global maxima solution, as opposed to locally maximising the utility of the memory cache in isolation.

HyCache uses a *coordinated* caching policy across memory and storage that ensures the exclusivity of tensors stored in each store. The memory and storage caches are created sequentially, keeping track of cached data items such that tensors cached in memory and storage do not overlap. Equation 4 ensures that, collectively, the memory and storage caches encompass a part of or the entire dataset (i.e, all elements N). The ILP solver finally returns the optimal step(s) for storage (S_{optD}) and the number of tensors to be cached (N_{optD}). Recall from Section 3.4 that C_{M_i} and C_{D_i} represent the recomputation latency saved by caching a tensor in memory and storage, respectively. Note that C_{M_i} and C_{D_i} are measured in the profiling phase. The equation 5 finds the values of N_{M_i} and N_{D_i} that maximizes the saving of recomputation cost.

Figure 4 pictorially depicts the working of key components of HyCache, including the ILP solver. The solver is also capable of identifying and recommending multiple optimal steps for storage caching, along with the number of tensors to cache from each. This capability is helpful when the cache size (e.g., storage cache) is big enough to accommodate outputs of more than one intermediate step.

Working memory size estimation: The size of the working memory directly influences the amount of memory available for caching (Section 3.2). Manually calculating the memory available for caching introduces complexity and is prone to mistakes. If the working memory size is underestimated, then it can lead to an out-of-memory error during preprocessing. Its overestimation leads to memory under-utilisation since the cache size could be set to a lower number than necessary. HyCache alleviates this requirement by dynamically determining the required working memory for a given pipeline, hardware configuration, and dataset, using Equation 6.

$$\begin{aligned} working_mem = (max(Sz_i) \times bs \times prefetch_depth) \\ + (mem_per_fetcher \times max(F_i)) + metadata \end{aligned} \tag{6}$$

The equation incorporates the following: First, it considers the maximum size of intermediate tensors (Sz_i), batch size (bs), and $prefetch_depth$. The $prefetch_depth$ factor is for the number of fully preprocessed batches that can be kept in memory waiting to be transferred to the GPU for training. It is set to two by default. Typically, a batch is processed by parallel threads. Second, the memory requirement of a fetcher is multiplied by the maximum of F_i to ensure sufficient working memory for any possible number of fetchers. Finally, some space is also reserved for Python's metadata, e.g., a list of all files, which is easily determined empirically.

Pipeline creation and cache population: HyCache automatically creates multiple pipelines to ingest data samples at different stages of preprocessing in the pipeline as chosen by the ILP solver. Specifically, HyCache generates one pipeline for each different stage of preprocessing whose output the ILP solver decides to cache. For example, in Figure 4, HyCache creates two pipelines – one ingesting tensors from S_{optM} and another from S_{optD}. Further, since HyCache allows the partial output of different stages of pipelines, it is also possible that the output of none of the steps is fully cached in either of the cache stores. In such cases, HyCache employs an additional pipeline to process the raw data samples.

Allowing data to be ingested from different steps and sources (e.g., memory, SSD) requires modification to a regular pipeline. HyCache annotates the preprocessing pipeline code to automatically insert conditional execution by using *if then else* conditions. For instance, if the output from the last step of a series of steps is fully cached, there is no need to execute all the intermediate steps in the series. In addition, HyCache can conditionally preprocess until a step for fetching partially preprocessed tensors from a cached source.

HyCache populates the memory and storage caches with tensors from the output of pipeline stages, S_{optM} and S_{optD},

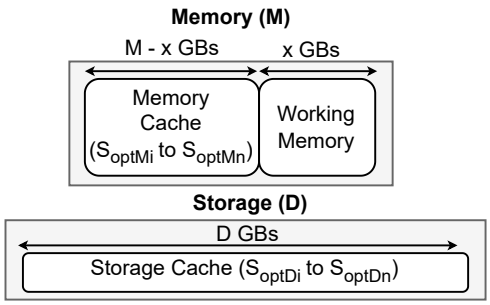


Figure 5: Overview of data placement in memory and storage.

as chosen by the ILP solver, and populates the caches during the execution of the first epoch (epoch 0). HyCache maintains directories that map each sample to the location where it is cached, which is just the filename for a raw sample. HyCache also sets the number of fetchers for different pipelines. For example, it sets the number of fetchers for S_{optD} as determined by the profiler to fetch data from the storage.

The profiler may choose to cache outputs of multiple steps in memory and/or storage if the available capacity permits. Thus, S_{optM} and S_{optD} can be a set of steps rather than a single step.HyCache creates a coordinated cache fetch and preprocessing pipelines for each of the steps in S_{optM} and S_{optD}. HyCache coordinates multiple pipelines to continuously supply a stream of batches from both cached and recomputed preprocessing steps automatically. While HyCache creates multiple internal pipelines, it presents a single pipeline iterator to the user. This abstraction ensures that users can seamlessly harness HyCache without code modifications.

Figure 5 shows the final allocation of both memory and storage resources by HyCache. The total memory budget M is split into two parts: x GB for working memory and $M-x$ GB for the in-memory cache, which is populated with the optimal in-memory steps $S_{\text{Opt}M_i}$ through $S_{\text{Opt}M_n}$. Simultaneously, the storage cache is populated with the optimal on-storage steps $S_{\text{Opt}D_i}$ through $S_{\text{Opt}D_n}$.

Training: With HyCache, the training remains unaffected. HyCache's improvements are solely limited to the input preprocessing pipeline. The data is immediately ready for training on any computational device after batch generation by the pipeline. HyCache further extends support for multi-GPU environments, where the CPU manages the input pipeline while facilitating concurrent training across several GPUs.

6 Ease of deploying HyCache

```
from hcLib import BasePipeline, HyCache                    1
class ImageRecognition(BasePipeline):                      2
    def defineGraph(self):                                 3
    ...                                                    4
    # Define the preprocessing steps here                  5
    ...                                                    6
# Include other arguments here too.                        7
HyCachePipeline = HyCache(ImageRecognition, M=100%,
    D=500, dataset_path="/path/to/dataset")                9
iterator = HyCachePipeline.build()                         10
# Initialize your model                                    11
model = models.resnet50().cuda()                           12
for batch in iterator:                                     13
    train_model(model, batch)                              14
```

Listing 1: Using `hcLib` library for an input pipeline.

In this section, we show how a user can leverage HyCache. Its key functionalities are encapsulated in a library that we name `hcLib`. A key benefit of using HyCache is that it needs minimal changes to a regular model training process. Listing 1 shows how a user can interface with our library. The user needs to first import `BasePipeline` as usual and then `HyCache` classes from the library `hcLib` (line 1). `BasePipeline` is a wrapper class over `nvidia.dali.Pipeline` object that internally enables profiling and other required changes to support our programming paradigm.

The user-defined preprocessing class must inherit this class instead of the original `nvidia.dali.Pipeline` (lines 2-6). Then, to finally enable pipeline optimisations and get the iterator, the user has to initialise an object of `HyCache` with the Pipeline class (lines 8-9). This object initialisation requires arguments such as the total memory budget (note, not the cache size), storage budget, batch size, dataset path, or list of dataset files, etc. Other important arguments include `storage_cloc`, which is the location of the storage cache, `world_size`, which should be used in case of distributed training, and `cache_steps`, which is an integer to indicate consideration for caching only up to a given step. If this argument is not provided, it is assumed that all the steps are offline and can be considered for caching. It is also important to note that the allowed budgets for memory and SSD are never exceeded. However, the value provided by the user cannot be smaller than the expected active memory size. Also, the user should expect an out-of-memory error if the memory budget exceeds the available free memory in the system.

To get the iterator for continually yielding batches, the user has to call the `build()` method on the returned object (line 10). Next, as in a regular training script, the user initialises their training model (line 12). We show an example of creating a `resnet50` model, with GPU training enabled using the `.cuda()` method. There is no constraint on using the GPU for training. Any other accelerator, even the CPU, can be used. However, the user needs to consider the memory bloat of the model in case the CPU is used. Finally, the user can use regular loop-yielding style calls to the iterator to get batches and do further training on the model (lines 13-14).

7 Evaluation

We present the performance evaluation of HyCache on a single server setup. We empirically assess the efficiency of Hycache's caching policies against prior works and the baseline. We then quantify improvements in end-to-end training jobs.

7.1 Methodology

Pipeline	Dataset	Model
Image Recognition-1 (IR-1) [47]	OpenImagesV5 (561GB) [24, 29]	MobileViT-BS16 [32]
Image Recognition-2 (IR-2) [13]	Imagenet-22K *(650GB) [19]	MobileNetV3-BS16 [18] ResNet50-BS16 [17] ViT-LoRA-BS3584 [52]
CubePP [20]	CubePP-JPG *(550GB) [12]	–
Segmentation [44]	Imagenet-22K *(650GB) [19]	LRASPP-BS16 [18, 37]
Object Detection [45]	OpenImagesV5 (561GB) [24, 29]	YoloV5-BS16 [39, 42]
Voice Recognition [20, 46]	FMA *(650GB) [5]	M5-BS12 [10]

Table 3: Pipeline characteristics and configurations. '*' indicates the dataset was reduced or augmented.

Workloads: Our evaluation spans six distinct preprocessing pipelines, listed in Table 3, covering a range of ML tasks including object detection, image recognition, voice recognition, and segmentation. Each pipeline is paired with a dataset and a downstream DNN model used for training.

Note that multiple models may share the same preprocessing pipeline and dataset. For example, MobileNet, ResNet50, and ViT-LoRA all use the same ImageNet-22K dataset and pipeline. However, the degree of preprocessing bottleneck experienced by each model can differ significantly depending on its architectural complexity and computational intensity.

All DNN models are sourced from the TorchVision library [1, 31] or implemented using `torch.nn` [35]. Table 3 also includes the batch sizes used in both end-to-end training and preprocessing-only evaluations. These batch sizes represent the largest configurations that could fit on a single GPU under data-parallel training.

Datasets marked with '*' have been reduced or augmented to fit within our 2 TB storage limit (see Table 4), ensuring sufficient space for storage-based caching.

Since our primary goal is to accelerate the input preprocessing pipeline, we focus first on how HyCache improves preprocessing throughput, independent of the model's training speed. Later in this section, we evaluate the end-to-end impact of HyCache on overall training time, including both preprocessing and model execution.

Experimental setup: Our evaluation platform is listed in Table 4. We show the performance of HyCache on various configurations, listed in Table 5, derived from this base platform. We design these configurations based on the available VM setups on Google Cloud Platform [9]. For example, for an A100 GPU with 40 GBs of memory, 85 GBs of CPU memory, and 12 vCPUs on the CPU are available for a single VM instance, and this ratio is consistent across their A100 offerings. We mention the evaluation configuration used for each experiment explicitly and mention any changes in the evaluation setup in their respective subsection. By default, for all experiments, we use the configuration C2 unless otherwise mentioned. Note that each vCPU is mapped to a separate physical core. In short, we do not oversubscribe the cores.

CPU	AMD EPYC 7313 16-Core @ 64× 3GHz
DRAM	512GB DDR4
SSD	2TB Samsung 980 PRO
Software	Python-3.10, Nvidia DALI v2.1

Table 4: Configuration of the evaluation platform.

Config	vCPUs	Memory	SSD Size
C1	12	85GB	250GB
C2	24	170GB	500GB
C3	24	170GB	1000GB
C4	48	340GB	500GB
C5	48	340GB	1000GB

Table 5: Configurations used for evaluation

Training environment and parameters: We implement HyCache atop Nvidia DALI [7]. We use Python 3.10 and DALI-v2.1.0 to create this library. Although we use PyTorch for model training, the `hcLib` library is agnostic to the model framework used. For end-to-end training, we use Pytorch's distributed data-parallel training [38] to allow training on multiple GPUs. For all experiments, the number of threads processing a batch is set to the number of cores available (e.g., 12 for 12 cores/vCPUs). The number of fetchers for HyCache is set to the ideal number of fetchers determined by its profiler. Related works do not have the ability to choose the number of fetchers automatically. Thus, while evaluating them, we set the number of fetchers to the number of available CPU cores.

Comparison with other caching policies: We measure our performance against two works that also optimise the input pipeline, MinIO [33] and PRESTO [20]. MinIO caches random data items as they are fetched from storage to populate the memory cache. Once the cache capacity is reached, MinIO does not evict any items in the cache. Instead, the requests to other data items fetch them directly from the storage. PRESTO decides the ideal step to cache in the storage, based on profiling. We also implemented PRESTO in DALI [7] to compare with our solution, where we choose the best step to cache that can *completely* fit in the given storage budget. If PRESTO does not have a step that can fit in the storage, it uses the raw dataset.

7.2 Performance of input pipelines

Figure 6 shows the normalised performance of HyCache and its competitors across various input pipelines. Since we

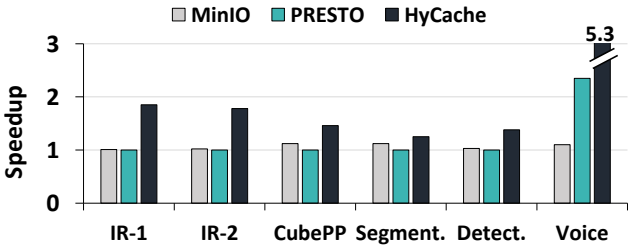


Figure 6: Performance of different caching policies.

Pipeline	MinIO	PRESTO	HyCache
IR-1	Fixed Pagecache Memory + Raw	OS Managed Memory + Raw	5M + 3D + Raw
IR-2			2M + 2D + Raw
CubePP-JPG			2M + 3M + 2D + Raw
Segmentation			3M + 3D + Raw
Detection			6M + 5D + Raw
Voice		OS Managed Memory + Step 1D	3M + 1D + 3D

Table 6: Strategies used for evaluation in section-7.1

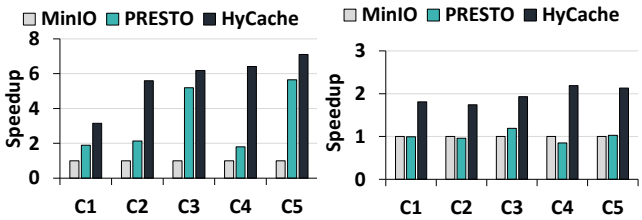
focus on improving input processing pipelines while leaving the training unchanged, we isolate the performance of the pipelines by measuring the number of samples preprocessed by each pipeline under various platforms, MinIO, PRESTO, and HyCache. The heights of each bar are normalised over the number of samples processed under the traditional OS page cache (higher is better). MinIO speeds up pipelines by up to 12% (average, 7%) [3] PRESTO significantly speeds up the voice pipeline as it could fit the output of the optimal intermediate step in its storage cache, but not for other pipelines.

We observe a speedup in the range of 1.11×-5.3× over MinIO, and 1.24×-2.26× over PRESTO. The improvements were the largest for the voice pipeline – through the coordinated caching in memory and storage, HyCache was able to save more than 80% of the compute time. The image processing pipelines (IR-1 and IR-2) save 58% and 42% of recomputing time and speed up by 1.84× and 1.74×, respectively, over MinIO. PRESTO can significantly outperform MinIO only for the voice pipeline through caching in storage.

Table 6 summarises the caching strategies employed by various platforms. MinIO does not cache the outputs of intermediate preprocessing steps. Instead, it focuses on caching raw datasets via the OS's page cache, optimising I/O for unprocessed samples. PRESTO supports caching intermediate outputs in storage, enabling the reuse of either raw or preprocessed samples by loading them into the OS page cache.

In contrast, HyCache offers significantly more flexible and

[3]The authors of MinIO [33] reported up to 50% speedup (Figure 13(a)). Our setup uses 3× faster SSD than the one used in the original paper [33]. This reduced the scope for MinIO's speedup.



(a) **Voice Pipeline** (b) **IR-2 pipeline**

Figure 7: Performance across different hardware configs.

fine-grained caching capabilities. It can cache the outputs of any intermediate step or the raw dataset itself. Moreover, HyCache supports caching different preprocessing stages in memory and storage independently. It can even cache outputs from multiple distinct steps within the same tier—either memory or storage—depending on available resources and recomputation tradeoffs.

In Table 6, nM' denotes that the output of the n^{th} intermediate step is cached in memory, while kD' indicates that the k^{th} step's output is cached in storage. Entries marked 'Raw' means raw dataset is cached directly in memory or storage.

This table highlights HyCache's versatility in leveraging both memory and storage to reduce preprocessing costs. For instance, in the voice pipeline, HyCache caches the output of the 3$^{\text{rd}}$ step in both memory and storage, and also caches the 1$^{\text{st}}$ step's output in storage. This coordinated strategy improves resource utilisation and minimises recomputation.

In summary, HyCache enables adaptable, multi-tier caching policies that exploit the full capacity of available memory and storage to accelerate input pipelines.

7.3 Performance scaling and Adaptability

A key advantage of HyCache is its ability to automatically configure cache sizes and determine the number of fetcher threads, in contrast to prior approaches that require manual tuning. In this section, we empirically demonstrate HyCache's adaptability to varying hardware configurations.

We evaluate the performance of HyCache against PRESTO and MinIO across five different system setups, listed in Table 5. Figures 7(a) and 7(b) show the normalized throughput of the three platforms for two representative pipelines: Voice and Image Recognition-2 (IR-2). On the x-axis, we vary the hardware configurations, while the y-axis reports throughput normalized to MinIO (higher is better).

Across all configurations, HyCache consistently outperforms both baselines. For the voice pipeline, HyCache achieves 3.14× to 7.10× speedup over MinIO and 1.2× to 3.5× over PRESTO. For the IR-2 pipeline, it provides 1.73× to 2.13× improvement over MinIO and 1.61× to 2.57× over PRESTO.

Further analysis reveals that for the voice pipeline, both PRESTO and HyCache choose to cache the output of the same step (step 3). However, HyCache outperforms PRESTO

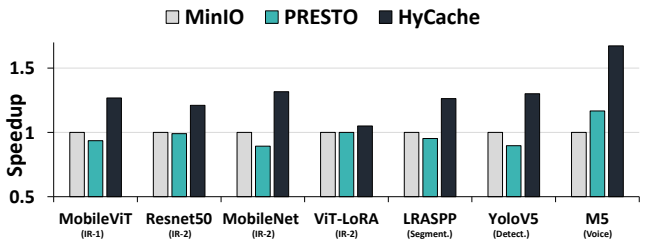


Figure 8: Improvement in end-to-end training throughput.

due to its coordinated use of memory and storage, which PRESTO lacks. In configurations like C4, HyCache's support for partial caching gives it a significant edge, allowing it to utilize constrained resources more effectively.

We observe similar trends in the IR-2 pipeline. In some configurations, HyCache gains from its partial caching strategy, while in others, its ability to split caching across memory and storage proves more beneficial.

7.4 End to End Training

Thus far, we have focused on evaluating preprocessing pipelines in isolation. In this section, we quantify the impact of HyCache on full end-to-end training jobs, including model training on the GPU. The models and corresponding batch sizes used in these experiments are listed in Table 3.

Figure 8 presents the normalized end-to-end training performance for each model. Unlike prior graphs, which were pipeline-centric, this figure reports model-specific speedups. As shown in Table 3, multiple models may share the same preprocessing pipeline—e.g., MobileNet and ResNet both use the IR-2 pipeline for image recognition. To clarify this, we annotate the associated pipeline below each model name on the x-axis.

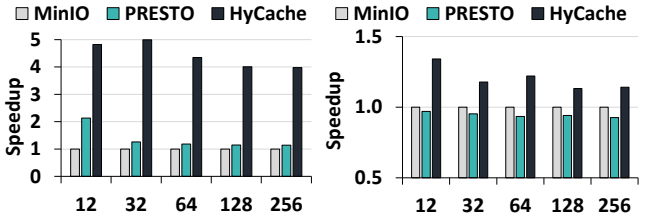
This experiment uses configuration C4 from Table 4. We observe that HyCache delivers speedups ranging from *1.05*× to *1.67*× over MinIO, and from *1.05*× to *1.47*× over PRESTO. These gains depend on both the preprocessing bottleneck and the GPU training load of the model.

For instance, ViT-LoRA—based on the `vit_base_patch16_224` architecture is the most compute-heavy model in our evaluation, and achieves a more modest *1.05*× speedup.

Because DNN models are computationally intensive, improvements in the input pipeline only impact a portion of the total training time. Nonetheless, HyCache achieves up to *1.67*× end-to-end speedup without any GPU-level training optimisations. Notably, we do not apply techniques such as mixed-precision training (e.g., Apex [6]), which could further shift the performance bottleneck toward input preprocessing, potentially increasing the benefits of HyCache.

Pipeline	Profile(s)	Epoch(s)	Over head %	Batch size
IR-1	427	7016	6.1	48MB
IR-2	273	2093	13.0	135MB
CubePP-JPG	496	2794	17.8	152MB
Segmentation	648	3567	18.2	192MB
Detection	700	10075	6.9	64MB
Voice	204	8150	2.5	43MB

Table 7: Profiling overhead as a percentage of a vanilla epoch.



(a) **Voice Pipeline** (b) **Detection pipeline**

Figure 9: Performance of pipelines across different batch sizes.

7.5 Profiling Overhead

HyCache profiles 1% dataset to find out the output size of each of the intermediate steps of a pipeline. It also profiles the runtimes of each step of a pipeline. The ILP solver uses these for choosing a caching strategy. We report the profiling overheads in Table 7. We report the time needed for profiling in seconds, the time needed to preprocess samples for an epoch. We report the batch size in MB. The profiling overhead ranges from 2.5% to 18.2% of the preprocessing time of *one* epoch (i.e, one pass over the dataset). The profiling is done *only* during the first epoch, while a typical training job runs for hundreds of epochs. Thus, it adds little overhead to the overall preprocessing pipeline. Note that the profiling overheads vary across pipelines based on the number of steps in the pipeline. The higher the number of steps, the larger the profiling overheads. Further, a larger batch size (bytes) increases overheads as larger amounts of raw data must be accessed during profiling.

7.6 Sensitivity to Varying Batch Sizes

The batch size used for model training is a hyperparameter. This section highlights that our framework is able to adapt to varying batch sizes and offers consistently high performance across different sizes compared to prior work. Figure 9 shows the normalised performance of the two representative pipelines under varying batch sizes for the three frameworks. As before, the throughputs are normalised to those under MinIO. We notice that across batch sizes, HyCache outperforms other platforms. This shows that HyCache is able to adapt well to

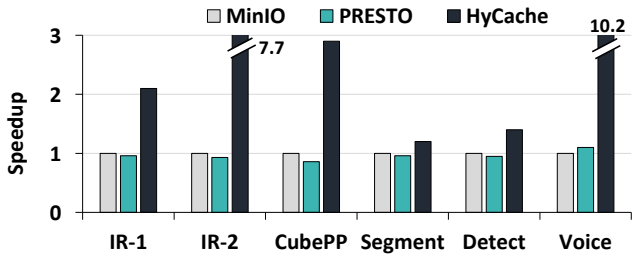


Figure 10: Throughput with remotely connected storage.

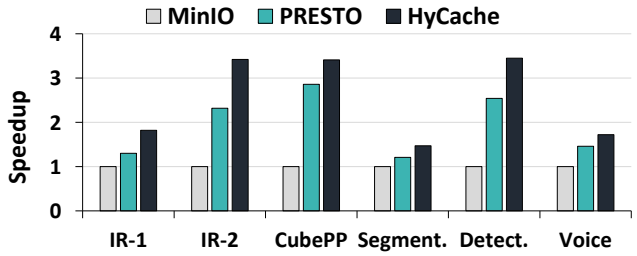


Figure 11: Pipeline throughputs for small datasets

different batch sizes. We also notice that increasing batch size often reduces the speedups for HyCache. Larger batch sizes necessitate larger working memory for computation and, thus, smaller memory caches. This, in turn, limits HyCache's ability to leverage caches for limiting recomputation.

7.7 Implications of Remote Storage

Finally, we evaluate the implications of keeping the raw dataset in a remote NFS file server [2] This emulates the scenario where the dataset is remotely located in a cloud storage. Fetching raw data from a remote server introduces network latency and potential bandwidth constraints into the data ingestion process, making it worthwhile to evaluate it separately.

For this experiment, we use configuration C4 from Table 5. Figure 10 shows the performance of PRESTO and HyCache relative to MinIO. We observe a speedup in the approximate range of 1.11× - 10.1× over MinIO, and 1.19× - 9.28× over PRESTO. The IR-2 and voice pipeline witness substantial speedups with HyCache over the other two because they have a solution that completely fits in the available storage and memory budget. In contrast, the other two suffer a significant drop in performance since they have to rely on the NFS throughput. The maximum available throughput on this device is 64.8 MiB/s, which we tested using the `fio` [4] utility. In summary, HyCache is able to perform well across a wide variety of settings and configurations – a treatment to HyCache's ability to automatically adapt to varying deployment scenarios.

7.8 Analyzing PRESTO's usefulness

One would notice that PRESTO [20] is generally ineffective in speeding up preprocessing pipelines, except for the voice

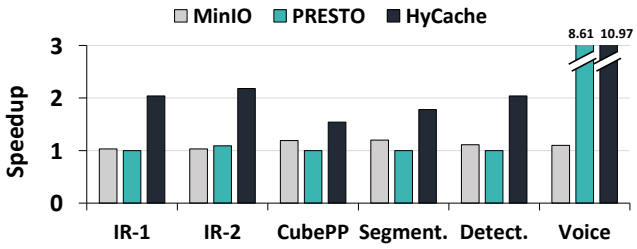


Figure 12: Pipeline throughput for C5 configuration.

pipeline. The efficacy of PRESTO is limited by its inability to partially cache the output of an intermediate step and its inability to harness both memory and storage for coordinated caching, unlike HyCache. We found that PRESTO is useful only when used with small datasets such that the output of all or most of the intermediate steps could fit well in its cache budget. The original paper used datasets as small as 250MBs, up to 146GBs. However, small datasets are contrary to current trends of using ever larger dataset sizes. In our evaluation, we use datasets that exceed at least 550GBs.

To demonstrate PRESTO's efficacy for smaller datasets, we limited the preprocessing pipelines to use only 100 GB of the dataset but allocated 1650GB storage cache. Figure 11 shows the performance of different pipelines under MinIO, PRESTO, and HyCache, normalised to MinIO. For smaller datasets, PRESTO provides a more significant speedup of 1.21×-2.86× over MinIO. HyCache performs 1.3× better on average over PRESTO in this setting as well, demonstrating its adaptability to varying system configurations.

Finally, while Figure 6 reported performance under configuration C2, we further evaluate the effectiveness of different caching mechanisms under configuration C5, which features a larger SSD, as shown in Figure 12.

Despite the increased storage capacity, a 1000 GB SSD was still insufficient to fully cache intermediate outputs for four of the six pipelines. Among the successful cases, the Voice pipeline achieved the optimal caching configuration and delivered a speedup of 8.61×, while the IR-2 pipeline showed a modest improvement of 1.09×.

In summary, PRESTO provides performance gains only in limited scenarios, typically when the pipeline fits entirely within the storage budget. In contrast, HyCache continues to improve performance even under tighter resource constraints due to its support for partial and coordinated caching.

7.9 HyCache in distributed environment

In this subsection, we will discuss how HyCache can be extended in a distributed environment. The current implementation of HyCache uses a single node to demonstrate its key ideas. However, its design can seamlessly operate in distributed environments where data is partitioned across multiple

machines. In homogeneous clusters—where hardware configurations are identical and data is evenly distributed—the profiler and ILP solver can be run once, and the resulting cache plan applied uniformly across all nodes. In contrast, heterogeneous clusters with varying hardware or skewed data distributions can run HyCache independently on each machine, allowing each node to tailor its caching strategy to local workload and resource characteristics. Since the profiler relies only on the underlying hardware and the pipeline, HyCache in a distributed setting does not require any coordination across nodes and can seamlessly generate distributed plans.

8 Related Works

Distributed training: FastFlow [43] automatically offloads input pipeline preprocessing to remote CPUs, aided by lightweight profiling. Meta [51] discusses preprocessing bottlenecks causing stalls in their Deep Learning Recommendation Models(DLRMs). Cachew [14] is a fully managed service for ML data processing that dynamically scales distributed resources to avoid stalls in training jobs. Cachew's autocaching policy fails when materialized tensors are too large to fit in memory or disk. Nectar [16] focuses on automating the management of data and computation across their cluster of many-node servers. Our work is orthogonal to Nectar and our caching strategies can be used as a basis for cache creation. Quiver [28] boosts DNN training efficiency with transparent data reuse, flexible cache hits, and job-aware allocation. The distributed cache that is shared across multiple users and training jobs can use our caching policy coupled with their hash-based addressing to boost training performance.

Random Augmentations: Several recent works have explored techniques for optimizing DNN training pipelines through caching randomly augmented datasets, i.e., cache creation, coupled with cache management or replacement. Data Echoing [8] t reuses intermediate outputs from earlier pipeline stages to reclaim idle accelerator capacity for faster training when reading data over a network. Data subsampling [27] utilizes subsampled versions of training data for augmentation, reducing computation and memory requirements while maintaining performance. Revamper [30] reuses partially augmented samples within specific stages of the pipeline to speed up training. While these works focus on speeding up random augmentation, we focus on offline/deterministic preprocessing steps that has reuse. Caching random steps can affect training accuracy and convergence. However, HyCache does not modify the output of a pipeline and has no impact on accuracy.

Caching Policies and single node training: MinIO [33] reduces the amount of disk I/O (fetch stalls) in single-server training. While saving disk I/O can be useful, our analysis shows that saving computation by using cached tensors proves to be more advantageous than caching raw data in memory. Cachew [14] also considers caching intermediate step on the memory or optionally to the disk, an all-or-nothing policy that may fail. Plumber [26] uses `tf.data.Dataset.cache` to greedily cache a pre-processed `Dataset` into memory or storage. Greedily choosing a step to fully fit against HyCache partial caching does not perform well, as shown in Figure 2.

Accelerator-assisted preprocessing: DALI allows offloading pre-processing to the GPU. However, it may cause interference with the training or inference task itself [33], and limit available GPU memory for training itself. Similarly, SMOL [21] also presents a system that offloads preprocessing to an accelerator, by pipelining preprocessing of image-based data and model training on the GPU while controlling accuracy convergence.

9 Conclusion

We present HyCache, a novel hybrid caching mechanism designed to accelerate DNN training pipelines. It leverages both memory and storage for caching, orchestrating the usage of these resources automatically, showing a significantly reduced data preprocessing time, thereby addressing the bottleneck in DNN training input pipelines. HyCache can achieve substantial performance improvements, with gains ranging from 1.11× to 5.3× over traditional preprocessing pipelines.

Acknowledgments

This material is based upon work supported by NSF award number 2224319, the REAL@USC-Meta center, a VMware gift, the Rukmini Gopalakrishnachar Visiting Faculty Chair at IISc, and an AMD research grant. This work is also partially supported by a Google PhD Fellowship awarded to Shweta Pandey. The views, opinions, and/or findings expressed are those of the author(s) and should not be interpreted as representing the official views or policies of the U.S. Government.

References

[1] Torchvision models. https://pytorch.org/vision/stable/models.html, 2023.

[2] Service - nfs. https://ubuntu.com/server/docs/service-nfs, 2024. Accessed: 2024-02-05.

[3] Bharath Ramsundar Volodymyr Kuleshov Mark DePristo Katherine Chou Claire Cui Greg S. Corrado Sebastian Thrun Jeff Dean Andre Esteva, Alexandre Robicquet. A guide to deep learning in healthcare. *Nature Medicine*, 25(1):24–29, 2019.

[4] Jens Axboe. Flexible I/O Tester, 2022.

[5] Kirell Benzi, Michaël Defferrard, Pierre Vandergheynst, and Xavier Bresson. FMA: A dataset for music analysis. *CoRR*, abs/1612.01840, 2016.

[6] NVIDIA Developer Blog. Nvidia apex: Tools for easy mixed-precision training in pytorch. https://developer.nvidia.com/blog/apex-pytorch-easy-mixed-precision-training/, 2018.

[7] NVIDIA Developer Blog. Fast ai data preprocessing with nvidia dali. https://devblogs.nvidia.com/fast-ai-data-preprocessing-with-nvidia-dali/, 2019.

[8] Dami Choi, Alexandre Passos, Christopher J. Shallue, and George E. Dahl. Faster neural network training with data echoing. *CoRR*, abs/1907.05550, 2019.

[9] Google Cloud. Google cloud gpus documentation. https://cloud.google.com/compute/docs/gpus#a100-gpus, 2024.

[10] Wei Dai, Chia Dai, Shuhui Qu, Juncheng Li, and Samarjit Das. Very deep convolutional neural networks for raw waveforms. In *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, page 421–425. IEEE Press, 2017.

[11] Junhua Ding, XinChuan Li, and Venkat N. Gudivada. Augmentation and evaluation of training data for deep learning. In *2017 IEEE International Conference on Big Data (Big Data)*, pages 2603–2611, 2017.

[12] Egor Ershov, Alexey Savchik, Illya Semenkov, Nikola Banić, Alexander Belokopytov, Daria Senshina, Karlo Koščević, Marko Subašić, and Sven Lončarić. The cube++ illumination estimation dataset. *IEEE Access*, 8:227511–227527, 2020.

[13] Hugging Face. Transforms factory in pytorch image models. https://github.com/huggingface/pytorch-image-models/blob/main/timm/data/transforms_factory.py, 2019.

[14] Dan Graur, Damien Aymon, Dan Kluser, Tanguy Albrici, Chandramohan A. Thekkath, and Ana Klimovic. Cachew: Machine learning input data processing as a service. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 689–706, Carlsbad, CA, July 2022. USENIX Association.

[15] Alex Graves, Abdel-rahman Mohamed, and Geoffrey E. Hinton. Speech recognition with deep recurrent neural networks. *CoRR*, abs/1303.5778, 2013.

[16] Pradeep Kumar Gunda, Lenin Ravindranath, Chandu Thekkath, Yuan Yu, and Li Zhuang. Nectar: Automatic management of data and computation in data centers. Technical Report MSR-TR-2010-55, May 2010.

[17] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015.

[18] Andrew Howard, Mark Sandler, Grace Chu, Liang-Chieh Chen, Bo Chen, Mingxing Tan, Weijun Wang, Yukun Zhu, Ruoming Pang, Vijay Vasudevan, Quoc V. Le, and Hartwig Adam. Searching for mobilenetv3, 2019.

[19] ImageNet. Imagenet-22k dataset. http://www.image-net.org/releases, 2020.

[20] Alexander Isenko, Ruben Mayer, Jeffrey Jedele, and Hans-Arno Jacobsen. Where is my training bottleneck? hidden trade-offs in deep learning preprocessing pipelines. In *Proceedings of the 2022 International Conference on Management of Data*, SIGMOD '22, page 1825–1839, New York, NY, USA, 2022. Association for Computing Machinery.

[21] Daniel Kang, Ankit Mathur, Teja Veeramacheneni, Peter Bailis, and Matei Zaharia. Jointly optimizing preprocessing and inference for dnn-based visual analytics. *Proc. VLDB Endow.*, 14(2):87–100, oct 2020.

[22] Inhan Kim, Joonyeong Lee, and Daijin Kim. Learning mixture of domain-specific experts via disentangled factors for autonomous driving. *Proceedings of the AAAI Conference on Artificial Intelligence*, 36(1):1148–1156, Jun. 2022.

[23] Ana Klimovic. Rethinking data storage and preprocessing for ml. https://www.sigarch.org/rethinking-data-storage-and-preprocessing/, 2021.

[24] Ivan Krasin, Tom Duerig, Neil Alldrin, Vittorio Ferrari, Sami Abu-El-Haija, Alina Kuznetsova, Hassan Rom, Jasper Uijlings, Stefan Popov, Shahab Kamali, Matteo Malloci, Jordi Pont-Tuset, Andreas Veit, Serge Belongie, Victor Gomes, Abhinav Gupta, Chen Sun, Gal Chechik, David Cai, Zheyun Feng, Dhyanesh Narayanan, and Kevin Murphy. Openimages: A public dataset for large-scale multi-label and multi-class image classification. *Dataset available from https://storage.googleapis.com/openimages/web/index.html*, 2017.

[25] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C.J. Burges, L. Bottou, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 25. Curran Associates, Inc., 2012.

[26] Michael Kuchnik, Ana Klimovic, Jiri Simsa, Virginia Smith, and George Amvrosiadis. Plumber: Diagnosing

and removing performance bottlenecks in machine learning data pipelines. In D. Marculescu, Y. Chi, and C. Wu, editors, *Proceedings of Machine Learning and Systems*, volume 4, pages 33–51, 2022.

[27] Michael Kuchnik and Virginia Smith. Efficient augmentation via data subsampling. *CoRR*, abs/1810.05222, 2018.

[28] Abhishek Vijaya Kumar and Muthian Sivathanu. Quiver: An informed storage cache for deep learning. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 283–296, Santa Clara, CA, February 2020. USENIX Association.

[29] Alina Kuznetsova, Hassan Rom, Neil Alldrin, Jasper Uijlings, Ivan Krasin, Jordi Pont-Tuset, Shahab Kamali, Stefan Popov, Matteo Malloci, Alexander Kolesnikov, Tom Duerig, and Vittorio Ferrari. The open images dataset v4: Unified image classification, object detection, and visual relationship detection at scale. *International Journal of Computer Vision*, 128(7):1956–1981, March 2020.

[30] Gyewon Lee, Irene Lee, Hyeonmin Ha, Kyunggeun Lee, Hwarim Hyun, Ahnjae Shin, and Byung-Gon Chun. Refurbish your training data: Reusing partially augmented samples for faster deep neural network training. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 537–550. USENIX Association, July 2021.

[31] TorchVision maintainers and contributors. TorchVision: PyTorch's Computer Vision library. https://github.com/pytorch/vision, November 2016.

[32] Sachin Mehta and Mohammad Rastegari. Mobilevit: Light-weight, general-purpose, and mobile-friendly vision transformer. In *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022*. OpenReview.net, 2022.

[33] Jayashree Mohan, Amar Phanishayee, Ashish Raniwala, and Vijay Chidambaram. Analyzing and mitigating data stalls in dnn training. *Proc. VLDB Endow.*, 14(5):771–784, jan 2021.

[34] Derek G. Murray, Jiří Šimša, Ana Klimovic, and Ihor Indyk. tf.data: a machine learning data processing framework. *Proc. VLDB Endow.*, 14(12):2945–2958, jul 2021.

[35] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[36] Luis Perez and Jason Wang. The effectiveness of data augmentation in image classification using deep learning. *CoRR*, abs/1712.04621, 2017.

[37] PyTorch. Lraspp model - torchvision. https://pytorch.org/vision/main/models/lraspp.html, 2022.

[38] PyTorch. Distributed data parallel training tutorial. https://pytorch.org/tutorials/intermediate/ddp_tutorial.html, 2024. Accessed: 2024-02-06.

[39] Joseph Redmon, Santosh Kumar Divvala, Ross B. Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*, pages 779–788. IEEE Computer Society, 2016.

[40] Cristina Silvano, Daniele Ielmini, Fabrizio Ferrandi, Leandro Fiorin, Serena Curzel, Luca Benini, Francesco Conti, Angelo Garofalo, Cristian Zambelli, Enrico Calore, Sebastiano Fabio Schifano, Maurizio Palesi, Giuseppe Ascia, Davide Patti, Stefania Perri, Nicola Petra, Davide De Caro, Luciano Lavagno, Teodoro Urso, Valeria Cardellini, Gian Carlo Cardarilli, and Robert Birke. A survey on deep learning hardware accelerators for heterogeneous HPC platforms. *CoRR*, abs/2306.15552, 2023.

[41] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

[42] Ultralytics. Yolov5: A highly performant object detection framework. *https://github.com/ultralytics/yolov5*, 2020.

[43] Taegeon Um, Byungsoo Oh, Byeongchan Seo, Minhyeok Kweun, Goeun Kim, and Woo-Yeon Lee. Fastflow: Accelerating deep learning model training with smart offloading of input data pipeline. *Proc. VLDB Endow.*, 16(5):1086–1099, jan 2023.

[44] Kaggle user: ligtfeather. Semantic segmentation is easy with pytorch. https://www.kaggle.com/code/ligtfeather/semantic-segmentation-is-easy-with-pytorch/notebook, 2021.

[45] Kaggle user: outwrest. Yolov5 roi batch dali preprocessing pipeline. https://www.kaggle.com/code/outwrest/yolov5-roi-batch-dali-preprocessing-pipeline/notebook, 2022.

[46] Kaggle user: thailssonclementino. Mel spectrograms with nvidia's data loading. https://www.kaggle.com/code/thailssonclementino/melspectrograms-with-nvidia-s-data-loading, 2020.

[47] Kaggle user: umongsain. Vision transformer from scratch in pytorch. https://www.kaggle.com/code/umongsain/vision-transformer-from-scratch-pytorch, 2022.

[48] Subhashini Venugopalan, Marcus Rohrbach, Jeffrey Donahue, Raymond Mooney, Trevor Darrell, and Kate Saenko. Sequence to sequence - video to text. In *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, December 2015.

[49] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V. Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, Jeff Klingner, Apurva Shah, Melvin Johnson, Xiaobing Liu, Lukasz Kaiser, Stephan Gouws, Yoshikiyo Kato, Taku Kudo, Hideto Kazawa, Keith Stevens, George Kurian, Nishant Patil, Wei Wang, Cliff Young, Jason Smith, Jason Riesa, Alex Rudnick, Oriol Vinyals, Greg Corrado, Macduff Hughes, and Jeffrey Dean. Google's neural machine translation system: Bridging the gap between human and machine translation. *CoRR*, abs/1609.08144, 2016.

[50] Yi Xu, Qi Qian, Hao Li, and Rong Jin. Why does multi-epoch training help? *CoRR*, abs/2105.06015, 2021.

[51] Mark Zhao, Niket Agarwal, Aarti Basant, Buğra Gedik, Satadru Pan, Mustafa Ozdal, Rakesh Komuravelli, Jerry Pan, Tianshu Bao, Haowei Lu, Sundaram Narayanan, Jack Langman, Kevin Wilfong, Harsha Rastogi, Carole-Jean Wu, Christos Kozyrakis, and Parik Pol. Understanding data storage and ingestion for large-scale deep recommendation model training: Industrial product. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ISCA '22, page 1042–1057, New York, NY, USA, 2022. Association for Computing Machinery.

[52] Yitao Zhu, Zhenrong Shen, Zihao Zhao, Sheng Wang, Xin Wang, Xiangyu Zhao, Dinggang Shen, and Qian Wang. Melo: Low-rank adaptation is better than fine-tuning for medical image diagnosis. *CoRR*, abs/2311.08236, 2023.

The Koala Benchmarks for the Shell: Characterization and Implications

https://kben.sh

Evangelos Lamprou
Brown University

Ethan Williams
Brown University

Georgios Kaoukis
National Technical University of Athens

Zhuoxuan Zhang
Brown University

Michael Greenberg
Stevens Institute of Technology

Konstantinos Kallas
University of California, Los Angeles

Lukas Lazarek
Brown University

Nikos Vasilakis
Brown University

Abstract

KOALA is a benchmark suite aimed at performance-oriented research targeting the Unix and Linux shell. It combines a systematic collection of diverse shell programs collected from tasks found out in the wild, various real inputs to these programs facilitating small and large deployments, extensive analysis and characterization aiding their understanding, and additional infrastructure and tooling aimed at usability and reproducibility in systems research. The KOALA benchmarks perform a variety of common shell tasks; they combine all major language features of the POSIX shell; they use a variety of POSIX, GNU Coreutils, and third-party components; and they operate on inputs of varying size and composition—available on both permanent archival storage and scalable cloud storage. Applying KOALA to four systems aimed at accelerating shell programs offers a broader perspective on their trade-offs, generalizes their key results, and contributes to a better understanding of these systems.

1 Introduction

Shell programming is as prevalent as ever. It consistently ranks among the top ten programming languages, with a recent popularity increase that dwarfs those of established languages such as C and Python [32]. These trends are mirrored by recent academic activity on the shell aimed at accelerating shell programs through parallelization [50, 72, 85], distribution [66, 75], and other forms of scale-out [34, 52, 54, 74].

Unfortunately, research on the shell is often held back by the absence of an established benchmark suite—a systematic collection of representative, diverse, and well-studied programs released as a reusable artifact. Consider the difficulties that the Shark authors faced in evaluating the performance benefits of its program transformations [14]:

> *To our knowledge, there does not exist a set of shell language benchmarks. We present preliminary results on a handful of microbenchmarks that we wrote ourselves.*

The absence of such a suite decelerates research on the shell, as authors waste time searching for new benchmarks. It also eschews core scientific principles, such as replicability and reproducibility, and hinders fair comparison, as different systems are compared against different baselines. It creates additional and unnecessary work, as it forces researchers to hand-roll their own benchmarks. Finally, it limits the impact of otherwise sound and widely applicable techniques, due to the lack of supporting evidence.

The KOALA benchmarks: This paper presents KOALA, a benchmark suite aimed at performance-oriented research targeting the Unix and Linux shell. KOALA combines a collection of diverse, real-world, POSIX shell programs; realistic inputs of varying size; extensive analysis and characterization of these benchmarks; and additional infrastructure for packaging, automation, and reporting aimed at reusability and reproducibility. Its goal is to enable and support research on shell-script performance optimization, as well as broader systems research within the context of the shell—*e.g.*, CI/CD, analytics, automation, *etc.* (see §2–3).

The KOALA benchmarks are sourced from multiple time periods and perform a wide variety of tasks typically found in the shell—including log analysis, data processing, system administration, bioinformatics, software building, and continuous integration and deployment. They combine all major language features of the POSIX shell; use a variety of POSIX, GNU Coreutils, and third-party commands; and operate on inputs of varying size and composition—available via both permanent archival storage and scalable cloud storage for rapid access. Additional automation aims at setting up KOALA across a variety of environments, confirming input and output correctness, and reporting on several execution characteristics. This additional automation is structured to maximize usability, configurability, replicability, and reproducibility.

Extensive analysis of the KOALA suite reveals that it covers the full range of features of the POSIX shell; represents these features in proportions that align with real-world shell scripts; captures a wide range of script sizes, complexities, and styles—reflecting the shell's diverse use cases; and exhibits a variety

of runtime characteristics—from compute- to memory- to I/O-intensive and short-lived to long-running computations.

Contributions: In summary, this paper contributes:

- **A collection of benchmark programs:** A systematic and diverse collection of real-world programs representative of tasks commonly found in the context of the shell and combining a variety of computational domains, language features, and shell components.
- **Permanent and scalable storage of varying inputs:** A set of real-world inputs stressing these benchmarks as well as smaller inputs aimed at immediate results, both stored on highly available and scalable cloud storage backed up by permanently available archival storage.
- **Tooling and automation for reuse:** Additional infrastructure offering tooling, automation, and configurability for executing these benchmarks on a variety of environments, confirming input and output correctness, and reporting on their observed characteristics.
- **Characterization and implications:** Extensive analysis and characterization of these benchmark programs and their inputs over several dimensions, and application of the entire suite on four prior optimization systems and tools targeting the shell.

Availability: The full set of benchmarks, their inputs, and infrastructure for automation, containerization, and reporting are all available as an MIT-licensed open-source repository.

https://kben.sh

2 Example & Motivation

KOALA targets the systematic evaluation of shell-related systems and tools—for example: Shark [14], a system for accelerating shell script execution using syntactic transformations; GNU `parallel` [59], a command-line utility for parallelizing shell pipelines; *hS* [52], a system speculatively re-ordering the execution of shell programs; and PASH [50], a just-in-time shell-script parallelization system.

Available options and the pains of characterization: The authors of these systems currently have a limited set of options for characterizing their performance benefits.

Microbenchmarks—small, isolated code fragments—are necessary for zooming into key trade-offs, stressing particular behaviors, and highlighting system limits—and are often handwritten to meet these goals [14, 48, 50, 75, 85]. Such synthetic workloads differ significantly from ones seen in practice. Microbenchmarks don't admit generalizable conclusions or holistic evaluations of complex systems.

Standards tests [33, 78] offer a thorough evaluation of shell behavior—*e.g.*, that pipelines take precedence over redirections in their constituent commands. The POSIX test suite is one example [78]. They focus on behavioral equivalence against a specification, not judging the performance of real workloads. Lacking the size and features of workloads seen in practice, they offer little insight into how optimization systems affect real programs.

Open-source code repositories are ripe targets for longitudinal studies—*e.g.*, understanding a community's use of certain shell features [22, 71]. Leaving scraping and parsing challenges aside, these programs typically lack inputs, setup scripts, explicit dependency declaration, and portable constructs—often consisting of noisy, low-quality, or even incomplete programs. Ad hoc collections are also not well understood by the community or the authors themselves, who risk accidentally skewing results due to these challenges.

User or developer study corpora are primarily focused on understanding developer patterns [21, 69]. Due to time constraints and the need to control for confounding factors, user study corpora broadly optimize for program characteristics that do not necessarily support generalization to the diversity of size, shape, and complexity of real-world programs.

Requirements and goals: The aforementioned options available to the authors of systems such as Shark, `parallel`, *hS*, and PASH are ill-suited for performance characterization.

For a set of benchmarks to support research on the shell and related systems, it needs to meet the following criteria: (1) real-world programs performing real computations on real inputs; (2) diversity in workloads, domains, shell features, and commands used; (3) automated setup, analysis, validation, and reporting; (4) community-wide understanding through extensive analysis and characterization; (5) permanent and scalable availability, allowing anyone to download and use them. KOALA meets all of these criteria.

Using KOALA: The authors of these systems enjoy a variety of options for downloading and setting up KOALA:

```
curl -s up.kben.sh | sh &&
  ./koala/main.sh --min --bare
```

This option sets up platform-specific dependencies, then downloads or generates a minimal set of inputs (§3). It then executes the full suite using its default parameters, such as the default shell interpreter—except for (1) `--min`, which uses minimal inputs (just enough to exercise the entire suite, all output validators, and its reporting infrastructure), and (2) `--bare`, which executes in the current environment instead of a Docker container launched afresh for the execution of each benchmark. KOALA then confirms output correctness, guarding against failures and cross-run interference, and generates a report of the entire execution. On a fresh AWS environment, on a `t3.large` instance, such a minimal bare-metal setup, execution, and reporting completes in about 20 minutes.

Omitting the `--bare` launches KOALA in a Docker container, avoiding cross-run contamination and permanent effects on the host environment beyond result creation (*e.g.*,

dependency installation), and executes KOALA on full-sized inputs, revealing how a system under evaluation is affected by realistic workloads. KOALA aggregates the results over multiple runs (five by default), increasing statistical confidence, and reports the results of the execution. Using the default parameters in the same AWS environment and full inputs bumps execution to about 24 hours.

An example benchmark: To understand how Shark, `parallel`, *hS*, and PASH would benefit from KOALA, consider zooming into a single benchmark. Fig. 1 provides a real-world program (simplified for this discussion) iterating over `start` and `end` years to compute maximum, minimum, and average temperatures from a large dataset (*c.f.*, Tab. 1). The body of the loop consists of one pipeline per metric, each using `cat` to read the input data and `grep` to filter out sentinel values, then applying operation-specific computations. The `parallel` and Shark authors would need to apply manual rewrites for their systems; the rest of the authors could point `KOALA_SHELL` to their system (§4). They would then execute this benchmark on `small` (but, contrary to `min`, *real*) inputs:

```
./main.sh --small weather
```

Each of these systems accelerates **`weather`** differently. Shark eliminates `cat`s at the start of the pipelines by moving inputs to `cut`, and executes the three pipelines in parallel. GNU `parallel` can be applied to both the outer loop and the three pipelines; the user would need to choose which parts of the script to parallelize. *hS* runs commands speculatively, unrolling loop iterations to execute in parallel. And PASH parallelizes each pipeline stage. The four systems exploit different opportunities, optimizing different shell constructs and leaving different parts of each script unchanged.

On commodity infrastructure (§7), speedups average 2.3× for Shark, 2.7× for `parallel`, 1.64× for *hS*, and 1.08× for PASH. Most importantly, in contrast to similar numbers potentially collected over microbenchmarks, the composition and complexity of these benchmarks demonstrate the anticipated benefits—as well as potential limitations—of these systems in real-world settings.

3 KOALA Design Overview

Tab. 1 summarizes KOALA's full set of benchmarks and their characteristics. KOALA offers 126 programs, analogous to but diverse from the earlier **`weather`** example (§2), grouped into sets that share features, sources, or inputs. This section focuses on the first three sets of columns: Identification characteristics (Col. 1) list each benchmark set's name and (a subset of) the programs it contains; Descriptions (Cols. 2–4) summarize application domains ($\mathcal{D}$), the number of programs in the set (#.sh), and the total lines of code (LoC); and Inputs (Col. 5) summarize the size of each input. Later sections discuss other characteristics.

```
#!/bin/bash
d="./data/temperatures";
for y in $(seq $start $end); do
  cat $d/$y | cut -c 89-92 | grep -v 999 |
    sort -rn | head -n1 > max.$y
  cat $d/$y | cut -c 89-92 | grep -v 999 |
    sort -n | head -n1 > min.$y
  cat $d/$y | cut -c 89-92 | grep -v 999 |
    awk "{t += \$1; i++} END {print t/i}" > avg.$y
done
```

Fig. 1: Temperature analytics script. The script calculates temperature `max`, `min`, and `avg` across input files.

Inputs: Before discussing specific programs and inputs, it is worth outlining the general structure of inputs, its goals, and the infrastructure deployed to ensure scalable and permanent storage. These are guided by the experience of several users [9, 29, 72, 85] over multiple years.

KOALA comes with three input sizes for each of its benchmarks. Full inputs (`--full`) are either the original inputs programs were designed to operate on (*e.g.*, **`weather`**, **`covid`**) or, for older programs, realistic large-scale inputs on which these programs would operate today (*e.g.*, **`oneliners`**, **`unixfun`**). Small inputs (`--small`) are subsets of the full inputs useful for smaller-scale characterizations. They range between 0.1–30% of the original input (with some exceptions, noted below) and are carefully truncated to still be meaningful and semantically valid—for example, genomics pipelines still operate on valid genome sequences seen in practice, rather than arbitrary strings terminated at arbitrary points. Minimal inputs (`--min`) are synthetic data created to quickly confirm correct setup and facilitate further automated configuration. These inputs aim at near-immediate results and, depending on the characteristics of the system and environment, are either downloaded or generated. Downloading, detailed below, fetches inputs from cloud servers when high-bandwidth connectivity is available; generation operates repeatedly on benchmark-specific seeds, useful for environments with limited connectivity.

To meet key performance and availability requirements, inputs are hosted on infrastructure that combines two replication tiers—in addition to their source, *e.g.*, NOAA [2] or Wikipedia dumps [87]. The primary tier operates as KOALA's default configuration and stores all inputs on a replicated storage cluster managed by Brown University. The cluster is connected via 1Gb/s switched Ethernet network with access to Internet1 and Internet2 via Brown's fiber-optic backbone, aimed at high (but not permanent) availability—ensuring that inputs are available for concurrent users of the KOALA benchmarks. Additionally, infrastructure managed by Stevens Institute of Technology and UCLA hosts replicas of all inputs to offer additional availability.

The secondary tier ensures permanently available archival storage. Inputs (and the programs using them) are made available on Zenodo servers, split appropriately to comply with

Tab. 1: KOALA benchmark summary. The table summarizes all benchmarks in the KOALA suite. Col. 1: Identification characteristics, listing each benchmark set's name and (a subset of) the programs it contains. Cols. 2–4: Descriptions, summarizing application domains ($\mathcal{D}$), the number of programs in the set (#.sh), and the total lines of code (LoC). Col. 5: Inputs, summarizing the size each benchmark's inputs. Cols. 6–7: Static features, summarizing syntactic characteristics—*i.e.*, the number of shell constructs (#Cons) and the number of distinct commands (#Cmd). Cols. 8–11: Dynamic features, summarizing the execution characteristics—*i.e.*, time spent on shell evaluation (t_S), time spent on commands (t_C), memory consumption (Mem), and total input-output (I/O). Cols. 12–13: System, summarizing the number of system calls (#SC) and open file descriptors (#FD). Col. 14: Source, containing a reference to the source of the benchmark.

Benchmark/Script	Surface			Inputs		Syntax		Dynamic				System		Source
	$\mathcal{D}$	#.sh	LoC	Small	Full	#Cons	#Cmd	t_S	t_C	Mem	I/O	#SC	#FD	
analytics	SA/DA	4	53	1.94GB	78.9GB	10	21	10ms	84s	334MB	23.0GB	117.3m	84	[6, 66, 74]
`nginx.sh`			19			10	13	~0	1s	10.3MB	1.79GB			
...														
bio	DA/B	4	823	24.3GB	114GB	17	66	51s	6720s	25.1GB	352GB	35.3m	79	[17, 38, 41]
`data.sh`			226			13	44	46s	3521s	25.1GB	287GB			
...														
ci-cd	CI/BS	21	2592	N/A		20	134	30ms	128s	461MB	2.35GB	3.5m	885	[18, 64]
...														
covid	DA/DE	5	53	3.34GB	5.08GB	5	6	~0	67s	1011MB	80.6GB	14.3m	150	[81]
`1.sh`			9			5	6	~0	12s	13.2MB	17.5GB			
...														
file-mod	AN/MI	5	41	4.35GB	39.2GB	11	10	99ms	235s	164MB	13.9GB	1.5m	61	[66, 71, 74]
`encrypt.sh`			11			10	6	~0	2s	2.82MB	5.10GB			
`img-conv.sh`			11			11	6	99ms	170s	145MB	3.83GB			
...														
inference	ML/DA	3	61	3.85GB	11.7GB	15	27	40ms	1586s	7.16GB	83.7GB	37.9m	81	[27, 43]
...														
ml	ML/DA	1	47	4.71GB	15.0GB	7	1	~0	1156s	7.87GB	41.6GB	14.9m	10	[63]
nlp	TP/ML	23	303	3k bks	115.9k bks	10	19	15s	851s	9.62MB	272GB	178.6m	526	[45]
`bigrams.sh`			19			10	16	710ms	50s	7.93MB	22.5GB			
...														
oneliners	AN/TP	13	119	202MB	13.5GB	10	24	60ms	14s	199MB	3.77GB	4.5m	288	
`spell.sh`			11			6	7	~0	1s	8.38MB	523MB			[12]
`top-n.sh`			2			5	6	~0	960ms	8.25MB	408MB			[13]
`uniq-ips.sh`			2			4	3	~0	60ms	8.15MB	54.2MB			[55]
...														[67, 76]
pkg	CI/AN	2	43	110 pkgs	2.0k pkgs	16	22	5s	201s	572MB	15.3GB	35.2m	132	[10, 86]
`pacaur.sh`			29			14	19	5s	81s	490MB	14.6GB			
`proginf.sh`			14			11	6	10ms	120s	572MB	709MB			
repl	SA/MI	3	586	N/A		20	55	~0	24s	197MB	1.21GB	5.6m	79	[39, 66]
...														
unixfun	MI/TP	36	70	599MB	59.1GB	4	12	~0	5s	9.80MB	5.71GB	944.0k	733	[62]
`1.sh`			2			3	2	~0	50ms	680KB	108MB			
`2.sh`			2			3	3	~0	260ms	7.51MB	209MB			
...														
weather	DA/DE	2	74	893MB	146GB	11	20	260ms	958s	94.2MB	39.1GB	56.7m	50	[80]
...														
web-search	MI/TP	4	34	833MB	8.61GB	14	30	11s	1343s	1.32GB	17.1GB	112.6m	174	[16]
...														
Min		1	34	1.05MB	44.9MB	4	1	0	5.4	9.62MB	1.21GB	944.0k	10	
Mean		9	349.9	3.66GB	38.0GB	12.1	31.9	6.1	955.7	3.17GB	67.9GB	44.2m	238	
Median		4	65.5	1.54GB	13.5GB	11	21.5	0.1	218.2	398MB	20.1GB	25.0m	108	
Max		36	2592	24.3GB	146GB	20	134	52	6720.9	25.1GB	352GB	178.6m	885	

Zenodo's 50GB limit. While it does not offer the same bandwidth as the primary tier, it forms a transition path if input reconstruction becomes necessary—in the unlikely case that all university replicas become permanently unavailable.

Computational domains: KOALA draws from several domains, representing a broad range of computations—including both classic workloads typical of shell programs and modern workloads found pervasively today.

Data analytics programs (DA, 11 programs across three sets) extract, transform, and summarize quantitative datasets such as temperature records, public transit data, and genomic information. System administration programs (SA, seven programs across two sets) include typical system setup and maintenance tasks or system log manipulation. Continuous integration and deployment workflows (CI, 23 programs across two sets) include standalone shell programs or ones that automate software builds, program analysis, and software testing. Machine-learning programs (ML, 27 programs across three sets) include scripts either implementing learning tasks or gluing together third-party components in modeling pipelines. One-off automation scripts (AN, 18 programs across two sets) automate various operations such as file encryption, media conversion and one-off tasks such as spell-checking and content filtering. Other miscellaneous benchmarks (MI, 40 programs across two sets) consist of scripts that do not belong to any particular domain and perform a variety of tasks such as arbitrary text transformations, and web crawling.

Across and orthogonal to these domains, KOALA combines several diverse styles of computations and tasks (after the slash in column $\mathcal{D}$). For example, two data-extraction sets (DE) summarize information from large data sources; four text-processing sets (TP) manipulate text in multi-stage transformation pipelines; and three automation sets (AN) revolve around sequencing and managing task execution.

Benchmark programs: KOALA consists of fourteen sets of programs, presented in alphabetical order.

The **`analytics`** set contains four programs that analyze log files to extract key events [66, 74]. Operations include filtering and summarization. They operate on 78.9GB of line-oriented logs—including TCP traffic, Nginx access logs, and ZMap scan data—all collected from real network traces, as well as logs from a ray-tracing system [23, 26, 70, 82]. The small inputs (1.94GB) include truncated versions of the same log and packet-capture files.

The **`bio`** set consists of four programs for processing genomic and transcriptomic data. One performs population genomics analysis [17, 41], and the other three implement key stages of the TERA-Seq platform [38] for processing and aligning RNA sequences. Inputs include a BAM genome sequencing file [79] and auxiliary data such as gene annotations, totaling 114GB. The scripts exhibit fan-out/fan-in structures, offer opportunities for code de-duplication, implement work-queue-like parallelism, and operate on large files. The small inputs (24.3GB) omit much of the optional auxiliary data.

The **`ci-cd`** set contains 21 build-related programs, including scripts for building eight applications—such as Lua, Memcached, Redis, and SQLite [18]—as well as the `makeself` utility, which creates self-extracting archives [64]. These programs are typically used in continuous integration and deployment pipelines and feature multiple dependencies, but typically have small—if any—data inputs, thus using identical full and small inputs. The `makeself` script executes the program's test suite and requires no inputs.

The **`covid`** set contains five programs that calculate statistics about the public transit activity of a large city during the COVID-19 pandemic [81]. These programs come in two versions, amenable to different optimization strategies: a version that uses typical Unix staples such as `cut`, `sort`, and `unix`; and one written as a monolithic `awk` program. They operate on 5.08GB of CSV data about transit vehicle activity, and the small inputs cover a 3.34GB subset.

The **`file-mod`** set consists of five programs that automate file-level transformations, including compression, encryption, and format conversion. Two of the programs operate on 34.7GB of packet capture files collected from publicly available datasets [58], compressing and encrypting them using `openssl` [71]. The other three perform media format conversions [66, 74], reading from and writing to the filesystem in tight loops that process hundreds of image and audio files (4.4GB). The small version of the benchmark uses a reduced dataset with 4.35GB total of text and media files.

The **`inference`** set contains three programs that perform inference tasks on media files using large foundational models [31, 46, 90]. These include image captioning [43], music playlist generation via embedding interpolation [27], and sequential segmentation and classification of hieroglyphic images using SAM [46] and a custom classifier. These benchmarks process 9.3GB of image and audio data. The deep-learning models use total 2.4GB in size. The benchmarks perform model serving using external run-times [4, 61] interfacing with them via custom wrappers [73]. The small version operates on a subset (3.85GB) of the original images and music using the same back-end models.

The **`ml`** set consist of a typical of machine-learning workload comprising multiple stages written using the Scikit-Learn library [63]. It is a decomposition of a monolithic Python program into a series of scripts operating on 15GB of inputs and performing distinct steps for data ingestion, learning, inference, classification, and evaluation. The small version of this benchmark uses a subset of the same input (4.71GB).

The **`nlp`** set contains 23 scripts implementing natural language processing techniques, drawn from "Unix for Poets" NLP tutorial [45]. Most scripts consist of 1–2 lines, and can be combined in sequential operation. The input dataset contains over 115,000 ASCII books from Project Gutenberg [57], and and 3,000 books (1.42GB) for the small inputs.

The **`oneliners`** set contains 11 shell pipelines drawn from various sources across the academic and popular literature [6, 12, 13, 45, 55, 67, 76]. Some of these programs are classics in the Unix literature [12, 13, 67] and highlight the Unix philosophy, embodied by the shell; others are more recent [6, 55, 76], applying the same principles to mod-

ern workloads. They make extensive and, at times, complex use of streaming constructs, applying maps, filters, reductions, stream duplication, and window operators. Their inputs are shared between script subsets and combine books from Project Gutenberg, commands available in `PATH`, and packages from the `apt` repositories (13.5GB full, 202MB small).

The **`pkg`** set consists of two programs: one automates the installation of packages from the Arch User Repository (AUR) [1], and the other applies a static analysis tool to extract permissions from the node package manager (npm) repository [3, 86]. Its input consists of two lists—195 AUR package names and 1,768 npm package names. The benchmark downloads, builds, and installs the AUR packages in a loop, and analyzes the npm packages to infer their permissions. The small version of this benchmark includes the first 10 and 100 packages from each list.

The **`repl`** set contains two standalone programs. The first performs auditing for security vulnerabilities and misconfigurations. The second replays a development workflow on a large Git repository. Both scripts include non-trivial filesystem access patterns, with the second being metadata heavy. Neither script requires external input at runtime (N/A in Tab. 1).

The **`unixfun`** set contains 36 programs solving the Bell Labs 50-year Unix anniversary challenge [62]. They mimic classic Unix text-processing computations, often short ones that eliminate the vast majority of the input using a `head` or `tail`. They operate on inflated inputs (59.1GB) created by duplicating the original inputs (599MB), while maintaining the expected structure.

The **`weather`** set consists of two program phases calculating statistics on historical temperature data from a Hadoop book [80], with the second one performing some additional analysis and recreating Edward Tuft's famous weather diagram [24]. Some of these phases correspond to MapReduce and Spark computations exemplifying large-scale data processing—not part of KOALA, but useful for comparisons of shell-acceleration systems targeting similar or comparable scalability goals. These phases operate on large inputs (146GB) collected from NOAA [2] spanning multiple years. The small version of these inputs corresponds to a subset of temperatures from the year 2015 (893MB).

The **`web-search`** set contains several programs from a course on distributed systems, implementing web crawl, index, and query. It is implemented as a POSIX-shell streaming program, using complex streaming operators—including duplication, shifting, windows, and dataflow cycles. The input dataset consists of a Wikipedia snapshot (8.61GB), or a subset for the small version (833MB).

Principal component analysis: Before diving into various static and dynamic characteristics of the KOALA suite, it is worth getting a sense of its high-level diversity characteristics.

Fig. 2 shows this diversity using Principal Component Analysis (PCA) [5] on two distinct representations of each benchmark detailed below. PCA maps high-dimensional data to a lower-dimensional space that preserves structural differences for easier comparison and visualization. It computes weights for each of the n original dimensions to create k (where $k < n$) new and uncorrelated composite dimensions that are each linear combinations of the original ones.

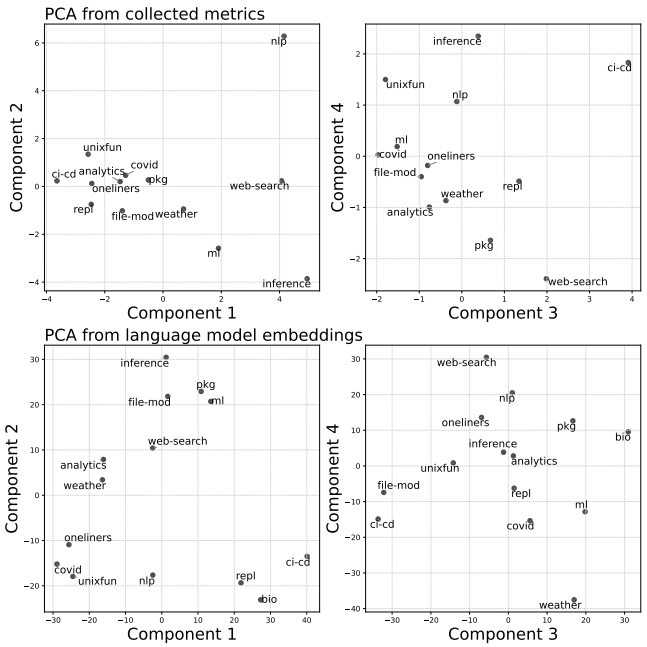


Fig. 2: Principal component analysis results. Top: PCA plot resulting from the static and dynamic analysis of each benchmark. Bottom: PCA plot resulting from embedding calculation of KOALA source code using a pre-trained model [60].

The top row of Fig. 2 visualizes benchmarks projected on this reduced space based on their static and dynamic characteristics (§5–6). The spread of the benchmarks across the principle components suggests that the suite covers a broad and non-overlapping range of syntactic and behavioral profiles. The bottom row of Fig. 2 visualizes each benchmark on the space based on dense vector representations that capture high-level program structure and logic. These embeddings, generated using OpenAI's `text-embedding-3-large` model [60], capture information targeting diversity at the semantic and syntactic level [7, 83]. Well-distributed across the projected space, the results also suggest substantial diversity in both syntax and semantics.

4 Infrastructure and Configuration

Each KOALA benchmark set adheres to a specification that defines installation dependencies, input configuration, benchmark execution, and validation of the final results. This specification, shown in Fig. 3, includes five infrastructure scripts: (1) `install.sh` sets up dependencies required by the benchmark; (2) `fetch.sh` downloads (or generates, §3) and, if nec-

```
sample-benchmark
├── scripts      # Benchmark scripts
│   ├── a.sh
│   ├── b.sh
│   └── ...sh
├── install.sh   # Dependency installation
├── fetch.sh     # Input data download/preparation
├── execute.sh   # Script execution
├── validate.sh  # Hash generation & verification
└── clean.sh     # Input and output file cleanup
```

Fig. 3: Benchmark structure. Each benchmark includes five support scripts (*e.g.*, `install`, `clean`) and a `scripts/` directory containing the source code of each benchmark. A top-level `main` driver executes all support scripts for one or all benchmarks.

essary, pre-processes inputs, accepting an argument that specifies their size; (3) `execute.sh` executes benchmark programs, collecting basic information such as execution time and resources; and (4) `validate.sh` confirms the correctness of the benchmark's output by hashing the output of each script and comparing it to a known-good hash. Finally, (5) `clean.sh` removes inputs, outputs, and temporary files created during the benchmark's execution. These scripts avoid redundant work—*e.g.*, dependency installation or input generation are skipped if possible, except if forced (`-f`).

KOALA also includes a top-level driver (`main.sh`) that executes the five scripts in sequence for any benchmark. The driver checks several configuration parameters, accepted as flags or environment variables, which it then passes to the rest of the scripts. For example, a `KOALA_SHELL` parameter configures the shell interpreter executing the suite—*e.g.*, `bash`, `zsh`, or a path to an executable, and defaulting to `sh`.

KOALA's structure aims to facilitate quick collection of preliminary results, not exhaustive coverage of all possible needs. To accommodate a wide range of systems, it is kept minimal—inviting users to modify it to fit their needs.

Containerization: KOALA provides optional infrastructure for running benchmarks in an isolated environment. It includes a Dockerfile that builds a Debian-based container, installs essential dependencies such as `git` and `sudo`, and fetches the suite. A volume is shared with the host system, simplifying access to inputs and outputs.

It also provides support for dynamically generating container images tailored to each benchmark. These images are self-contained and ephemeral: their Dockerfile contains a `CMD` command executing all infrastructure scripts via `main.sh`.

To avoid problems with permission changes or system-wide modifications, KOALA does not require elevated privileges (except for `fetch.sh` which installs software dependencies) and avoids privileged commands such as `sudo` and `setuid`.

Integration effort: The effort required to add a new benchmark to KOALA depends on its complexity, input size, and correctness constraints. It typically involves five steps: (1) identifying and encoding dependencies in `install.sh`; (2) preparing input datasets in multiple sizes—full, small, and min—via `fetch.sh`; (3) including scripts for automated execution in `execute.sh`; (4) defining output validation logic in `validate.sh`; and (5) specifying cleanup steps in `clean.sh`.

Integrating the current set of benchmarks ranged between 10–80 person-hours per benchmark (Tab. on the right). Small and self-contained benchmarks such as **`oneliners`**, **`unixfun`**, and **`nlp`** each took about 10–12 hours; more complex or data-heavy benchmarks such as **`file-mod`**, **`bio`**, and **`ci-cd`**, and took about 60–80 hours due to their size, dependencies, and nuanced output correctness. Most of our time constructing this release of KOALA was spent in building new logic and adapting datasets, with a significant portion devoted to testing and confirming correctness under various environments and configurations.

Benchmark	$\sim T$(h)
`analytics`	40
`bio`	60
`ci-cd`	75
`covid`	65
`file-mod`	60
`ml`	30
`inference`	35
`nlp`	10
`oneliners`	10
`pkg`	30
`repl`	25
`unixfun`	10
`weather`	30
`web-search`	40
Total	520

5 Syntactic Characterization and Analysis

Contrary to single-language environments, the shell often combines components in multiple languages. KOALA thus distinguishes between the characteristics of (and resources spent on, see §6) the shell portion of each benchmark, versus those of components called into by the shell and which often implement the core of a computation (Tab. 1, cols. 6–7).

Methodology: We use `libdash` [53] to parse and analyze both portions using SMOOSH's abstract syntax definition for the POSIX shell [33]: for the shell portion, we count the total occurrences of every AST node; for the command portion, we analyze only AST nodes counting commands, built-ins, and functions—yielding conservative results that exclude dynamic constructs (*e.g.*, `$CMD arg`) but avoid conflating static structure with repeated runtime executions.

Language features: Fig. 4 summarizes the occurrences of each construct across KOALA. Each cell shows the number of times a syntactic construct (vertical axis) occurs in a benchmark (horizontal axis). AST nodes that overlap with more specific nodes or that do not correspond to linguistic features (*e.g.*, escaped characters) are excluded. Overall, KOALA employs all the syntactic constructs of the POSIX shell, in varied combinations that reflect its various styles of computation.

KOALA covers several key shell characteristics worth mentioning. Most sets employ pipelines, shell constructs of the form `a | b` that pass the output of one command (`a`) as input to the next (`b`). Pipelines are an important shell feature, optimized by several data-parallel systems. Two sets use operators `&` and `wait` for explicit parallelism—a feature relevant to performance-oriented shell systems. Two sets use sub-shells, *e.g.*, `(echo hi)`, and several sets use control

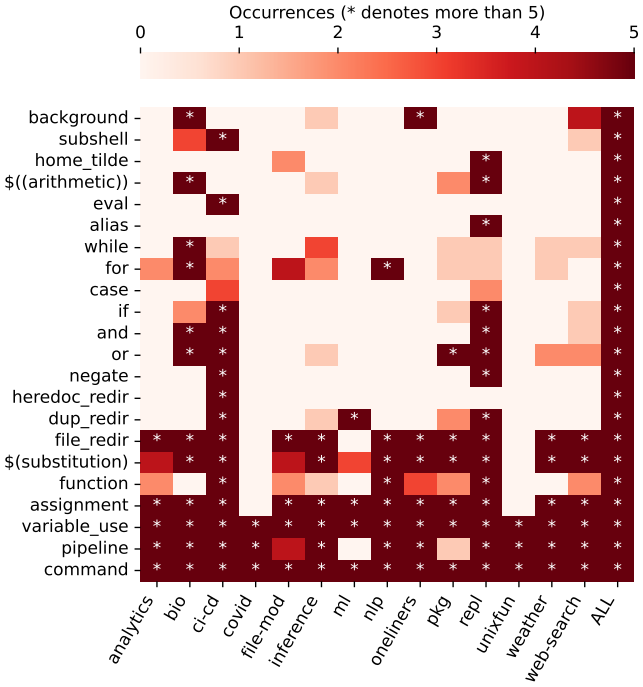


Fig. 4: Syntactic characteristics. Cells show the frequency of the syntactic construct (y-axis) in a benchmark (x-axis), up to five occurrences to maintain clarity of absences (zero counts). Stars denote more than five occurrences.

constructs such as loops and conditionals. Several sets use function definitions, and nearly all use variables, assignments, and stream or file redirections; other kinds of redirections occur less frequently. Many sets use command substitution, *e.g.*, `echo hi $(whoami)`.

Fig. 4 also highlights a key KOALA characteristic: no two benchmarks exhibit the same distribution of syntactic constructs. KOALA represents well the shell's most interesting features—*e.g.*, external command invocations, pipelines, the background operator, subshells, expansion, and redirection. In combination, these features are essential in making the shell the uniquely powerful and complex platform that it is. Different styles of scripts employ these features in different combinations and proportions—and KOALA aims to capture this variation. The proportion of these features align with shell usage studies [22], indicating that the distribution of KOALA's shell features corresponds to current trends and is representative of the programs found in the real world.

Component features: Fig. 5 shows the frequency of all KOALA commands occurring at least 30 times, excluding benchmark-specific functions and binaries. KOALA contains many of the most common commands found in the wild: `echo` tops the list with 441 occurrences; `cat`, `grep`, `sort`, and `tr` count about half of that; other commands exhibit lower frequencies. These results broadly match results from studies of commands in the wild [22], except for path-manipulation commands as most KOALA paths are static.

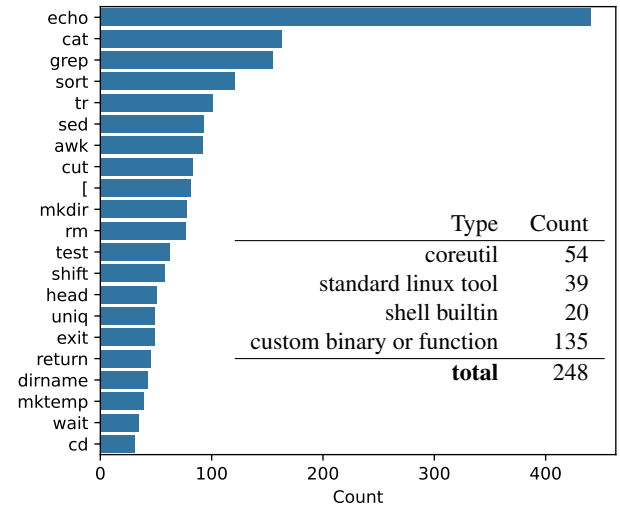


Fig. 5: Command occurrences. The chart shows the frequency of all KOALA commands occurring at least 30 times. The table categorizes the unique commands counted across the entire suite.

The table also groups KOALA's distinct commands into four classes. Most common are GNU Coreutils (*e.g.*, `echo` and `cat`), shell built-ins (*e.g.*, `cd` and `return`), and standard linux tools (*e.g.*, `grep`, `sed`, and `awk`). While the first three classes make up KOALA's vast majority of commands, in terms of frequency, custom binaries and functions account for 134 commands, or 54% of the unique command set.

6 Dynamic Characterization and Analysis

Beyond its syntactic diversity, KOALA also exhibits diverse dynamic characteristics. Exploring this diversity involves executing benchmarks to extract dynamic features such as CPU time, memory consumption, IO characteristics, and interaction with the broader environment (Tab. 1's cols. 8–13).

Methodology: To extract these features, we collect several metrics while executing benchmarks on a machine with 32GB RAM, an 8-core 3.80GHz Ryzen 7 9700X CPU utilizing hyper-threading, and a 1TB NVMe SSD. The shell interpreter is set as `bash --posix`. We measure CPU time and IO by probing `/proc/<pid>/{stat,io}` after the target process exits; and wall-clock time and memory use by calling Python's `time.perf_counter` and `psutil` at 0.01-second intervals. We aggregate results by set by summing timing and IO statistics for each program in that set. We take the maximum high-water-mark memory usage to compute a single set of results per benchmark.

Overview: Fig. 6 presents results both overall and with respect to input sizes. The plots form two coarse groups (x-axis is benchmarks, always): (1) the top two plots (left and right) show total execution time—left is the ratio of CPU time

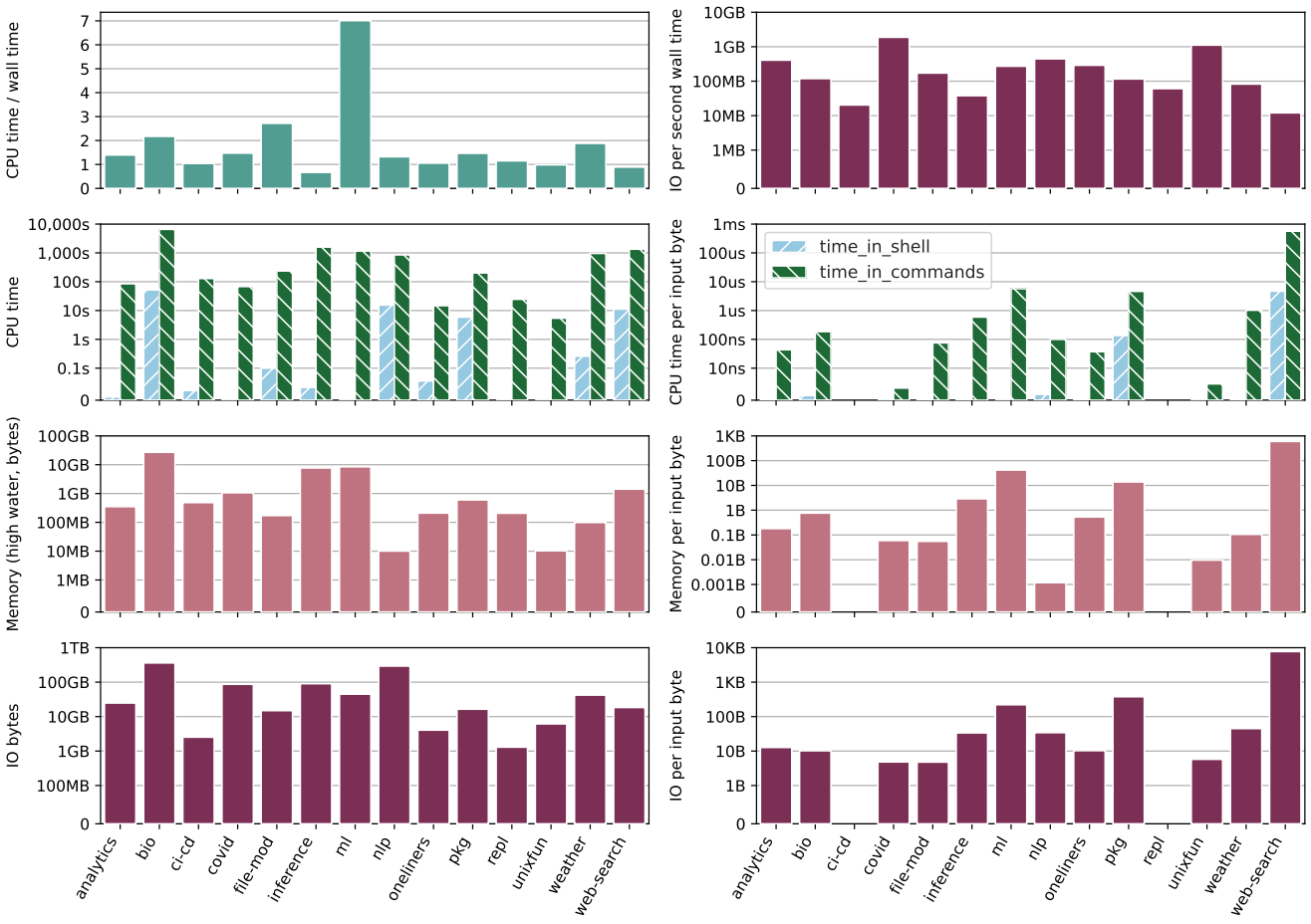


Fig. 6: Dynamic characteristics. The top two plots contextualize the rest: (left) ratio of CPU time to wall-clock time; (right) ratio of IO time to wall-clock time. The bottom three plots (rows 2, 3, 4) show CPU time, Memory, and IO: (left) absolute; (right) normalized over benchmark input. All y-axes except top left are symmetric log scale: values between 0 and the first tick are linearly scaled, and the next are logarithmic.

to wall-clock time, right is the total IO per second of wall-clock; (2) the next six plots (rows 2, 3, 4), show CPU time, high-water-mark memory, and IO—left is absolute, right is per input byte.

Absolute characteristics: As these characteristics vary by orders of magnitude, these plots use a symmetric log scale: values between zero and the first tick are linearly scaled, while the rest are logarithmic. The CPU-time plot (Fig. 6, row 2, left) shows that the benchmark runtime varies from seconds (*e.g.*, **`unixfun`**) to hours (*e.g.*, **`bio`**). CPU time is split between time spent in the shell versus commands: here, too, KOALA demonstrates significant diversity—ranging from a mix of shell execution and commands (*e.g.*, **`nlp`**) to command-dominated workloads (*e.g.*, **`bio`** and **`inference`**).

Similarly, the memory plot (Fig. 6, row 3, left) shows the diversity of memory-intensiveness, varying between 668KB (*e.g.*, **`unixfun`**) and around 10GB (*e.g.*, **`bio`**, **`inference`**, and **`ml`**). Likewise, the IO plot (row 4, left) shows that KOALA exhibits varying degrees of IO-heaviness, ranging from just 75.9MB of IO (*e.g.*, **`ci-cd`**) to 336GB (*e.g.*, **`bio`**).

Tab. 1's last few rows (pg. 4) offer additional context: CPU time, memory consumption, and IO range respectively between 2.1–6544.8s (shown separately for the shell and commands, average: 934.1s), 6.1MB–25.1GB (average: 3.14GB), and 85.7MB–336GB (average: 66.6GB).

Normalized characteristics: Normalized plots (rows 2–4, right) offer a sense of these characteristics normalized by each benchmark's input size, important due to their correlation with input size, which varies widely (0–146GB). This normalization is noticeable with **`bio`** and **`covid`**: these benchmarks have substantial absolute CPU times (on left), but their normalized CPU times (on right) are moderate—reflecting the fact that they are long-running mainly due to the volume of data they process, not the intensity of their computations. On the other hand, **`pkg`** is computationally intense, despite its middling absolute execution time. These characteristics do

not apply to benchmarks with no inputs (*e.g.*, **`ci-cd`** and **`repl`**), which have no bars in the normalized plots.

Overall, across the three dimensions (CPU, row 2; memory, row 3; and IO, row 4), KOALA enjoys significant diversity—multiple metrics vary by several orders of magnitude.

7 Applying KOALA to Optimization Systems

This section summarizes the process and results of applying four optimization systems (§2) on the KOALA benchmarks: Shark [14], GNU `parallel` [59], *hS* [52], and PASH [85] achieve performance gains on different subsets of KOALA.

Hardware & software setup: All experiments in this section were executed on an AWS `c6i.4xlarge` instance with 32GB of RAM and a 16-core 3.5 GHz CPU running Ubuntu 24.04.1 and `bash` v5.2.21 with `-posix` enabled.

Adoption effort: The four systems require different effort to use KOALA (see Appendix II), depending on their needs (*e.g.*, automation, inputs), characteristics, and goals (§2). GNU `parallel` required modifying benchmarks to export parallelizable pipelines and **`for`**-loops as functions passed to `parallel`. Shark required similar transformations guided by its optimizations. PASH and *hS*, both operating as drop-in shell replacements, could simply set `KOALA_SHELL`.

7.1 Shark

To understand whether KOALA aids the characterization of Shark [14], we manually apply its transformations as described in the Shark paper across several KOALA programs.

Methodology: The Shark paper [14] describes several potential syntactic transformations, which we apply manually across all KOALA benchmarks. We then execute the original and optimized versions to characterize the performance of the modified benchmarks.

Results: Shark achieves significant performance gains across KOALA, ranging between 1.01–13.43×, depending on benchmark characteristics. Benchmarks that involve trivially parallelizable loop iterations see major speedups—*e.g.*, Shark improves the **`nlp`** and **`weather`** benchmarks by 6.46× and 13.43×, respectively. Scripts with sequential operations, such as those found in **`ci-cd`**, exhibit less pronounced improvements—at times only marginal gains of 1.16×.

Shark accelerates most benchmarks, though benchmarks with highly interdependent operations that already use pipelines see more limited gains. For example, **`covid`**, **`oneliners`**, **`bio`**, and **`unixfun`** offer less opportunities for Shark-style optimizations, which result in speedups between 1.01–1.06×. The **`web-index`** benchmark sees the least improvement (1.01×) as it already runs the three n-gram calculations in parallel, making Shark obsolete. Shark's optimizations centered around command invocation, do not result in significant performance benefits, speeding up scripts where only such optimization opportunities were present (*e.g.*, web-search) by 1.01× on average. In contrast, optimizations that eliminate temporary files used for intermediate storage can offer more substantial speedups, but they also introduce significant complexity, as pipes require more careful coordination between producers and consumers.

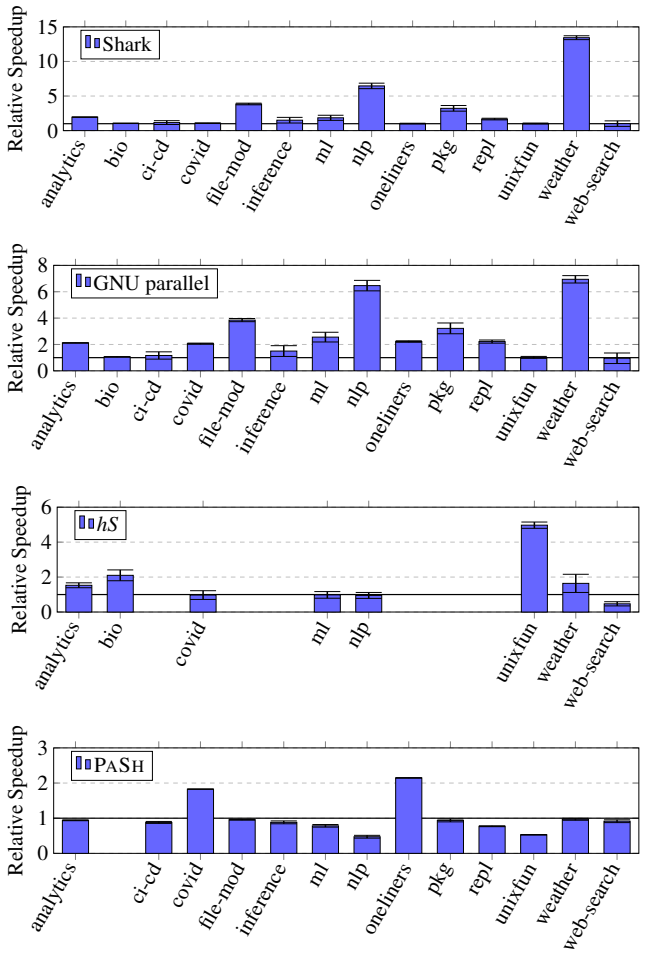


Fig. 7: Shark, GNU `parallel`, *hS*, and PASH speedups. Relative average speedup on the KOALA benchmarks. Baseline is the original runtime in a single-threaded `bash` shell with the `--posix` flag enabled.

Takeaway: KOALA confirms that Shark's optimizations are effective in scripts that involve operations on multiple inputs and independent commands, and less effective for I/O-heavy scripts or scripts that are not easily parallelizable. Its parallelization and pipeline optimizations offer significant speedups, up to 13.43×.

7.2 GNU `parallel`

To understand whether KOALA aids the characterization of GNU `parallel` [59], we apply it on several KOALA programs focusing on natural candidates for such parallel execution.

Methodology: We first identify parallelizable regions within each KOALA program and rewrite them to invoke `parallel`. We aimed for a reasonable use of `parallel`, modifying only parallelizable pipelines that each take under an hour to identify and rewrite—typically ones with (1) independently processed input files, and (2) efficient segment-based processing that requires minimal or no synchronization and aggregation. The latter often takes advantage of GNU `parallel`'s `--pipe` to parallelize input stream processing.

Results: GNU `parallel` achieves substantial speedups in benchmarks that involve I/O-bound operations or are trivially parallelizable. For instance, **`file-mod`** involves converting multiple media files concurrently, resulting in GNU `parallel` speeding it up by 3.84× and fully utilizing all available CPU cores. Similarly, **`nlp`** processes multiple data files independently, resulting in a speedup of 6.46×. GNU `parallel` speedups are less pronounced for scripts that do not have these features, such as those in **`ci-cd`** and **`repl`** that result in 1.16× and 0.95× respectively. These additionally either already use parallelism internally in their command invocations, compiling multiple files or include commands that do not benefit from `parallel`'s parallelization model, such as `git` or `find`.

A few cases do not yield speedup. For example, the **`unixfun`** benchmark involves pipeline stages dependent on the output of previous stages, limiting `parallel`'s approach: the interdependent nature of these stages prevented GNU `parallel` from fully exploiting parallelism, resulting an average speedup of 1.02×.

Takeaway: KOALA confirms that GNU `parallel` can accelerate I/O-bound tasks and loops with no interdependencies—*e.g.*, **`file-mod`**—achieving average speedups of 2.6×. Limited acceleration comes from scripts with sequential operations or ones that require sophisticated splitting and aggregation (*e.g.*, **`unixfun`** and **`web-search`**).

7.3 hS

To assess whether KOALA aids the characterization of *hS* [52], we apply it on KOALA focusing on programs likely to benefit from out-of-order execution.

Methodology: As *hS* is currently under development, we use an early-stage *hS* prototype shared by its authors, configured via the `KOALA_SHELL` variable.

Not all benchmarks are executable by the current version of *hS*. For the following benchmark sets, *hS* succeeded on only a subset of the scripts: **`analytics`**, **`bio`**, **`ml`**, **`nlp`**, **`unixfun`**, **`weather`**, and **`web-search`**. The following benchmark sets as a whole either fail or produce incorrect results with *hS*, and are thus omitted entirely: **`ci-cd`**, **`file-mod`**, **`llm`**, **`nlp`**, **`oneliners`**, **`pkg`**, and **`repl`**. In addition, *hS* was unable to run the `ray-tracing` script in **`analytics`**.

Results: *hS* achieves significant speedups on scripts that include syntactic regions that have no dependencies between them. Speedups range between 1.5–4.97×, with **`analytics`** and **`unixfun`** at the two ends of this range. Other benchmarks resulting in significant speedups include **`weather`** and **`bio`**.

Scripts that involve dependencies between stages do not benefit from *hS*. Such slowdowns range from 0.97× in **`covid`** to 0.47× in **`websearch`**, averaging 0.72× across all benchmarks. They can be attributed to constant costs stemming from *hS*'s speculative execution, related to the environment isolation and tracing, which allows *hS* to roll back effects in cases of misprediction.

Takeaway: KOALA reveals that *hS* can provide benefits that are significant, especially when computations feature independent stages. Typical computations, however, offer a mix of out-of-order optimization opportunities, at times masked by overheads in the *hS* prototype.

7.4 PaSh

We apply PASH on KOALA to understand whether it can characterize PASH's ability to accelerate shell programs.

Methodology: We use the latest version of PASH (commit 0d0a563) and set `KOALA_SHELL` to `./pa.sh --width 4`, *i.e.*, configuring the parallelization degree to 4×. We do not replace script fragments via `alias` and **`function`** constructs that can be annotated with additional parallelizability information; this would allow PASH to extract additional speedup but result in significantly more work. PASH fails when executing the **`bio`** benchmark.

Results: PASH's command-aware parallelization strategy achieves significant speedups in scripts with multi-stage pipelines or **`for`**-loops with no data dependencies across iterations. For example, it achieves speedups of 2.14× and 1.82× on **`oneliners`** and **`covid`** respectively.

Naturally, benchmarks or benchmark fragments for which PASH had no annotations, and thus does not parallelize, see no speedups—*e.g.*, **`pkg`**, **`bio`**, and **`file-mod`**. Additionally, PASH does not speed up scripts that include no syntactic constructs it can parallelize—*e.g.*, **`repl`** and **`ci-cd`** benchmarks, which do not include pipelines or parallelizable **`for`**-loops.

Takeaway: KOALA reveals that PASH can deliver substantial speedups for scripts that fall within its parallelization domain. However, its effectiveness depends on the availability of command annotations and the amenability of programs to the constructs PASH can operate on.

8 Related Work

Benchmark suites: Progress in research depends on apples-to-apples comparisons—and in computer science, that often

means open and reusable benchmark suites. Widely cited benchmark suites in memory-managed environments [15], database transactions [65], parallel processing [19], other areas [20, 36, 42] offer a good examples of their widespread applicability and use. Similar to these suites, KOALA comes with additional support and is expected to release improvements every few years; different from them, it targets performance-oriented systems for the shell—an area that has not yet enjoyed the existence of a systematic benchmark suite.

The DaCapo [15] and Gabriel [28] benchmarks offer particularly good parallels, as they focus on programming environments that did not have a benchmark suite before their release—like these benchmarks, KOALA fills a gap.

Shell studies and datasets: Recent studies collect shell scripts or fragments of scripts—typically in Bash—to analyze their source, understand their properties, and extract key insights. Examples of such studies include the use of Bash in the wild [22], the characteristics of build scripts in Linux distributions [40], and the use of aliases and shell customization in the wild [71], cybersecurity training [84], and interactive coding [89]. These studies do not aim at (creating collections for) evaluating performance optimizations. In contrast, KOALA aims at help characterizing systems and tools that optimize shell programs. KOALA offers curated end-to-end programs with known inputs, dependencies, and behaviors that stress both the shell and the underlying commands.

Shell test and correctness suites: A few test suites focus on evaluating the correctness of shell implementations and their conformance to certain standards. The POSIX test suite [78] is a key resource for validating compliance of a shell implementation with the POSIX standard. The *Smoosh* test suite [33] refines and complements the POSIX test suite. The Modernish shell library/polyfill has a sophisticated diagnostic routine that amounts to shell tests [56]. There are also a variety of testing frameworks designed for automated testing in and of shell scripts [11, 44, 49, 77]. All of these suites focus on testing functionality rather than characterizing performance, and are thus distinct from and complementary to KOALA.

Shell microbenchmarks: Several benchmark suites target the evaluation of specific shell-language constructs via microbenchmarks. `ShellBench` [48] provides a collection of small scripts (ranging from 8–95 lines of code) designed to stress individual shell features—*e.g.*, variable substitution, expansion, and subshell creation. Similarly, `zsh-bench` [68], the Oils benchmarks [8], and UnixBench [47] focus on isolated performance characteristics—*e.g.*, interactive shell behavior or command invocation times. In contrast, KOALA offers larger, more diverse, whole-program benchmarks (95 programs, 17–2428 lines each) that perform end-to-end computations, operate over large datasets, and involve substantial work outside the shell interpreter itself.

Performance properties and characterization: Prior research on benchmark sets often discusses language-specific properties about the programs involved—for example, the impact of garbage collection [15, 28, 51] or the structural features in object-oriented benchmark suites [15, 37, 51]. While important in other domains, these characteristics have no direct equivalent in shell programs; KOALA instead focuses on the shell as glue, carefully distinguishing between the time spent in the shell interpreter and in third-party commands.

Earlier work also studies the behavior of programs in simulation or on hardware [25, 30, 35]. KOALA's characterization focuses on properties that are independent of any particular hardware or operating system implementation. KOALA will make it easy to use shell programs as workloads in the evaluation of general-purpose computational substrates.

Evaluations of shell programs: A variety of systems from several different authors attempt to operate on, optimize, or accelerate shell programs—*e.g.*, achieving elision [14], parallelization [85], fusion [9], synthesis [72], distribution [66], mobile [88], serverless [54], and syscall refinement [29]. These further demonstrate the acute need for a standardized, usable, and replicable benchmark suite for the shell.

9 Discussion and Conclusion

Benchmark programs are crucial for evaluating ideas, comparing and contrasting approaches, and fueling academic and industrial research. They are especially needed in systems research, where many of the key theses revolve around performance-related arguments and their quantitative evaluation. This need is particularly acute in the context of the shell, where no benchmark suite currently exists.

KOALA is a benchmark suite aimed at the characterization of performance-oriented shell research. It combines a systematic collection of diverse shell programs collected from tasks found in the wild, various real inputs to these programs facilitating small and large deployments, extensive analysis and characterization aiding their understanding, and additional infrastructure and tooling aimed at usability and reproducibility in systems research. Static and dynamic characterization confirms that the KOALA programs perform a variety of tasks commonly performed in the shell; combine all major language features of the POSIX shell; use a variety of POSIX, GNU Coreutils, and third-party components; and operate on inputs of varying size and composition.

Acknowledgements

We thank the anonymous ATC'25 reviewers for their feedback; our shepherd, Maria Carpen-Amarie, for her guidance; the ATC'25 Artifact Evaluation reviewers for their time; and the Brown CS2952R (F'24) participants for their input on several iterations of this paper. This material is based upon research supported by NSF awards CNS-2247687 and CNS-2312346; DARPA contract no. HR001124C0486; a Fall'24

Amazon Research Award; a seed grant from Brown University's Data Science Institute; and a BrownCS Faculty Innovation Award.

References

[1] Arch User Repository (AUR). https://aur.archlinux.org. Accessed: 2025-01-13.

[2] National Oceanic and Atmospheric Administration (NOAA). https://www.noaa.gov. Accessed: 2025-01-13.

[3] npm.js. https://www.npmjs.com. Accessed: 2025-01-13.

[4] Ollama: Run large language models locally. https://ollama.com, 2023. Accessed: 2025-06-02.

[5] Abdi, Hervé and Williams, Lynne J. Principal component analysis. *Wiley interdisciplinary reviews: computational statistics*, 2(4):433–459, 2010.

[6] Adam Drake. Command-line Tools Can Be 235x Faster Than Your Hadoop Cluster. https://adamdrake.com/command-line-tools-can-be-235x-faster-than-your-hadoop-cluster.html, 2014. Accessed: 2025-06-01.

[7] Alina Petukhova and João P. Matos-Carvalho and Nuno Fachada. Text clustering with large language model embeddings. *International Journal of Cognitive Computing in Engineering*, 6:100–108, 2025.

[8] Andy Chu and Contributors. Oils Benchmarks. https://github.com/oils-for-unix/oils/tree/master/benchmarks, 2021. Accessed: 2025-04-28.

[9] Anna Herlihy and Periklis Chrysogelos and Anastasia Ailamaki. Boosting Efficiency of External Pipelines by Blurring Application Boundaries. In *12th Conference on Innovative Data Systems Research, CIDR 2022, Chaminade, CA, USA, January 9-12, 2022*. www.cidrdb.org, 2022.

[10] Armando Cerna. Pacaur building script. https://github.com/armandocerna/dotfiles/blob/master/scripts/pacaur.sh. Accessed: 2025-01-13.

[11] Bats-Core Project. Bats-Core - Bash Automated Testing System. https://bats-core.readthedocs.io/en/stable. Accessed: 2025-01-01.

[12] Bentley, Jon. Programming pearls: a spelling checker. *CACM*, 28(5):456–462, May 1985.

[13] Bentley, Jon and Knuth, Don and McIlroy, Doug. Programming pearls: a literate program. *CACM*, 29(6):471–483, June 1986.

[14] Berger, Emery D. Optimizing Shell Scripting Languages. 2003.

[15] Blackburn, Stephen M. and Garner, Robin and Hoffmann, Chris and Khang, Asjad M. and McKinley, Kathryn S. and Bentzur, Rotem and Diwan , Amer and Feinberg, Daniel and Frampton, Daniel and Guyer, Samuel Z. and Hirzel, Martin and Hosking, Antony and Jump, Maria and Lee, Han and Moss, J. Eliot B. and Phansalkar, Aashish and Stefanović, Darko and VanDrunen, Thomas and von Dincklage, Daniel and Wiedermann, Ben. The DaCapo benchmarks: java benchmarking development and analysis. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, OOPSLA '06, page 169–190, New York, NY, USA, 2006. Association for Computing Machinery.

[16] Brown University Department of Computer Science. CSCI 1380: Distributed Computer Systems. https://cs.brown.edu/courses/csci1380, 2025. Accessed: 2025-06-04.

[17] Cappellini, Enrico and Welker, Frido and Pandolfi, *et al.* Early Pleistocene enamel proteome from Dmanisi resolves Stephanorhinus phylogeny. *Nature*, 574(7776):103–107, Oct 2019.

[18] Charlie Curtsinger and Daniel W. Barowy. Riker: Always-Correct and Fast Incremental Builds from Simple Specifications. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 885–898, Carlsbad, CA, July 2022. USENIX Association.

[19] Christian Bienia and Sanjeev Kumar and Jaswinder Pal Singh and Kai Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT '08)*, pages 72–81, New York, NY, USA, 2008. Association for Computing Machinery.

[20] Cohen, Gregory and Afshar, Saeed and Tapson, Jonathan and van Schaik, André. EMNIST: Extending MNIST to handwritten letters. In *2017 International Joint Conference on Neural Networks (IJCNN)*, pages 2921–2926, 2017.

[21] Cora Coleman and William G. Griswold and Nick Mitchell. Do Cloud Developers Prefer CLIs or Web Consoles? CLIs Mostly, Though It Varies by Task. 2022.

[22] Dong, Yiwen and Li, Zheyang and Tian, Yongqiang and Sun, Chengnian and Godfrey, Michael W. and Nagappan, Meiyappan. Bash in the Wild: Language Usage, Code Smells, and Bugs . 32(1):1–22.

[23] Durumeric, Zakir and Wustrow, Eric and Halderman, J. Alex. ZMap: fast internet-wide scanning and its security applications. In *Proceedings of the 22nd USENIX Conference on Security*, SEC'13, page 605–620, USA, 2013. USENIX Association.

[24] Edward Tufte. New York City Weather Chart. https://www.edwardtufte.com/notebook/new-york-city-weather-chart, 2004. Accessed: 2025-06-02.

[25] Eeckhout, Lieven and Georges, Andy and De Bosschere, Koen. How java programs interact with virtual machines at the microarchitectural level. *SIGPLAN Not.*, 38(11):169–186, October 2003.

[26] Elias Dabbas. Web Server Access Logs. https://www.kaggle.com/datasets/eliasdabbas/web-server-access-logs, 2020. Accessed June 3, 2025.

[27] Evangelos Lamprou. Foundation Models and Unix. *Paged Out!*, (6):9, March 2025.

[28] Gabriel, Richard P. *Performance and Evaluation of LISP Systems*. The MIT Press, 08 1985.

[29] Gaidis, Alexander J. and Atlidakis, Vaggelis and Kemerlis, Vasileios P. SysXCHG: Refining Privilege with Adaptive System Call Filters. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, CCS '23, page 1964–1978, New York, NY, USA, 2023. Association for Computing Machinery.

[30] Gan, Yu and Zhang, Yanqi and Cheng, Dailun and Shetty, Ankitha and Rathi, Priyal and Katarki, Nayan and Bruno, Ariana and Hu, Justin and Ritchken, Brian and Jackson, Brendon and Hu, Kelvin and Pancholi, Meghna and He, Yuan and Clancy, Brett and Colen, Chris and Wen, Fukang and Leung, Catherine and Wang, Siyuan and Zaruvinsky, Leon and Espinosa, Mateo and Lin, Rick and Liu, Zhongling and Padilla, Jake and Delimitrou, Christina. An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 3–18, New York, NY, USA, 2019. Association for Computing Machinery.

[31] Gemma Team. Gemma 3 Technical Report. 2025.

[32] Github Inc. The top programming languages. https://octoverse.github.com/2022/top-programming-languages, 2022.

[33] Greenberg, Michael and Blatt, Austin J. Executable formal semantics for the POSIX shell. *Proc. ACM Program. Lang.*, 4(POPL), December 2019.

[34] Greenberg, Michael and Kallas, Konstantinos and Vasilakis, Nikos. The Future of the Shell: Unix and Beyond. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '21, pages 240–241, New York, NY, USA, 2021. Association for Computing Machinery.

[35] Hauswirth, Matthias and Diwan, Amer and Sweeney, Peter F. and Mozer, Michael C. Automating vertical profiling. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, page 281–296, New York, NY, USA, 2005. Association for Computing Machinery.

[36] Hazimeh, Ahmad and Herrera, Adrian and Payer, Mathias. Magma: A Ground-Truth Fuzzing Benchmark. *Proc. ACM Meas. Anal. Comput. Syst.*, 4(3), November 2020.

[37] Hölzle, Urs and Ungar, David. Do Object-Oriented Languages Need Special Hardware Support? In *Proceedings of the 9th European Conference on Object-Oriented Programming*, ECOOP '95, page 283–302, Berlin, Heidelberg, 1995. Springer-Verlag.

[38] Ibrahim, Fadhl and Oppelt, Julian and Maragkakis, Manolis and Mourelatos, Zissimos. TERA-Seq: true end-to-end sequencing of native RNA molecules for transcriptome characterization. *Nucleic Acids Research*, 49(20):e115, 2021.

[39] Israel, Abebe. VPS Audit. https://vpsaudit.vernu.dev. Accessed: 2025-01-13.

[40] Jeannerod, Nicolas and Régis-Gianas, Yann and Treinen, Ralf. Having Fun With 31.521 Shell Scripts. April 2017.

[41] Jon Puritz. Bio594: Using genomic techniques to examine the evolution of populations. https://git.io/JY6J7, 2019.

[42] Just, René and Jalali, Darioush and Ernst, Michael D. Defects4J: a database of existing faults to enable controlled testing studies for Java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, page 437–440, New York, NY, USA, 2014. Association for Computing Machinery.

[43] Justine Tunney. Bash One-Liners for LLMs. https://justine.lol/oneliners, 2023. Accessed: 2025-06-01.

[44] Kate Ward. shUnit2 - xUnit unit testing framework for Bourne based shell scripts. https://github.com/kward/shunit2. Accessed: 2025-01-01.

[45] Kenneth Ward Church. Unix for Poets, 1994.

[46] Kirillov, Alexander and Mintun, Eric and Ravi, Nikhila and Mao, Hanzi and Rolland, Chloe and Gustafson, Laura and Xiao, Tete and Whitehead, Spencer and Berg, Alexander C. and Lo, Wan-Yen and Dollár, Piotr and Girshick, Ross. Segment Anything. In *2023 IEEE/CVF International Conference on Computer Vision (ICCV)*, pages 3992–4003, 2023.

[47] Kirk D. Lucas and Contributors. UnixBench: The BYTE UNIX Benchmark Suite. https://github.com/kdlucas/byte-unixbench, 2012. Accessed: 2025-04-28.

[48] Koichi Nakashima. ShellBench: A POSIX Shell Benchmark Suite. https://github.com/shellspec/shellbench, 2021. Accessed: 2025-04-28.

[49] Koichi Nakashima and contributors. ShellSpec - A Full-featured BDD Framework for Shell Scripts. https://shellspec.info. Accessed: 2025-01-01.

[50] Konstantinos Kallas and Tammam Mustafa and Jan Bielak and Dimitris Karnikis and Thurston H.Y. Dang and Michael Greenberg and Nikos Vasilakis. Practically Correct, Just-in-Time Shell Script Parallelization. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 769–785, Carlsbad, CA, July 2022. USENIX Association.

[51] Lengauer, Philipp and Bitto, Verena and Mössenböck, Hanspeter and Weninger, Markus. A Comprehensive Java Benchmark Study on Memory and Garbage Collection Behavior of DaCapo, DaCapo Scala, and SPECjvm2008. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*, ICPE '17, page 3–14, New York, NY, USA, 2017. Association for Computing Machinery.

[52] Liargkovas, Georgios and Kallas, Konstantinos and Greenberg, Michael and Vasilakis, Nikos. Executing Shell Scripts in the Wrong Order, Correctly. In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems*, HOTOS '23, page 103–109, New York, NY, USA, 2023. Association for Computing Machinery.

[53] libdash developers. `libdash`. "https://github.com/binpash/libdash/tree/master".

[54] Mahéo, Aurèle and Sutra, Pierre and Tarrant, Tristan. The serverless shell. In *Proceedings of the 22nd International Middleware Conference: Industrial Track*, Middleware '21, page 9–15, New York, NY, USA, 2021. Association for Computing Machinery.

[55] Marek Majkowski. When Bloom filters don't bloom. https://blog.cloudflare.com/when-bloom-filters-dont-bloom, March 2 2020. Accessed: 2025-01-13.

[56] Martijn Dekker. Modernish. https://github.com/modernish/modernish, 2016. Accessed: 2025-06-02.

[57] Michael S. Hart and Project Gutenberg. Project Gutenberg. https://www.gutenberg.org, 1971.

[58] Netresec. Publicly available PCAP files. https://www.netresec.com/?page=PcapFiles, 2025. Accessed: 2025-06-02.

[59] Ole Tange. GNU Parallel - The Command-Line Power Tool. *;login: The USENIX Magazine*, 36(1):42–47, Feb 2011.

[60] OpenAI. *OpenAI API: Embeddings Guide*, 2024. Accessed: 2025-06-02.

[61] Paszke, Adam and Gross, Sam and Massa, Francisco and Lerer, Adam and Bradbury, James and Chanan, Gregory and Killeen, Trevor and Lin, Zeming and Gimelshein, Natalia and Antiga, Luca and Desmaison, Alban and Köpf, Andreas and Yang, Edward and DeVito, Zach and Raison, Martin and Tejani, Alykhan and Chilamkurthy, Sasank and Steiner, Benoit and Fang, Lu and Bai, Junjie and Chintala, Soumith. *PyTorch: an imperative style, high-performance deep learning library*. Curran Associates Inc., Red Hook, NY, USA, 2019.

[62] Pawan Bhandari. Solutions to unixgame.io. https://git.io/Jf2dn, 2020. Accessed: 2020-04-14.

[63] Pedregosa, Fabian and Varoquaux, Gaël and Gramfort, Alexandre and Michel, Vincent and Thirion, Bertrand and Grisel, Olivier and Blondel, Mathieu and Prettenhofer, Peter and Weiss, Ron and Dubourg, Vincent and Vanderplas, Jake and Passos, Alexandre and Cournapeau, David and Brucher, Matthieu and Perrot, Matthieu and Duchesnay, Édouard. Scikit-learn: Machine Learning in Python. *J. Mach. Learn. Res.*, 12(null):2825–2830, November 2011.

[64] Peter, Stéphane. makeself - Make self-extractable archives on Unix. https://makeself.io. Accessed: 2025-01-13.

[65] Poess, Meikel and Floyd, Chris. New TPC benchmarks for decision support and web commerce. *SIGMOD Rec.*, 29(4):64–71, December 2000.

[66] Raghavan, Deepti and Fouladi, Sadjad and Levis, Philip and Zaharia, Matei. POSH: a data-aware shell. In *Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC'20, USA, 2020. USENIX Association.

[67] Ritchie, Dennis M. and Thompson, Ken. The UNIX time-sharing system. *CACM*, 17(7):365–375, July 1974.

[68] Roman Perepelitsa and Contributors. zsh-bench: Benchmark for Interactive Zsh. https://github.com/romkatv/zsh-bench, 2021. Accessed: 2025-04-28.

[69] S. Greenberg and I.H. Witten. Directing the User Interface: How People Use Command-Based Computer Systems. *IFAC Proceedings Volumes*, 21(5):349–355, 1988. 3rd IFAC Conference on Analysis, Design and Evaluation of Man-Machine Systems 1988, Oulu, Finland, 14-16 June 1988.

[70] Sadjad Fouladi and Francisco Romero and Dan Iter and Qian Li and Shuvo Chatterjee and Christos Kozyrakis and Matei Zaharia and Keith Winstein. From Laptop to Lambda: Outsourcing Everyday Jobs to Thousands of Transient Functional Containers. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 475–488, Renton, WA, July 2019. USENIX Association.

[71] Schröder, Michael and Cito, Jürgen. An empirical investigation of command-line customization. *Empirical Software Engineering*, 27(2):30, 2021.

[72] Shen, Jiasi and Rinard, Martin and Vasilakis, Nikos. Automatic Synthesis of Parallel Unix Commands and Pipelines with KumQuat. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '22, pages 431–432, New York, NY, USA, 2022. Association for Computing Machinery.

[73] Simon Willison. LLM: A CLI tool and Python library for interacting with Large Language Models. https://llm.datasette.io. Accessed: 2025-06-02.

[74] Spinellis, Diomidis and Fragkoulis, Marios. Extending Unix Pipelines to DAGs. *IEEE Transactions on Computers*, 66(9):1547–1561, 2017.

[75] Tammam Mustafa and Konstantinos Kallas and Pratyush Das and Nikos Vasilakis. DiSh: Dynamic Shell-Script Distribution. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 341–356, Boston, MA, April 2023. USENIX Association.

[76] Taylor, Dave and Perry, Brandon. *Wicked Cool Shell Scripts: 101 Scripts for Linux, OS X, and UNIX Systems*. No Starch Press, 2016.

[77] The Open Group. The Test Environment Toolkit. https://tetworks.opengroup.org. Accessed: 2025-01-01.

[78] The Open Group. VSCPCTS 2016 Test Suite. https://www.opengroup.org/testing/testsuites/vscpcts2016.htm. Accessed: 2025-01-01.

[79] The SAM/BAM Format Specification Working Group. Sequence Alignment/Map Format Specification. https://samtools.github.io/hts-specs/SAMv1.pdf, November. Version 1.6, last modified on 6 Nov 2024. Accessed: 2025-01-13.

[80] Tom White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc, 2009.

[81] Tsaliki, Eleftheria and Spinellis, Diomedes. The Real Numbers for Athens Buses, 2020.

[82] U.S. Securities and Exchange Commission. EDGAR Log File Data Sets. https://www.sec.gov/data-research/sec-markets-data/edgar-log-file-data-sets, 2024. Accessed June 3, 2025.

[83] Utpala, Saiteja and Gu, Alex and Chen, Pin-Yu. Language Agnostic Code Embeddings. In Duh, Kevin and Gomez, Helena and Bethard, Steven, editor, *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pages 678–691, Mexico City, Mexico, June 2024. Association for Computational Linguistics.

[84] Valdemar Švábenský and Jan Vykopal and Pavel Seda and Pavel Čeleda. Dataset of shell commands used by participants of hands-on cybersecurity training. *Data in Brief*, 38:107398, 2021.

[85] Vasilakis, Nikos and Kallas, Konstantinos and Mamouras, Konstantinos and Benetopoulos, Achilles and Cvetković, Lazar. PaSh: light-touch data-parallel shell processing. In *Proceedings of the Sixteenth European Conference on Computer Systems*, EuroSys '21, page 49–66, New York, NY, USA, 2021. Association for Computing Machinery.

[86] Vasilakis, Nikos and Staicu, Cristian-Alexandru and Ntousakis, Grigoris and Kallas, Konstantinos and Karel, Ben and DeHon, André and Pradel, Michael. Preventing Dynamic Library Compromise on node via RWX-Based Privilege Reduction. In *ACM Conference on Computer and Communications Security (CCS)*, pages 1821–1838, 2021.

[87] Wikipedia contributors. Wikipedia: Database Download. https://en.wikipedia.org/wiki/Wikipedia:Database_download. Accessed: 2025-01-13.

[88] Winstein, Keith and Balakrishnan, Hari. Mosh: an interactive remote shell for mobile clients. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, page 15, USA, 2012. USENIX Association.

[89] Yang, John and Prabhakar, Akshara and Narasimhan, Karthik and Yao, Shunyu. InterCode: standardizing and benchmarking interactive coding with execution feedback. In *Proceedings of the 37th International Conference on Neural Information Processing Systems*, NIPS '23, Red Hook, NY, USA, 2023. Curran Associates Inc.

[90] Zhao, Tianqi and Kong, Ming and Liang, Tian and Zhu, Qiang and Kuang, Kun and Wu, Fei. CLAP: Contrastive Language-Audio Pre-training Model for Multi-modal Sentiment Analysis. In *Proceedings of the 2023 ACM International Conference on Multimedia Retrieval*, ICMR '23, page 622–626, New York, NY, USA, 2023. Association for Computing Machinery.

For the latest source code, software dependencies, and multi-tiered input data for the KOALA benchmark suite, please refer to the project's online repository.

Appendix I Benchmark dependencies

The dependencies of each KOALA benchmark can be found in the project's repository. In general, KOALA benchmarks depend on three elements: (1) the source code of each benchmark; (2) input datasets for each benchmark (e.g., Wikipedia mirror for `web-search`); (3) additional third-party components (e.g., Arch Linux packages for `pkg`). The infrastructure offers highly available replicas of all three elements—including over 600 Debian packages, 50 PyPI packages, 110 AUR packages, 1964 npm packages, and their transitive dependencies totalling more than 400 GB of data—to achieve permanent availability and aid reproducibility. This replication is achieved at two levels, namely (1) across all three universities involved in this research, each featuring additional replication and fault-tolerance support, and (2) scalable cloud storage further backed up by permanently-available archival storage on Zenodo.

Appendix II Adoption effort

Evaluating systems with KOALA requires varying levels of effort depending on their design goals, degree of automation, and artifact maturity. For the Shark and `parallel` systems, this involved manual changes to the benchmark scripts. For Shark, modifications typically included identifying parallelizable regions in loops and applying background execution with `&` and `wait`, along with the removal of unnecessary `cat` commands and the use of tee to preserve I/O semantics (*e.g.*, in **`bio`**, **`analytics`**, and **`file-mod`**). In some benchmarks, such as **`ci-cd`**, **`repl`**, and **`unixfun`**, changes spanned tens

of lines due to multiple independent compilation or execution steps. For `parallel`, adaptations involved wrapping loops or commands using the `parallel` utility. This process required changes in the order of 1-3 LoC in benchmarks with stateless pipelines or simple loops (*e.g.*, **`pkg`** and **`nlp`**), but was more intrusive in cases like **`covid`**, **`analytics`**, and **`repl`**, where it required identifying parallelizable regions, exporting computations as shell functions, and occasionally introducing temporary files to manage intermediate data. These modifications reflect the intended use and capabilities of each system rather than any requirement imposed by KOALA, are documented in the public repositories kbensh/koala-shark and kbensh/koala-parallel. In contrast, both *hS* and PASH were applied without any manual changes to the benchmarks.

Appendix III Artifact

The structure of this section mirrors the artifact evaluation process. All information can be found in the frozen `atc25-ae` branch of KOALA's GitHub repository.

- Artifact available: All links pointing to the benchmark source code and accompanying data hosted on two-tiered storage.
- Artifact functional: Usage documentation for the KOALA harness and how to exercise individual benchmarks.
- Results reproducible: Instructions for reproducing static (§5), dynamic (§6) and diversity (fig. 2) characterization results for all benchmarks.

Scope: The available artifact is a set of benchmark programs, all their inputs, the infrastructure for running them, and tooling for their static and dynamic characterization. Specifically, it includes:

- A diverse collection of real-world shell scripts spanning multiple domains and tasks.
- A scalable and durable infrastructure for managing benchmark inputs at different sizes.
- Reusable infrastructure including automation scripts for dependency setup, execution, validation, and correctness checking.
- A suite of static and dynamic analysis tools to characterize the benchmarks.

This artifact can be used by systems researchers and performance engineers to evaluate prototype systems and conduct reproducible experiments over a common shell benchmark suite.

Contents: The artifact includes:

- **Benchmark programs**: 126 scripts grouped in 14 sets.
- **Input datasets**: Three input sizes (`min`, `small`, and `full`), hosted on both archival and scalable storage.
- **Execution infrastructure**: Scripts to automatically download inputs, set up containers, install dependencies, and run benchmarks.
- **Analysis infrastructure**: Scripts for static/dynamic characterization, correctness validation, and principal-component analysis.
- **Documentation**: A top-level `README` and benchmark-specific subdirectories, describing each benchmark's purpose, usage, and input/output structure.

Hosting: The artifact is hosted at:

- GitHub: kbensh/koala Branch: `main` (latest version), `atc25-ae` (frozen for artifact evaluation)
- Permanent archival storage (Zenodo).
- Scalable, fast-access storage.

See Appendix I in the `README` for the full table of input links, or visit kben.sh/data for an up-to-date index of all inputs and benchmark dependencies across all storage tiers.

Installation: KOALA is available via several means, including:

- Git: `git clone git@github.com:kbensh/koala.git`
- Docker: `docker pull ghcr.io/kbensh/koala`
- Shell: `curl -s up.kben.sh | sh`

Installation instructions can also found at the top-level README.

KVCache Cache in the Wild: Characterizing and Optimizing KVCache Cache at a Large Cloud Provider

Jiahao Wang[†1], Jinbo Han[†1], Xingda Wei[✉1], Sijie Shen[2], Dingyan Zhang[1], Chenguang Fang[2], Rong Chen[1], Wenyuan Yu[2], and Haibo Chen[1]

[1]Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University [2]Alibaba Group

Abstract

Serving large language models (LLMs) is important for cloud providers, and caching intermediate results (KV$) after processing each request substantially improves serving throughput and latency. However, there is limited understanding of how LLM serving benefits from KV$ caching, where system design decisions like cache eviction policies are highly workload-dependent.

In this paper, we present the first systematic characterization of the KV$ workload patterns from one of the leading LLM service providers. We draw observations that were not covered by previous studies focusing on synthetic workloads, including: KV$ reuses are skewed across requests, where reuses between single-turn requests are equally important as multi-turn requests; the reuse time and probability are diverse considering all requests, but for a specific request category, the pattern tends to be predictable; and the overall cache size required for an ideal cache hit ratio is moderate. Based on the characterization, we further propose a workload-aware cache eviction policy that improves the serving performance under real-world traces, especially with limited cache capacity.

1 Introduction

Large language models (LLMs) have become an important cloud service thanks to their emerging capabilities in various tasks. Both customer-facing services (to-C) like conversation-based chatbots [43] and business services (to-B) through LLM API calls have been widely deployed on the cloud [42, 21, 4]. As online services, serving LLMs with low latency and high throughput is critical, but achieving this in a cost-efficient way is challenging due to the huge computational requirements to serve each LLM request.

Caching intermediate results of served requests (KV$) has become a common methodology to improve serving performance [45, 5]: when two requests share the same input prefix, their KV$ are identical; so if we cache the KV$ of the first (KV$ cache), we don't need to compute the KV$ for the next. Hence, the serving latency is reduced with the throughput improved thanks to the reduced computation under hits.

Like traditional data caching systems [60, 7, 40], the effectiveness of KV$ cache system components like cache eviction policy should be tailored to the KV$ reuse features of workloads. However, LLM serving workloads are complex due to their diverse request types as well as the unique features of each request type. For example, a production LLM service handles both to-C workloads (like chatting) and to-B workloads (like task automation with API calling). These requests can further be processed either as single-turn or multi-turn interactions (§2). Since the KV$ can be reused across different single-turn requests and multi-turn requests, both the single-turn request types and the multi-turn patterns can possibly affect the design choices of KV$ cache.

	Time	Type	UID	Turn	Content
ShareGPT [2]	✗	✗	✗	✓	✓
Mooncake [1]	✓	✗	✗	✗	✓
Traces in this work	✓	✓	✓	✓	✓

Table 1: *A comparison of currently available LLM serving traces. **Time** is the invocation time of each request. **Type** is the request type, e.g., whether it is an API call. **UID** is the user ID that invokes the request. **Turn** is the multi-turn information of the request. **Content** is the (anonymous) input and output tokens of each request.*

There remains a significant gap in the understanding of common LLM serving workloads for KV$ cache due to the lack of in-depth analysis of real-world traces. First, it is unclear whether production workloads have many reuses and which request type contributes most to the reuses. Second, the distributions of the reuse time or reuse probability of KV$ remain unknown, which is critical to cache policy designs. Finally, there is a lack of characterization of the lifespan of KV$, which is essential in helping us estimate the required cache capacity for serving LLMs in the wild.

Characterizing production traces. To bridge this gap, we collected and characterized two representative production serving workloads from one of the world's largest cloud providers (ALIYUN): a to-C workload where users submit requests to the cloud-hosted LLM service via browser-based chatbots or mobile applications, and a to-B workload where developers send OpenAI-compatible requests to the LLM service through API calls in their programs. We collected as much detailed information as possible within the company's strict privacy policy, including request submission times, request types (single-turn, multi-turn, API requests, or others),

[†]The two authors contributed equally to this work.

[✉]Xingda Wei is the corresponding author (wxdwfc@sjtu.edu.cn).

and anonymized content to analyze caching effectiveness. To our knowledge, no other available LLM serving traces match the information depth and scale of our traces (summarized in Table 1). For example, ShareGPT [2] only provides the input and output of each request, not the submission time of each request, which is critical for analyzing the real KV$ hit ratio and usage over time. We believe our collected workloads represent typical LLM serving workloads and can provide a foundation for future KV$ cache system design.

From the perspective of designing an efficient KV$ cache system—both in performance and resource usage, our *key takeaways* are as follows:

1. KV$ reuses are common, but the reuse ratio is smaller than previously reported numbers on synthetic datasets [65, 65]. The reuses follow a skewed distribution: 10 % of KV$ blocks contribute to 77 % of the reuses. The contribution of KV$ reuses also varies across traces: contrary to a straightforward observation that multi-turn requests may dominate most reuses, single-turn requests dominate 97 % KV$ reuses in to-B workloads (§3.2).
2. Key workload features related to KV$ designs, like reuse time and reuse probability distributions—vary across different LLM request types (API vs. chat) and the number of turns in multi-turn requests. Despite this diversity, for each specific request category (type plus turn number), the reuse time is predictable based on the historical information (§3.4). Since the request categories are known by the serving system, we can systematically use the estimated probability for improving cache policies.
3. The lifespan of KV$ is ephemeral: the P99 lifespan of KV$ in to-B workloads is 97 seconds, so a small cache is typically sufficient for the KV$ cache. For example, in the to-B workload, a KV$ with capacity 2 × of the GPU HBM per-GPU is sufficient to approach an ideal hit rate under infinite capacity on common GQA models [3] using standard cache eviction policies like LRU (§3.5).

Our characterizations above reveal several important design decisions that current KV$ cache systems may have overlooked. First, optimizing caching for single-turn requests is equally important as optimizing for multi-turn requests. Second, we could improve cache hits by adopting a workload-aware caching policy instead of only adopting a workload-agnostic policy like LRU. Finally, the ephemeral lifespan of KV$ suggests that a least-frequently-used (LFU) policy is not suitable because a short-lived KV$ may have high frequency in the past. Meanwhile, a small (on-GPU) cache is sufficient for API-dominated to-B workloads, eliminating the cost and complexity of deploying and managing a CPU-RDMA-SSD storage hierarchy for KV$ cache. Besides the main takeaways described above, we also discovered several interesting findings, such as the KV$ hits across users being extremely low (§3.2), despite the possibility of shared prompts via prompt libraries [6].

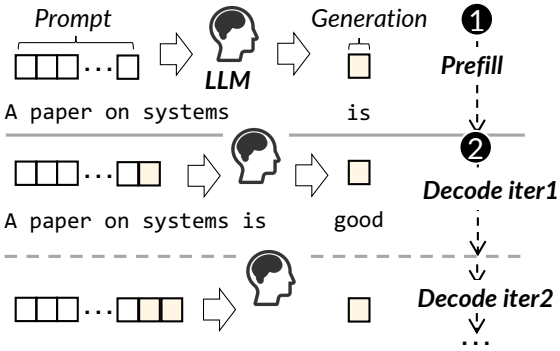


Figure 1: *An illustration showing how an LLM processes requests.*

Improved KV$ cache eviction policy. Based on our findings, we propose a workload-aware KV$ eviction policy (§4.2) that leverages the profiled reuse probability distributions of each workload to enhance cache hit ratios on real-world workloads. Specifically, we leverage the empirically sampled probability distributions of the reuse probability of each workload to decide the priority of each KV$ block, with considerations of both the characterized spatial locality and the lifespan of KV$. To demonstrate the effectiveness of our improved design, we have integrated our policy into vLLM [30]—a popular LLM serving system on the cloud, and our extensive evaluations on production traces show that the improved design can improve 3.9 % cache hits as well as up to 41.4 % mean response time improvements compared to workload-agnostic policies like LRU and LFU.

Discussion. Before moving on, we note two limitations of our work. First, our results are based on one week production traces of large-scale LLM serving workloads. While we believe such workloads (chatbot and API) are representative nowadays, LLM serving workloads are evolving, with new workloads like reasoning [44] emerging. We leave the characterization of these newer workloads to future work. Second, our primary focus is on the KV$ cache policy design, yet we believe the design of other system components for LLM serving like the policy of global scheduling could also benefit from our findings, and we leave other explorations as future work. Nevertheless, we believe our methodology—first characterizing real-world serving features and then optimizing a specific serving component—generalizes to other LLM serving workloads and systems.

A sample of our analyzed trace is available at https://github.com/alibaba-edu/qwen-bailian-usagetraces-anon.

2 Background: LLM serving, KVCache, KV-Cache cache and Multi-turn Serving

LLM serving. Large language models (LLMs) process requests in an auto-regressive way, as shown in Figure 1. Given a request (or a batch of requests), where each request contains multiple tokens (prompt), the model first executes a prefill

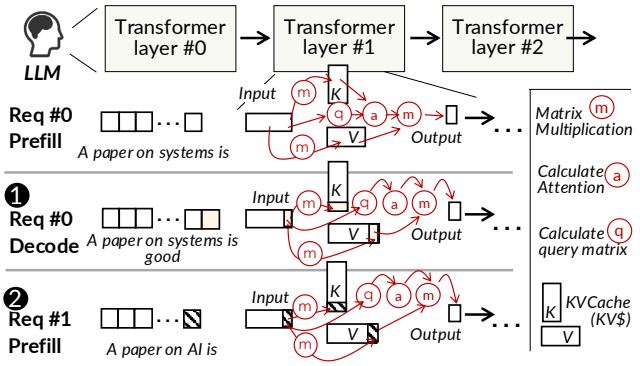


Figure 2: *An illustration of:* ❶ *how* KV$ *from a prefill request (Req#0) can be reused by the decoding of Req#0, and* ❷ *how* KV$ *can be reused for the prefill of a future request (Req#1).*

phase to generate the first token (❶) with a forward pass. The model then enters a decoding phase (❷) that iteratively generating the remaining tokens with multiple forward passes. The iteration ends after the module outputs an end-of-sequence (EOS) token. During each iteration, the input consists of both the original prompt and all previously generated tokens.

KVCache (KV$). LLM performs three steps for a forward pass [55] as shown in Figure 2: (1) calculating the Q, K, and V matrices using matrix multiplication, (2) using Q and K to derive attention scores, and (3) combining the attention scores with the V matrix to produce the attention result. These steps are computation-intensive. Since inputs with the same tokens share the same K and V matrices (see ❶), all LLM serving systems cache the K and V matrices generated during the prefill phase on the GPUs to save computation time. These cached K and V matrices are then reused during the decode phase for the same request to avoid recomputation, so they are termed as KV$.

KV$ cache and KV$ blocks. Besides KV$ reuse across prefill and decode phases when processing a single request, the KV$ can also be reused across different requests if they share the same token prefix. As shown in Figure 2 (❷), the K and V values of "A paper on" are identical for both #Req0 and #Req1. Thus, even when #Req0 finishes decoding, we can still cache its KV$ for #Req1, which dramatically improves the reaction time (also termed time to first token, *TTFT*) of #Req1: it only needs to compute the K and V of "AI is" during the prefill. KV$ caching further increases the serving throughput thanks to the saved computation. Due to the above reasons, existing serving systems all cache the KV$ of completed requests. We term this approach KV$ cache [30, 18, 20, 69, 26, 37, 65], though other names like prefix cache or prompt cache also exist. The KV$ can be cached in GPU memory, on the host memory (or remote machine's memory), or even SSD [30, 18, 65].

Since checking prefix-match may be costly at a per-token level, the cache granularity of existing KV$ system is *block*, where each block contains the KV of a configurable number


```
{ "timestamp":          "202x-xx-yy 17:41:33.945",
  "chat_id":            "1234567890abcdef1234567890abcdef",
  "parent_chat_id":     "aabbccddeeff00112233445566778899",
  "user_id":            "10000012345678900",
  "req_type":           "text",
  "num_input_tokens":   11026,
  "num_output_tokens":  312,
  "tokens_hashes_input": [1001,2002,3003,4004,5005,...],
  "tokens_hashes_output": [1234,2341,3412,4123,10000,...], }
```


Figure 3: *An example of the collected trace record.*

of tokens. For example, vLLM [30] by default groups 16 tokens as a block.

Multi-turn serving. Interacting with LLMs through multi-turn conversations is a common pattern to improve serving quality [58, 56]. Specifically, after receiving the generation from the LLM, the user may issue a next-turn request with a new prompt. Since the LLM needs the previous conversation context for processing, the serving system will concatenate the input and output of requests from the previous turns to the new request's input, and feed the new input to the LLM for serving. To help manage the history of multi-turn requests, requests in a multi-turn conversation are typically grouped in a *session*. A multi-turn request naturally reuses KV$ from previous turns since current LLM systems deliberately append the history of inputs and outputs as a prefix to each new request.

3 Characterize KV$ Workload in the Wild

3.1 Trace data collection

We collected and analyzed two weeks (February 2025 and December 2024) of LLM serving request traces from one of ALIYUN TONGYI's serving clusters. For brevity, we present results from one representative day per trace. The results remain consistent throughout the workdays in each week (see §A.1). *Note that we cannot collect the raw input and output for each request due to* ALIYUN*'s strict privacy policy.* For a specific trace, all its collected requests are served by the same model.

Below, we first describe the detailed anonymized trace record provided by the operational team of the serving cluster, then we move on to highlight the differences between two traces, both representing typical serving workloads nowadays. Figure 3 presents the layout of a collected request:

1. Timestamp: The time when the serving cluster receives the request.

2. Chat ID: The unique ID of the request.

3. Parent Chat ID (under multi-turn serving): For requests in a multi-turn session, the serving system assigns a parent

ID to each request, linking it to the previous request in the session. By traversing the parent IDs, we can identify all requests belonging to the same session. If the parent ID is empty, then the request is the first-turn of a session and we call them *single-turn request*. Otherwise, we call requests with a non-empty parent ID *multi-turn request*.

4. User ID: The unique ID of the user submitting the request. These IDs are mapped to a randomly selected domain for anonymity.

5. Request type: The serving gateway categorizes LLM requests into five types: *Text*: Plain text input via interfaces like ChatBot. *File*: file analysis (e.g., summarizing document content using LLM). *Multimodal*: Image understanding through OCR-enhanced processing. *Search*: Frontend with web-search integrated for knowledge augmentation. *API*: Programmatic access via OpenAI-compatible interfaces [42].

6. Number of Input/Output Tokens: The number of input tokens provided to the LLM and the number of output tokens generated by it.

7. The hashed values of In/Output tokens: Due to the strict privacy policy, the runtime system generate the traces through salted hashing and domain remapping. Specifically, the collector employs SipHash [8] to generate unique hashes for every four consecutive tokens with random salts per trace, then remapping the hash IDs to consecutive natural numbers. This fine-grained hashing enables more fine-grained cache analysis while enhancing privacy protection compared to single-token hashing.

Trace A vs. Trace B. We collected two traces (A and B) representing two representative LLM serving workloads. Trace A represents a common to-C (customer) scenario where users interact with the model through services built by the cloud, i.e., ChatBot (Text) or File/Multimodal/Search services. A key feature of to-C workloads is that they are typically human-in-the-loop. Trace B represents a to-B (business) workload where business users interact with the LLM by calling OpenAI-compatible APIs hosted by the cloud through computer programs [42]. API calling is critical for business users because they can systematically leverage LLM to automate tasks like document translation or user request classification. Trace B has no request type information because, similar to the LLM interface calls currently offered by other vendors [42, 21, 4], this information is concatenated on the client side.

3.2 KV$ reuses analysis in the wild

How many KV$ can be reused? Figure 4 shows the ideal cache hit rates on two traces, i.e., assuming a cache having infinite capacity. We can see that LLM workloads can benefit from caching—it has 62% and 54% hit rates on Trace A and B, respectively. The hit rate is calculated as follows: for a total of N KV$ blocks that needed to be computed by LLM in a trace, if the hit rate is 62%, this implies that 62% of the

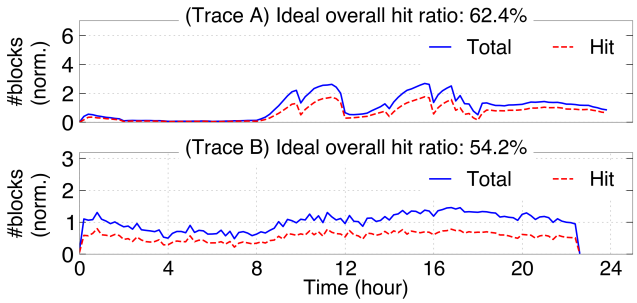


Figure 4: *An analysis of the ideal cache hit ratio of the* KV$ *cache under real-world LLM serving workloads within a day. The reported accessed and hit block numbers are normalized (norm.).*

Types (Trace A)	Workload ratio (%)	Multi-turn Ratio (%)
Text	78	47
File understanding	4	51
Multimodal understanding	3	10
Search	15	49
Types (Trace B)	**Ratio (%)**	**Multi-turn Ratio (%)**
API	100	0.08

Figure 5: *Workload types and multi-turn ratio of requests in our collected traces.*

N blocks can reuse a KV$ computed and cached in a prior request. While the ideal hit ratio is high, it is smaller than the reported hit ratio (e.g., more than 80% [65, 69, 19]) on synthetic workloads.

Single-turn and multi-turn requests are equally important for KV$ reuse. While it is a common belief that multi-turn requests are key contributors to the cache hits [65, 19], this is not the case on the to-B workloads (Trace B). Figure 5 shows the contribution of each request type as well as their multi-turn ratio. For Trace A, it is dominated by the requests sent by the chatbot (78 %), and all request types except for the multimodal have a similar multi-turn ratio (47–51 %). For Trace B, it only has API requests with a negligible multi-turn ratio (less than 0.1 %). Nevertheless, the single-turn requests contribute 97 % of the total cache hits for Trace B.

Single-turn API requests dominate the hits in to-B workloads for two reasons. First, LLM API requests share KV$ through common system prompts. Specifically, each request comprises: (1) a system prompt providing task instructions (e.g., document translation guidelines) and (2) a user prompt containing request-specific data (e.g., target document). When programs make API calls [35], they typically hardcode identical system prompts in the program, thus enabling KV$ sharing across requests, even for single-turn requests. Second, API requests often exhibit rapid submission rates (e.g., > 10 QPS), enabling frequent reuse of the same shared KV$ from system prompts.

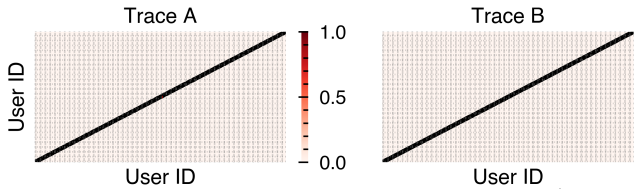


Figure 6: *An analysis of whether a user may hit the* KV$ *cached by another user. The darker the color, the higher the normalized hit count. Note that we have amplified the diagonal line in case it is too thin, since the trace provider serves numerous users.*

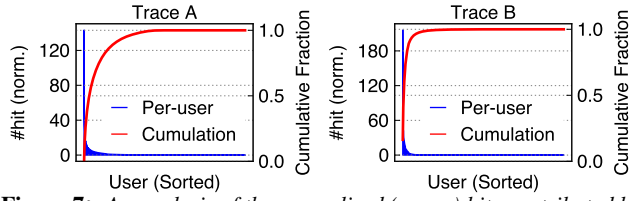


Figure 7: *An analysis of the normalized (norm.) hits contributed by different users in an ideal setup with infinite cache size.*

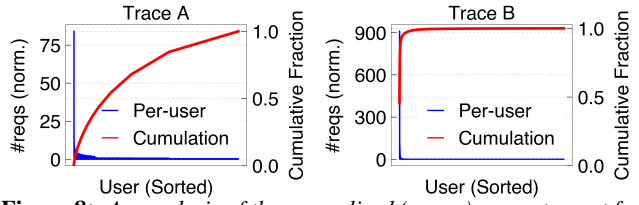


Figure 8: *An analysis of the normalized (norm.) request count for different users.*

Cross-user KV$ hits are rare. Figure 6 shows the heatmap of the cache hits between users, assuming an infinite-size KV$ cache. Darker colors indicate higher normalized hit counts. We can see that for each user, most hits come from requests submitted by themselves (the diagonal line). While users naturally reuse KV$ from their own requests—either via multi-turn conversations or through programs calling API with the same system prompt for the same task, the low inter-user hit rate suggests that users tend to customize their own system prompts rather than using template prompts provided by third-party libraries [6]. As a result, we suggest paying more attention to intra-user KV$ reuses rather than inter-user [20, 69, 26, 37] when designing KV$ systems.

The KV$ reuses are skewed. Figure 7 shows the KV$ cache hits contributed by different users, along with the cumulative hits. The KV$ hit pattern is highly skewed: 19 % of the user requests in trace A and 4 % in trace B account for more than 90 % of the KV$ block hits. This skewness stems from two factors. First, the requests are unevenly distributed across users, especially in Trace B. Figure 8 shows the normalized number of requests sent by different users: 15 % of the users (head users) contribute 90 % of the requests in Trace B. Since KV$ hits primarily occur for requests from the same user (see Figure 6), the skewness in request counts is expected to lead to a similar skewness in KV$ hits. Second, even when requests are sent relatively evenly by different

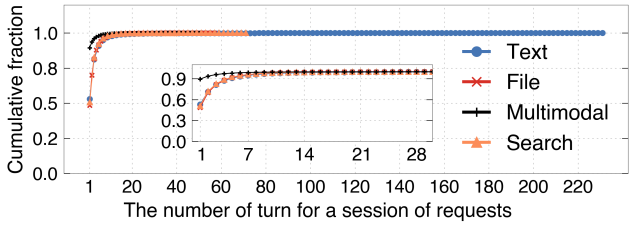


Figure 9: *Distribution of turn number of multi-turn requests on Trace A.*

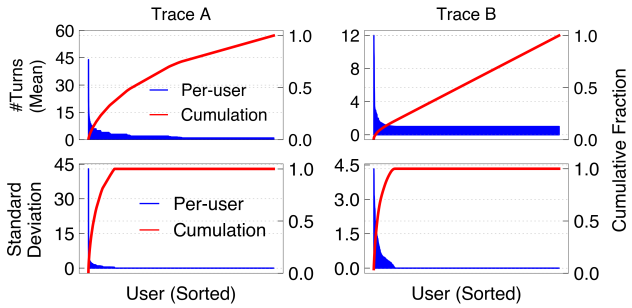


Figure 10: *An analysis of the mean (upper) and standard deviation (lower) of the number of turns of requests sent by each user. The users are sorted across to their mean number of turn (upper) and standard deviation (lower).*

users, individual users vary significantly in their KV$ hits. As also shown in Figure 8, Trace A exhibits a moderately skewed request distribution: 19 % of users generate 50 % of the requests, and 72 % generate 90 % of the requests. Nevertheless, the KV$ hits are still skewed. We suspect that this is due to their tendency to engage in multi-turn requests.

3.3 Analyses of multi-turn requests features

We first examine the features of multi-turn requests, as they still dominate the KV$ reuse in to-C workloads. We omit the results on Trace B because multi-turn requests are rare in it.

Multi-turn requests have high variability. We observed two kinds of variability. First, the number of turns varies significantly across different sessions. Figure 9 shows the distributions of turn numbers on Trace A: 54 % of requests are single-turn requests, while the 50^{th} and 90^{th} percentile turn numbers are 1 and 5, respectively. The distribution has a long tail—the longest session has 232 turns, two orders of magnitude larger than the median (232 ×).

Second, multi-turn request distributions exhibit significant user-level variation. Figure 10 shows that most users have similar mean turn counts, except for a few head users—the top 10% exhibit 8.9 × more turns than the average. Additionally, turn count deviations vary substantially, ranging from 1 to 44. This suggests diverse interaction patterns: some users show high variability, while others remain stable.

The probability of issuing the next turn request is predictable given a workload category and time interval, and the probability differs across request categories. A request

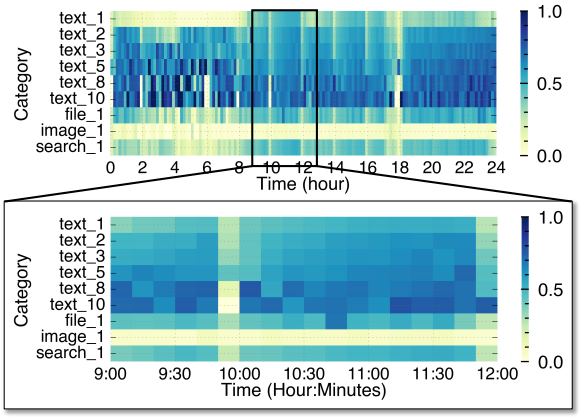


Figure 11: *An illustration of how the frequencies of issuing the next turn request change over time for different request types. The x-axis is the timeline. The workload categories are sorted according to their popularity.*

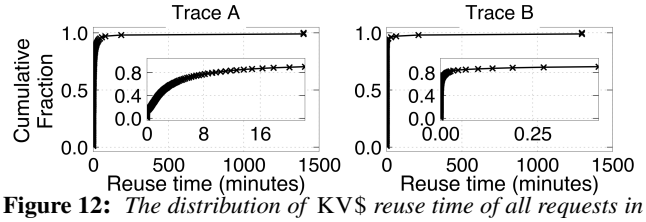


Figure 12: *The distribution of* KV$ *reuse time of all requests in Trace A and B.*

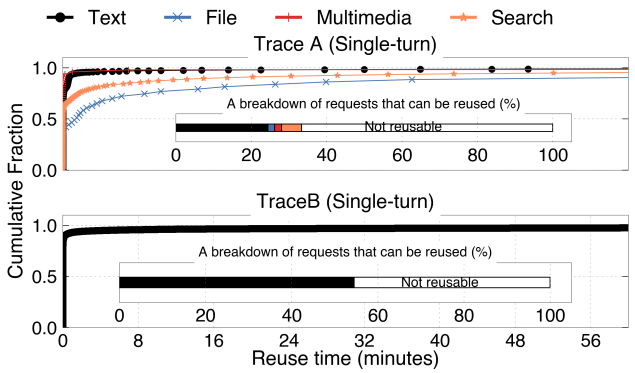


Figure 13: *The distribution of* KV$ *reuse time of single-turn requests in Trace A and B. B has one category of request (API).*

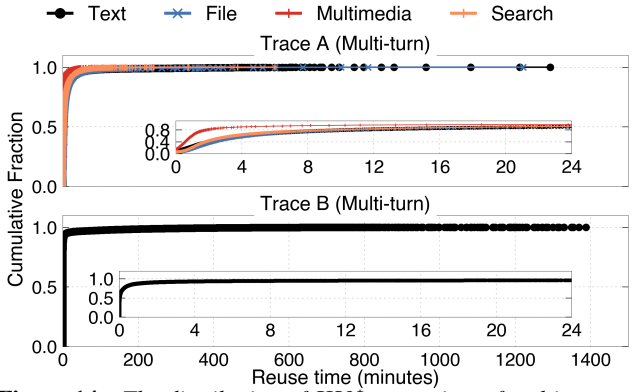


Figure 14: *The distribution of* KV$ *reuse time of multi-turn requests in Trace A and B. B has one category of request (API).*

category includes both the original request type (e.g., Text and Search) as well as the turn number of the request, i.e., a 1-turn text (single-turn text request) is different from the 2-turn text request. Figure 11 shows a heatmap of how the probability of issuing the next-turn request changes over the long time interval (24 hours) as well as a zoomed-in view of a short time interval (3 hours). The intensity of the color indicates the profiled frequencies within a time window (10 minutes). We can see that though different request categories have different frequencies (shown by vertical lines), for a specific category, the frequency of issuing the next-turn request remains consistent over a one-hour time interval (shown by horizontal lines). We attribute this to the fact that the probability distribution of KV$ reuse is quite fixed given a specific time interval and request category. §3.4 further elaborates on that. Interestingly, the frequency of issuing the next-turn request differs across request categories. For example, single-turn requests demonstrate significantly lower rates of issuing next-turn requests (mean=0.5) compared to multi-turn requests (mean=0.8).

3.4 Analyses of temporal and spatial locality

Temporal locality can be characterized by profiling the distribution of reuse time, i.e., the interval between when a KV$ block is reused by another request.

KV$ reuse time is short. Figure 12 shows the distributions of KV$ reuse time on both traces. We can see that the reuse time for both workloads is typically small: In Trace A, 80% of the reuse time falls within less than 10 minutes, while in Trace B it falls within 10 seconds. The differences lie in the human-in-the-loop versus computer-in-the-loop interactions that generated the requests in two traces.

Different workload types have different temporal locality. First, single-turn requests have much lower locality in the to-C workloads, while to-B workloads are the opposite. Figure 13 reports whether a single request can be reused by another single-turn request on both traces in the middle sub-figures. As we can see, less than 30 % of the single-turn requests are reusable in Trace A, but more than 50 % of requests can be reused in Trace B.

Second, different request types show different temporal locality. Figure 14 shows the reuse time distribution of multi-turn requests categorized by their request types. We can see that chat requests (Text) shows longer reuse time than image understanding (Multimodal), but shorter than file understanding (File). This stems from workload characteristics: users typically spend more time processing complex LLM outputs based on file content than casual outputs from conversational chats. Conversely, visual information processing occurs faster than textual processing [48], leading to shorter reuse times for image tasks. Finally, Search workloads exhibit the second-longest reuse time, as users often require substantial time to

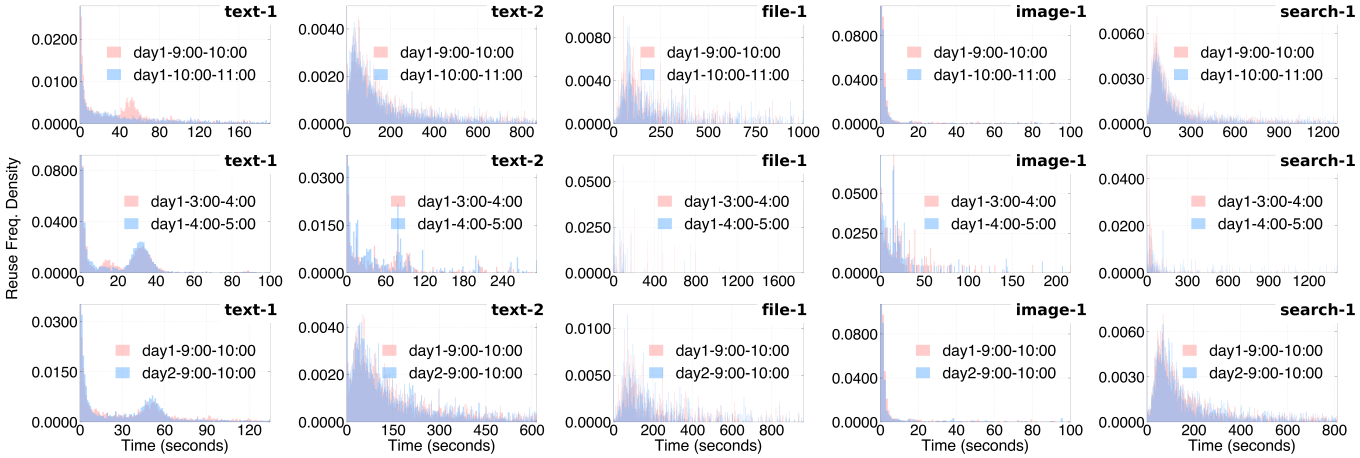


Figure 15: *An empirical analysis of the reuse time probability distribution of* KV$ *blocks. The columns present data from different request categories. The first row: distributions under heavy traffic (daytime). The second row: distributions under low traffic (nighttime). The third row: a comparison of distributions under different days but with similar traffic patterns.*

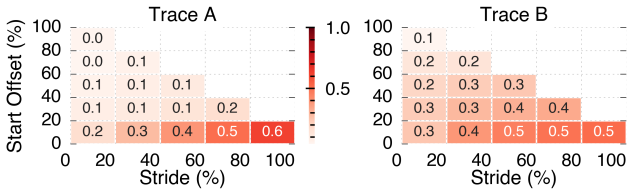


Figure 16: *Overall spatial locality of two traces.*

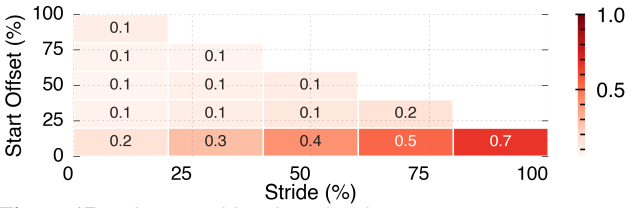


Figure 17: *The spatial locality of multi-turn requests on Trace A.*

analyze the searched results.

Given a time period and request category, the probability distribution of KV$ reuse time follows an exponential distribution. Figure 15 presents the reuse time probability distribution of KV$ blocks across different request categories. For each category, we measure the likelihood of its KV$ reuse at specific time intervals after caching. We plot these distributions by counting reuse event frequencies at each second following request execution.

From the results, we draw three key observations. First, exponential distributions fit well with the reuse probability given a request category. Second, the probability is workload-aware, i.e., the distributions of different request categories, including the same request type but different turns, are different (e.g., text-1 vs. text-2). This implies a workload-aware design that distinguishes different request categories. Finally, the distributions of requests with similar traffic patterns are similar. Take text-1 as an example: as shown in the first row, the distribution drawn from 9:00–10:00 a.m. is similar to that drawn from 10:00–11:00 a.m., but is significantly different from that drawn from the nighttime—3:00–4:00 a.m. (second row). The distributions are also similar across different days (first row vs. third row). The first and third observation in combine imply that we can acquire the distribution of a request category using recent historical data.

Spatial locality. Figure 16 shows the overall spatial locality of KV$ across both traces. We characterize the spatial locality by measuring the ideal cache hit ratio with various cache strides, i.e., the percentage of tokens cached per request, as well as different starting token offsets for caching. This gives a two-dimensional heatmap, for example, offset (10 %, 10 %) means that for each request, we only cache 10 % of the tokens starting from the 10 % token position.

First, KV$ shows better spatial locality when caching from the beginning, which is expected since only requests with identical prefixes can share the KV$ (§2). We emphasize this seemingly trivial point because some KV$ optimizations [27] cache the tail tokens of requests to hide KV$ load time through concurrent computation and loading. However, this approach may reduce spatial locality, as Figure 17 particularly demonstrates that reusing intermediate parts in the cache yields minimal benefits for multi-turn conversations.

Second, the spatial locality, like temporal locality, is also workload-aware. In Figure 18, we observed workloads like text and multimodal exhibit obvious spatial locality, while file and search exhibit almost no spatial locality. Due to the lack of user input content, we suspect the cause as follows: the applications add a fixed system prompt for text and multimodal workloads, whereas the system prompt for file and search is dependent on user inputs.

Finally, we found that increasing the stride size implies a higher cache hit ratio in Trace A (Figure 17), but it is not the case in Trace B (the last column in Figure 18). In Trace A, multi-turn requests contribute to the cache hits, so caching

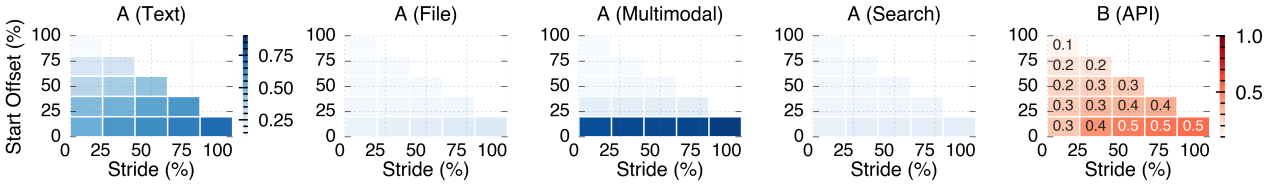


Figure 18: *The spatial locality of single-turn requests.*

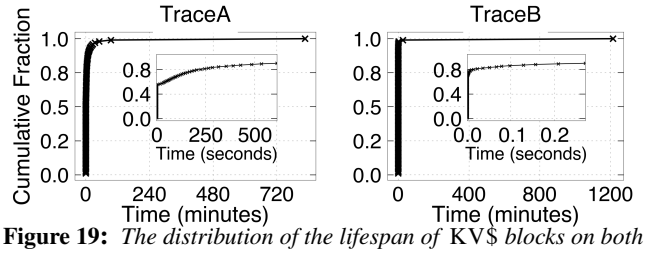


Figure 19: *The distribution of the lifespan of* KV$ *blocks on both traces.*

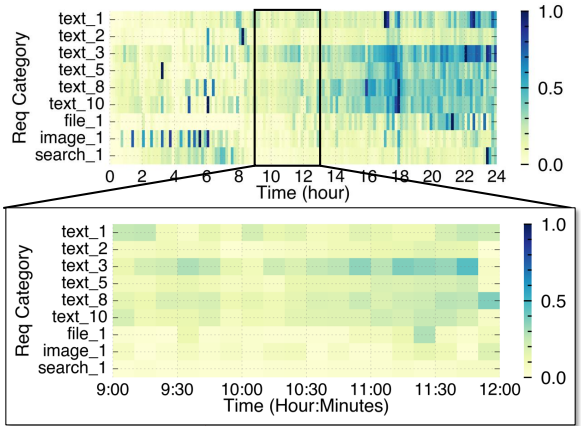


Figure 20: *An illustration of how the mean lifespan of* KV$ *blocks changes over time for different request categories. Note that the x-axis represents the timeline. The request categories are sorted according to their usage.*

more tokens for a request will lead to more cache hits. In Trace B, only the system prompt contributes to the cache hits. Thus, caching a stride of 50 % of the request already achieved a peak hit ratio of 50 %. This implies that most system prompt lengths are below 50% of the total request length in Trace B, though we don't have the detailed request to support this suspicion.

3.5 Analyses of KV$ cache capacity requirement

Determining the right cache capacity is critical for a caching system. Unlike traditional storage systems, the cached KV$ is ephemeral, so identifying the right size requirement can save platform costs. This section analyzes the KV$ capacity using a bottom-up approach: First, we characterize the features of the lifespan of requests' KV$. Then we analyze the storage usage of each request. In combination, this gives us an estimation of the total KV$ capacity required.

KV$ lifespan is ephemeral and predictable. Figure 19 analyzes the lifespan of KV$ blocks. A KV$ block "dies" if it has not been reused anymore. We can see that most KV$ blocks "live" shortly. In Trace A, 90 % of the KV$ blocks are not reused after 612 seconds, and the number for Trace B is 0.3 seconds. This aligns with our analysis in the previous sections, e.g., many multi-turn requests will have a short reuse time (Figure 12), and most requests have a few turns (Figure 9). As a multi-turn request ends, its KV$ will not be reused anymore. Blocks in Trace B have a shorter lifespan despite some outliers. Unlike Trace A, most of its cached KV$ blocks are reused for system prompts, so they can be reused after a long time. For example, in Trace B, the longest lifespan is 1,200 minutes while it is 800 minutes in Trace A.

Figure 20 further shows how the mean lifespan of KV$ blocks changes over time for different request categories. In the graph, the color intensity represents the normalized lifespan, with darker colors indicating longer-lived KV$ blocks. We can see that although the average lifespan at the 24-hour scale is changing over time, when we zoom in to a smaller timescale (9:00 a.m. to 11:00 a.m.), the pattern is quite stable. This implies that we can predict the lifespan of a KV$ block using historical data to reduce the KV$ cache size for saving costs, as storing KV$ at fast storage medium (e.g., CPU's DRAM) is expensive.

Per-request KV$ cache usage is moderate. A key factor when considering caching is the size of KV$ to be cached, which is proportional to the input length of each request and the corresponding model output length. Figure 21 shows the size distribution of requests on different traces. Note that we report the relative size when normalized to the HBM available for serving the request on a serving instance[1]. This is because a relative value gives more intuitive results on the capacity required to cache the KV$ of each request. The available HBM for KV$ is calculated by subtracting the total HBM of the GPU(s) in a serving instance from the model parameter and activation sizes. Since the KV$ size of each request depends on the detailed attention mechanism of the model, we report the results on three representative models with two widely used attention methods: group query attention (GQA [3]) and multi-head attention (MHA [55]). GQA uses less memory for the KV$.

We can see that the overall size of the KV$ is moderate compared to the HBM. Take Trace A as an example. On GQA

[1]A serving instance is the minimal number of GPUs required to serve a model. For example, the serving instance of an 8 B model only needs one GPU, while serving a 72 B model requires 4 in the common case.

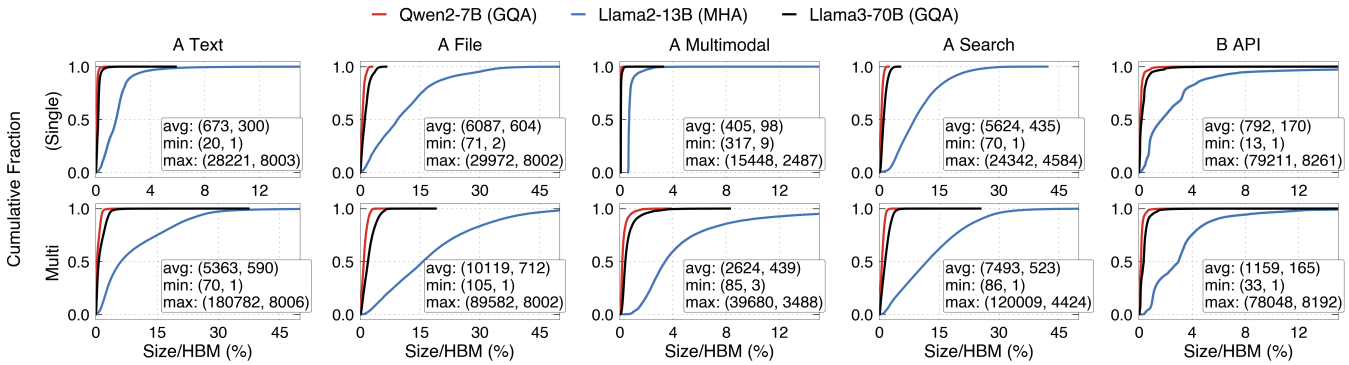


Figure 21: *The distribution of* KV$ *size of requests on different traces for Single and Multi-turn requests. The size is normalized to the GPU memory (HBM) available for serving. The number in the tuple inside each graph represents the length of input and output.*

models like Qwen2-7B, the 50^{th}, 90^{th}, and 99^{th} HBM usage of the request in Text—the dominant request type are 0.09 %, 0.15 %, and 0.37 % and 0.36 %, 1.31 %, and 2.21 % for single-turn and multi-turn requests, respectively. The average usage is 0.09 % and 0.56 % for single- and multi-turn requests, respectively. This small usage is due to the fact that each token consumes few megabytes and requests in production traces tend to be small: in Qwen2-7B, it consumes 0.875 MB KV$ for 16 tokens, while the mean total token lengths (input+output) for single-turn and multi-turn is 973 and 5,953, respectively. For other models, though Llama3-70B also adopts GQA, it has a slightly higher HBM usage due to the reduced number of GPUs per instance: the parameter size increases 10 × while the number of GPUs we measured only increases 4 ×. Finally, MHA models have the highest usage due to the increased KV$ per token. Note that recent open-source models tend to use GQA (or other attention with even fewer KV$ per token like MLA [36]).

Considering that each serving instance typically processes a few requests per second, our characterization implies that an instance can cache a request for a modest time—e.g., for Qwen2-7B, suppose a GPU can process 5 requests per second, it takes at least 1 minute to saturate the HBM assuming an average 0.32 % HBM per request (derived with a 47 % multi-turn ratio of Text request type). This time is within the reuse distance of KV$ in many scenarios (e.g., API). Note that in practical setups, we may not achieve such a long reuse time due to tail requests (e.g., we do observe long context requests with more than 12,000 tokens), and LLM serving systems typically serve requests in large batches to improve computation utilization.

Overall memory required for KV$ cache is moderate. We analyze the KV$ cache capacity required by measuring how large a cache can ensure an ideal cache hit ratio assuming an ideal eviction policy, i.e., never evict a KV$ that will be reused. The measurement is as follows: at a given time, upon processing a request, we update the KV$ size by adding the KV$ of this request if it will be reused, and subtracting the size of all cached KV$ that will never be reused. One tricky

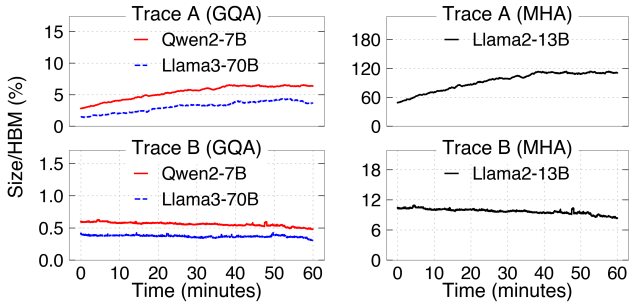


Figure 22: *An analysis of the* KV$ *cache size required to achieve an ideal cache hit ratio on different models and traces. The data is collected from 10:00 a.m. to 11:00 a.m.*

thing is that the size required is dependent on the number of serving instances deployed for a workload: if the instances are over-provisioned, then the estimated requirement would be small. To obtain the maximal requirement, we use a setup that uses the minimal number of serving instances, i.e., each instance serves with the maximum QPS, so there is no over-provisioning.

Figure 22 shows the results normalized to the total HBM of a serving instance. We can see that GQA models require a relatively small cache capacity to reach an ideal cache hit rate: in Trace A, Llama3-70B only needs 4 × of the available HBM for KV$. This is typically achievable in modern serving clusters without the need for more storage hierarchies like remote memory or SSD [19]. For example, in a typical setup featuring 8 A100 GPUs and 1 TB of CPU memory, each GPU has access to an average of 128 GB of CPU memory, which is nearly 4 × the amount of HBM space reserved for storing the KV$. Moreover, for Trace B, the KV$ capacity is even smaller than the reserved HBM, so caching on GPU may be sufficient.

Though GQA needs small cache, we should note that MHA models do require a huge amount of KV$ cache, This is due to their huge per-token KV pairs (see Figure 21). Thus, optimizing cache eviction policies is still important under limited cache capacity.

4 Improved KV$ cache system

4.1 An overview of the design space

Existing KV$ cache systems [19, 30, 69, 65] follow similar caching mechanisms and policies. Below we give a brief overview of their designs before moving on to our improved policy.

Current KV$ cache mechanism and policies. Regarding the mechanism: existing systems cache KV$ primarily on GPU and CPU in a hierarchical manner. The KV$ is managed at block granularity (see §2): when a KV$ block is generated, it is first cached on the GPU HBM. If the GPU is unable to cache newly generated blocks due to out of memory, the caching system adopts an eviction policy (described below) to evict some blocks from the GPU to the CPU. The evicted blocks are asynchronously swapped to CPU in a layer-by-layer manner so the swap cost is negligible. If the CPU is also out of memory, the blocks are removed. Before executing a new request, if the request hits cached blocks, the serving system directly reuses the cached blocks to avoid recomputation. Note that if the hit is on CPU, the block is swapped to the GPU also in a layer-by-layer manner.

Regarding the policies: a KV$ cache system mainly needs two policies: how to determine cache hits and how to evict cached blocks. For the first policy, the design choices differ in their granularity of checking which blocks may contribute to the hits: some systems check for matches across all requests [69, 26] while others only check for matches within requests from individual users [65, 19]. For the second policy, current systems mainly adopt standard eviction policies like LRU or FIFO. Though they adopt some extensions like checking whether the evicted blocks are in the scheduling queue [19], the overall rationale does not change.

The drawback. Existing workload-agnostic policies like LRU may miss the workload features we analyzed in §3, and we argue that they are suboptimal. Take Trace A as an example: given the same amount of time since the last access (e.g., greater than 10 seconds), 10-turn text requests are more likely to have a subsequent turn than 1-turn text requests (see Figure 11). However, if many 1-turn requests are flooding into the system, 10-turn requests can be evicted by LRU because the reuse time is longer.

Our solution: workload-aware reuse-probability-distribution-based KV$ eviction policy. To this end, we propose a workload-aware cache eviction policy that considers the reuse probability distributions of each request category based on the characterizations presented in §3.4. At a high level, at the eviction time, for each block, we calculate a priority—the likelihood of a KV$ being reused in the future based on the profiled reuse probability distribution of its request category (workload). The priority further holistically considers the spatial locality as well as the short lifespan of KV$ blocks. The blocks with the lowest priority are evicted first. Table 2 summarizes our techniques as well as their corresponding observations from §3.

Observations	Techniques
The probability of KV$ reuse follows exponential distributions given a workload (Figure 11 and Figure 15).	① Workload-aware reuse-probability-distribution-based priority estimation.
Spatial locality (Figure 16–18).	② Assign a higher priority to the head KV$ blocks.
Ephemeral KV$ lifespan (Figure 19).	③ Remove frequency when calculating the priority, as well as limiting the time span of probability calculation.

Table 2: *A summary of the main techniques in our workload-aware cache eviction policy based on the findings in §3*

Our caching mechanisms follow existing systems [19, 30, 69, 65]—including asynchronous swapping and pipelined loading, as these techniques have converged to a standard design.

4.2 Workload-aware KV$ cache eviction policy

Existing workload-aware policy. Workload-aware cache eviction policy has been extensively studied in the literature [14, 41, 29, 25, 38, 11, 49]. A representative policy is the Greedy-Dual-Frequency-Size (GDFS) policy [14], which evaluates the priority of each cached object (e.g., KV$) using the following workload-dependent features:

$$\text{Priority} = \text{Clock} + \frac{\text{Frequency} \times \text{Cost}}{\text{Size}} \ (\underline{\text{Existing GDFS}})$$

The `Clock` captures the last access time of an object (similar to LRU), the `Frequency` is the total access count of an object (similar to LFU), the `Size` is the size of the cached object, and `Cost` is the cost to bring the cached content back to the cache. Objects with the lowest priority are evicted first.

While the above GDFS formulation can be retrofitted for calculating the priority of KV$ blocks, we found it does not fully consider the KV$ usage in LLM serving as well as the known probability distributions of requests in different categories. For example, a frequently accessed block may not be reused in the future due to the short lifespan of KV$ blocks, causing a frequently accessed then died block wasting the KV$ space. Besides, a block with higher cost to bring to the cache does not necessarily mean it is more likely to be reused in the future. Instead, we need to bring the more likely to be reused blocks to the cache first. While such workload information is not available in the setup of GDFS, it is readily available in serving LLMs based on our characterization in the previous section.

Our policy. We follow GDFS to calculate the priority of each KV$ block and use the priority to determine the eviction order, but our priority is retrofitted with our characterized

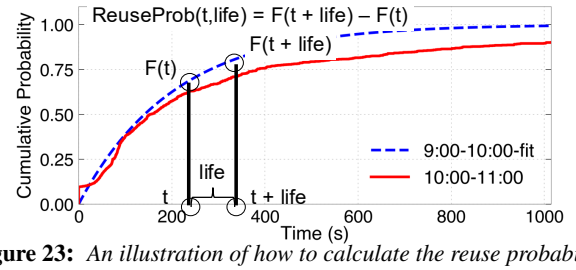


Figure 23: *An illustration of how to calculate the reuse probability of a* KV$ *block given its workload (request category).*

KV$ reuse features:

$$\text{Priority} = (\text{ReuseProb}_w(t, \text{life}), -\text{Offset}) \ (\underline{\text{Workload-aware}})$$

Specifically, the priority is represented with a tuple containing two metrics and we compare the tuple with lexicographical order. The rationale behind the two metrics is:

- **ReuseProb**$_w$ (①) calculates the reuse probability of a KV$ block at a given time t, where t means the time since its last access. For each workload (request category), the reuse probability is calculated using a two-step process. First, we sample a period of the recently accessed data in the background. Second, based on the observations in Figure 15, we fit an exponential distribution to the sampled data, and look up the fitted curve to get the reuse probability. Figure 23 shows an example of how to calculate the reuse probability. We first use the background sampled data from 9:00 a.m. to 10:00 a.m. to fit the cumulative distribution function (CDF) of the reuse probability of a given workload (blue line).

  Note that the fitted curve is close to the real data (red line). Afterward, we can look up the curve to get the reuse probability of a KV$ block in the future. Note that the function further considers the expected lifespan (life) of the block, which is critical considering the short lifespan of KV$ and we describe below.

- **Offset** (②) considers the spatial locality described in §3.4: the priority is reversely proportional to the prefix length of the KV$ block in the entire request.

We don't consider frequency explicitly (③) because we found it can not provide useful information for cache policy due to the short lifespan of KV$ blocks (see Figure 19) even for frequently reused ones. More importantly, we also need to regulate the calculated reuse probability with the lifespan. If not, the fitted reuse probability will return a large probability for a request with long-tailed reuse time distribution for a long time. This contradicts the short KV$ lifespan observation and leads to the inefficient utilization of limited cache space.

The detailed eviction algorithm. Algorithm 24 presents a simplified pseudocode of our workload-aware cache eviction algorithm. Upon eviction, we first iterate through all the cached blocks (line 4–6) and calculate the reuse probability

```
Input:  Bs: cached KV$ blocks;
        Life: expected lifespan of the KV$ blocks;
        H(w): hyper parameters for the workload (w)'s fitted probability.
        distribution that are periodically updated in the background.
Output: B: the evicted KV$ block.
1   CurT = time.time()
2   MinPriority = +∞
3   ChosenBlock = PlaceHolder
4   For b in Bs:
5       Priority = ReuseProb_H(b.w)(CurT - b.cached_t, Life)
6       ▸ Priority is the reuse probability in
          [CurT – b.cached_t, CurT – b.cached_t+ Life)
7       If Priority < MinPriority or (Priority == MinPriority and
                    b.offset > ChosenBlock.offset):
8           MinPriority  = Priority
9           ChosenBlock = b
10  return ChosenBlock
```

Figure 24: *A simplified pseudocode of our workload-aware cache eviction algorithm.*

for each block using the workload ($b.w$). The priority is then compared with the offset of the block to update the block to be evicted (line 7–9).

Performance optimization. One drawback of the above naive algorithm is that it requires computing the reuse probabilities of all KV$ blocks to identify the least reusable candidate. To avoid excessive computation overhead, we observe that all KV$ blocks within each workload are naturally ordered by their last accessed timestamps according to the exponential distribution, so we can avoid computing the reuse probability when comparing blocks within the same workload. Specifically, for each workload, we maintain a priority queue of its KV$ blocks ordered by last accessed time. During eviction, instead of scanning all blocks, we first treat the least-recently-used block of each workload as the candidate for eviction. Afterward, we run the algorithm described in Algorithm 24 to obtain the final evicted block. This reduces the complexity from $O(N)$ to $O(W)$, where W is the number of workloads (typically in the range of tens). As a result, the performance overhead of workload-aware policy is negligible to serving, e.g., we observed a 79 μs policy delay for each eviction, with the aggregated overhead only 1.2 % of the scheduling overhead of a typical serving engine (vLLM [30]).

4.3 Performance evaluation

Testbed. We conducted our evaluations on a server with 8 NVIDIA A800-80GB GPUs. The GPUs are connected via 400 GBps bidirectional NVLink, allowing us to use multiple GPUs (e.g., 4) to serve a single large model with tensor parallelism [39]. The GPUs are connected to the host via PCIe Gen4, offering a bidirectional bandwidth of up to 32 GBps between GPU HBM and the host memory.

Implementations and baselines. Since the state-of-the-art KV$ cache systems that incorporate both CPU and GPU KV$ like CachedAttention [19] and Pensieve [65] are not

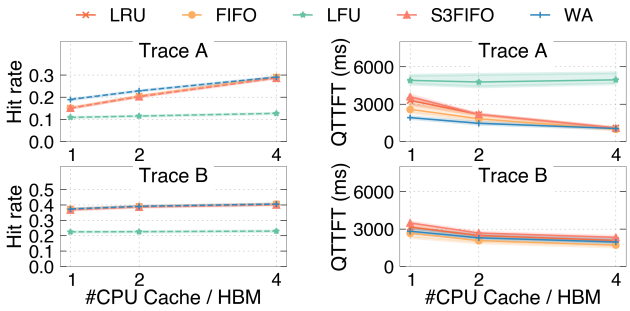


Figure 25: *An analysis of the cache hit ratio and QTTFT with respect to the CPU cache provisioned on Qwen2-7B.*

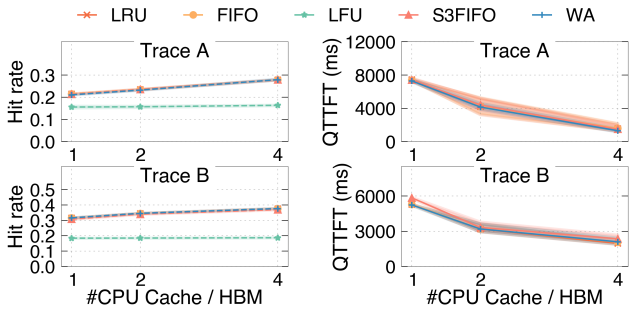


Figure 26: *An analysis of the cache hit ratio and QTTFT with respect to the CPU cache provisioned on Llama2-13B.*

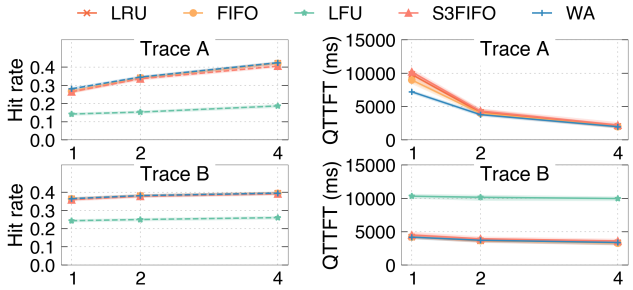


Figure 27: *An analysis of the cache hit ratio and QTTFT with respect to the CPU cache provisioned on Llama3-70B.*

open-sourced, we implement a CPU-GPU KV$ cache on vLLM [30], the state-of-the-art LLM serving system that adopts GPU KV$ cache by default. We have further integrated the policies and mechanisms from state-of-the-art (see §4.1) to it. We have calibrated that our implementation has a similar performance to state-of-the-art like CachedAttention on the same synthetic workload with a similar testbed.

Besides integrating a CPU KV$ cache on vLLM, we have also implemented the KV$-centric global scheduler described in Mooncake [1], because serving LLMs on multiple instances is common. Specifically, the global scheduler will record the global KV$ cache usage and schedule the request to the instance with cached KV$ whenever possible. Note that our workload-aware caching policy only applies to a single serving instance, extending it to a global layer is left as our future work.

Evaluated models and workloads. We use three representative open-source models for the performance evaluations: Qwen2-7B [9], Llama2-13B [54] and Llama3-70B [15]. The rationale for choosing these models and the differences between them have been discussed in the per-request KV$ size analysis in §3.5. For Qwen2-7B and Llama2-13B models, we use one GPU per serving instance. For Llama3-70B, we use 4 GPUs. We use the Trace A and B described in §3.1 as our evaluated workloads. Since we don't have sufficient GPUs to support the real workload, we scale the traces to fit the processing capability of our testbed using the scaling method that preserves the temporal pattern of the trace [51].

Evaluation methodology on anonymous traces. Even though the traces are anonymous, we can evaluate the system serving performance by constructing inputs that accurately reproduce the original trace's cache hit patterns. Specifically, for each anonymous block without assigned tokens, we randomly generate a token list, ensuring each generated list's hash differs from all other token lists. Moreover, to match the original input's iteration count, we constrain the model using the trace's output length. Finally, we replace the model-generated output with the constructed tokens to preserve the cache patterns of multi-turn requests.

Improved cache hit. Figure 25–27 show the cache hit ratio and serving performance for standard cache policies and our workload-aware policy (WA). We compare our policy with the standard policies like least-recently-used (LRU), first-in-first-out (FIFO), least-frequently-used (LFU) and recent optimized FIFO method (S3-FIFO) [61]. We compare with the vanilla GDFS in Figure 28.

Since the performance depends on cache capacity, we report the hit rates (as well as performance) by varying the provisioned KV$ cache capacity. The capacity is normalized to the GPU HBM size for KV$, i.e., the #CPU Cache / HBM in the figures means the CPU memory used to store the KV$ normalized to the GPU HBM. For example, a #CPU Cache / HBM of 1 means the CPU memory used to store the KV$ is equal to the GPU HBM size. We draw three observations from the results.

First, our policy WA achieves 8.1–23.9 % higher hit rate compared to other baselines, and it has a 1.5–3.9 % higher hit rate compared to the best of other baselines. Our policy is more effective thanks to its knowledge of the workload reuse patterns. Second, the effectiveness of WA decreases when the cache capacity increases, this is as expected because a larger cache can tolerate a less effective policy. Finally, WA is more effective on Trace A than Trace B, this is due to two reasons. (1) Trace B has little workload information (> 99 % of its requests are 1-turn requests). Thus, our fitted exponential distribution-based policy essentially falls back to a normal LRU. (2) Trace B requires a relatively small KV$ capacity, so GPU HBM is already close to optimal except for LFU. The poor hit rate of LFU comes from the fact that the most

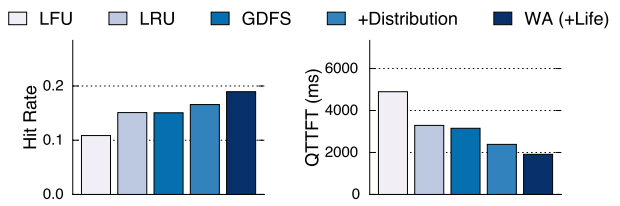


Figure 28: *An ablation study on Qwen2-7B with #CPU Cache / HBM is 1.*

frequently accessed blocks may die early thus polluting the KV$ cache, as we have extensively discussed before.

Improved performance. The improved cache hit ratio leads to a significant performance improvement in the serving performance. Figure 25–27 also reports the queued TTFT (QTTFT) when serving different traces, i.e., the TTFT plus the time when waiting for GPU to process the request. We can see that WA achieves 28.3–41.9% QTTFT reduction thanks to the improved KV$ cache hit rate.

Ablation study. Figure 28 conducts an ablation study on the Qwen2-7B model on Trace A with different policies. Other models share a similar trend. First, we can see that using a distribution-based approach improves the hit rate by 1 % (with technique ① in Table 2). Considering the short lifespan of KV$ with the life regulator further increases the hit rate by 2.4 % (③). Note that we omit describing the locality optimization ② as it is applied to all the policies.

Discussion: fairness issue. Like all existing systems (and their policies), a malicious client can monopolize prefix cache by sending a large volume of requests with long identical prefixes, thus causing fairness issues. Ensuring fairness is an orthogonal topic to cache policy and can be addressed via a co-design of LLM serving strategies like FairRide [47, 52].

5 Related Work

Reuse KV$ across requests via KV$ cache. Reusing KV$ for accelerating LLM serving has been widely studied [30, 63, 19, 20, 69, 62] and used in commercial LLM serving [42, 21, 4, 1]. Currently, production systems only treat cache hits by prefix matches [63, 69, 19, 42, 21, 4, 1], because it preserves the original algorithm of model inference with no accuracy loss. A number of studies [20, 62, 23] have studied methods for non-prefix KV$ cache. We can revisit our characterization once they have matured and been deployed in production.

Other KV$-related optimizations. Emerging works[59, 68, 33, 13] propose runtime KV$ compression and deletion methods to reduce KV$ size. StreamingLLM [59] keeps only a finite attention window for recent tokens combined with a few initial tokens. H2O [68] only involves tokens that contribute most to the attention score for inference. InfiniGen [33] speculatively selects tokens critical for attention scores and drops others. KVQuant [22] enables 3-bit KV cache quantization with minimal perplexity degradation for large language models, achieving fast million-length context inference. These methods require fewer KV$ per-request, albeit with accuracy degradation. Our study is compatible with these works: if an uncompressed/full KV$ is reusable, a compressed or deleted version would also be reusable.

Optimizing caching policies. Optimizing caching has long been studied in the literature, in general-purpose caching policies [41, 29, 28, 53, 32, 25, 38, 71, 10, 24, 16, 11, 49, 67, 14] or specific domains [60, 12, 50, 17]. We continue this line of research, and leverage our characterized KV$ reuse properties for optimizing KV$ cache policies.

Optimizing LLM serving. We continue the line of research in optimizing the performance of LLM serving systems [70, 57, 31, 34, 64, 46, 18, 66], with a particular focus on characterizing serving workloads for KV$ cache. Our work is orthogonal to optimizations other than KV$ cache.

6 Conclusion

We present the first systematic and in-depth characterization of production serving workloads for KV$ cache systems. Our study reveals several important findings that were not captured by synthetic workloads used in previous studies. Based on these findings, we further propose a workload-aware cache eviction policy that improves upon existing workload-agnostic policies like LRU and LFU. We believe our work, as a starting point, can benefit both the research and industry fields in improving current and future LLM serving systems with a workload-driven approach.

Acknowledgment

We would like to thank ATC reviewers and our shepherd Lei Zhang for their insightful feedback. We sincerely thank Guanhu Wang for helping us with the collection of the trace data. We also thank Xiating Xie for proofreading our paper. We thank Xiangfan Wu for helping us correct the presentation about hashing the token IDs. This work was supported in part by the National Natural Science Foundation of China (No. 62202291, 62272291, 62132014) and the Fundamental Research Funds for the Central Universities. This work was also supported by a research grant from Alibaba Group through the Alibaba Innovative Research Program.

References

[1] Mooncake trace. https://github.com/kvcache-ai/Mooncake/blob/main/mooncake_trace.jsonl, 2025.

[2] Sharegpt. https://sharegpt.com, 2025.

[3] Ainslie, J., Lee-Thorp, J., de Jong, M., Zemlyanskiy, Y., Lebrón, F., and Sanghai, S. GQA: training generalized multi-query transformer models from multi-head checkpoints. In *Proceedings of the 2023 Conference on Empirical*

Methods in Natural Language Processing, EMNLP 2023, Singapore, December 6-10, 2023 (2023), H. Bouamor, J. Pino, and K. Bali, Eds., Association for Computational Linguistics, pp. 4895–4901.

[4] ANTHROPIC. Claude api. https://www.anthropic.com/api, 2025.

[5] ANTHROPIC. Prompt caching. https://docs.anthropic.com/en/docs/build-with-claude/prompt-caching, 2025.

[6] ANTHROPIC. Prompt library. https://docs.anthropic.com/en/prompt-library/library, 2025.

[7] ATIKOGLU, B., XU, Y., FRACHTENBERG, E., JIANG, S., AND PALECZNY, M. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE joint international conference on Measurement and Modeling of Computer Systems* (2012), pp. 53–64.

[8] AUMASSON, J., AND BERNSTEIN, D. J. Siphash: A fast short-input PRF. In *Progress in Cryptology - INDOCRYPT 2012, 13th International Conference on Cryptology in India, Kolkata, India, December 9-12, 2012. Proceedings* (2012), S. D. Galbraith and M. Nandi, Eds., vol. 7668 of *Lecture Notes in Computer Science*, Springer, pp. 489–508.

[9] BAI, J., BAI, S., CHU, Y., CUI, Z., DANG, K., DENG, X., FAN, Y., GE, W., HAN, Y., HUANG, F., HUI, B., JI, L., LI, M., LIN, J., LIN, R., LIU, D., LIU, G., LU, C., LU, K., MA, J., MEN, R., REN, X., REN, X., TAN, C., TAN, S., TU, J., WANG, P., WANG, S., WANG, W., WU, S., XU, B., XU, J., YANG, A., YANG, H., YANG, J., YANG, S., YAO, Y., YU, B., YUAN, H., YUAN, Z., ZHANG, J., ZHANG, X., ZHANG, Y., ZHANG, Z., ZHOU, C., ZHOU, J., ZHOU, X., AND ZHU, T. Qwen technical report. *arXiv preprint arXiv:2309.16609* (2023).

[10] BANSAL, S., AND MODHA, D. S. CAR: clock with adaptive replacement. In *Proceedings of the FAST '04 Conference on File and Storage Technologies, March 31 - April 2, 2004, Grand Hyatt Hotel, San Francisco, California, USA* (2004), C. Thekkath, Ed., USENIX, pp. 187–200.

[11] BECKMANN, N., CHEN, H., AND CIDON, A. LHD: improving cache hit rate by maximizing hit density. In *15th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2018, Renton, WA, USA, April 9-11, 2018* (2018), S. Banerjee and S. Seshan, Eds., USENIX Association, pp. 389–403.

[12] BERG, B., BERGER, D. S., MCALLISTER, S., GROSOF, I., GUNASEKAR, S., LU, J., UHLAR, M., CARRIG, J., BECKMANN, N., HARCHOL-BALTER, M., AND GANGER, G. R. The CacheLib caching engine: Design and experiences at scale. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)* (Nov. 2020), USENIX Association, pp. 753–768.

[13] CAI, Z., ZHANG, Y., GAO, B., LIU, Y., LIU, T., LU, K., XIONG, W., DONG, Y., CHANG, B., HU, J., AND XIAO, W. Pyramidkv: Dynamic KV cache compression based on pyramidal information funneling. *CoRR abs/2406.02069* (2024).

[14] CHERKASOVA, L. *Improving WWW proxies performance with greedy-dual-size-frequency caching policy*. Hewlett-Packard Laboratories Palo Alto, CA, USA, 1998.

[15] DUBEY, A., JAUHRI, A., PANDEY, A., KADIAN, A., AL-DAHLE, A., LETMAN, A., MATHUR, A., SCHELTEN, A., YANG, A., FAN, A., GOYAL, A., HARTSHORN, A., YANG, A., MITRA, A., SRAVANKUMAR, A., KORENEV, A., HINSVARK, A., RAO, A., ZHANG, A., RODRIGUEZ, A., GREGERSON, A., SPATARU, A., ROZIÈRE, B., BIRON, B., TANG, B., CHERN, B., CAUCHETEUX, C., NAYAK, C., BI, C., MARRA, C., MCCONNELL, C., KELLER, C., TOURET, C., WU, C., WONG, C., FERRER, C. C., NIKOLAIDIS, C., ALLONSIUS, D., SONG, D., PINTZ, D., LIVSHITS, D., ESIOBU, D., CHOUDHARY, D., MAHAJAN, D., GARCIA-OLANO, D., PERINO, D., HUPKES, D., LAKOMKIN, E., AL-BADAWY, E., LOBANOVA, E., DINAN, E., SMITH, E. M., RADENOVIC, F., ZHANG, F., SYNNAEVE, G., LEE, G., ANDERSON, G. L., NAIL, G., MIALON, G., PANG, G., CUCURELL, G., NGUYEN, H., KOREVAAR, H., XU, H., TOUVRON, H., ZAROV, I., IBARRA, I. A., KLOUMANN, I. M., MISRA, I., EVTIMOV, I., COPET, J., LEE, J., GEFFERT, J., VRANES, J., PARK, J., MAHADEOKAR, J., SHAH, J., VAN DER LINDE, J., BILLOCK, J., HONG, J., LEE, J., FU, J., CHI, J., HUANG, J., LIU, J., WANG, J., YU, J., BITTON, J., SPISAK, J., PARK, J., ROCCA, J., JOHNSTUN, J., SAXE, J., JIA, J., ALWALA, K. V., UPASANI, K., PLAWIAK, K., LI, K., HEAFIELD, K., STONE, K., AND ET AL. The llama 3 herd of models. *CoRR abs/2407.21783* (2024).

[16] EINZIGER, G., EYTAN, O., FRIEDMAN, R., AND MANES, B. Lightweight robust size aware cache management. *ACM Trans. Storage 18*, 3 (2022), 27:1–27:23.

[17] FUERST, A., AND SHARMA, P. Faascache: keeping serverless computing alive with greedy-dual caching. In *ASPLOS '21: 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Virtual Event, USA, April 19-23, 2021* (2021), T. Sherwood, E. D. Berger, and C. Kozyrakis, Eds., ACM, pp. 386–400.

[18] GAO, B., HE, Z., SHARMA, P., KANG, Q., JEVDJIC, D., DENG, J., YANG, X., YU, Z., AND ZUO, P. Cost-Efficient large language model serving for multi-turn conversations with CachedAttention. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)* (Santa Clara, CA, July 2024), USENIX Association, pp. 111–126.

[19] GAO, B., HE, Z., SHARMA, P., KANG, Q., JEVDJIC, D., DENG, J., YANG, X., YU, Z., AND ZUO, P. Cost-efficient large language model serving for multi-turn conversations with cachedattention. In *Proceedings of the 2024 USENIX Annual Technical Conference, USENIX ATC 2024, Santa Clara, CA, USA, July 10-12, 2024* (2024), S. Bagchi and Y. Zhang, Eds., USENIX Association, pp. 111–126.

[20] GIM, I., CHEN, G., LEE, S., SARDA, N., KHANDELWAL, A., AND ZHONG, L. Prompt cache: Modular attention reuse for low-latency inference. In *Proceedings of the Seventh Annual Conference on Machine Learning and Systems, MLSys 2024, Santa Clara, CA, USA, May 13-16, 2024* (2024), P. B. Gibbons, G. Pekhimenko, and C. D. Sa, Eds., mlsys.org.

[21] GOOGLE. Gemini api. https://ai.google.dev/api, 2025.

[22] HOOPER, C., KIM, S., MOHAMMADZADEH, H., MAHONEY, M. W., SHAO, Y. S., KEUTZER, K., AND GHOLAMI, A. Kvquant: Towards 10 million context length LLM inference with KV cache quantization. In *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024* (2024), A. Globersons, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. M. Tomczak, and C. Zhang, Eds.

[23] HU, J., HUANG, W., WANG, H., WANG, W., HU, T., ZHANG, Q., FENG, H., CHEN, X., SHAN, Y., AND XIE, T. Epic: Efficient position-independent context caching for serving large language models, 2024.

[24] JIANG, S., CHEN, F., AND ZHANG, X. Clock-pro: An effective improvement of the CLOCK replacement. In *Proceedings of the 2005 USENIX Annual Technical Conference, April 10-15, 2005, Anaheim, CA, USA* (2005), USENIX, pp. 323–336.

[25] JIANG, S., AND ZHANG, X. LIRS: an efficient low interreference recency set replacement policy to improve buffer cache performance. In *Proceedings of the International Conference on Measurements and Modeling of Computer Systems, SIGMETRICS 2002, June 15-19, 2002, Marina Del Rey, California, USA* (2002), R. R. Muntz, M. Martonosi, and E. de Souza e Silva, Eds., ACM, pp. 31–42.

[26] JIN, C., ZHANG, Z., JIANG, X., LIU, F., LIU, X., LIU, X., AND JIN, X. Ragcache: Efficient knowledge caching for retrieval-augmented generation. *CoRR abs/2404.12457* (2024).

[27] JIN, S., LIU, X., ZHANG, Q., AND MAO, Z. M. Compute or load KV cache? why not both? *CoRR abs/2410.03065* (2024).

[28] JOHNSON, T., AND SHASHA, D. E. 2q: A low overhead high performance buffer management replacement algorithm. In *VLDB'94, Proceedings of 20th International Conference on Very Large Data Bases, September 12-15, 1994, Santiago de Chile, Chile* (1994), J. B. Bocca, M. Jarke, and C. Zaniolo, Eds., Morgan Kaufmann, pp. 439–450.

[29] KAREDLA, R., LOVE, J. S., AND WHERRY, B. G. Caching strategies to improve disk system performance. *Computer 27*, 3 (1994), 38–46.

[30] KWON, W., LI, Z., ZHUANG, S., SHENG, Y., ZHENG, L., YU, C. H., GONZALEZ, J., ZHANG, H., AND STOICA, I. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles, SOSP 2023, Koblenz, Germany, October 23-26, 2023* (2023), J. Flinn, M. I. Seltzer, P. Druschel, A. Kaufmann, and J. Mace, Eds., ACM, pp. 611–626.

[31] KWON, W., LI, Z., ZHUANG, S., SHENG, Y., ZHENG, L., YU, C. H., GONZALEZ, J., ZHANG, H., AND STOICA, I. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles, SOSP 2023, Koblenz, Germany, October 23-26, 2023* (2023), J. Flinn, M. I. Seltzer, P. Druschel, A. Kaufmann, and J. Mace, Eds., ACM, pp. 611–626.

[32] LEE, D., CHOI, J., KIM, J., NOH, S. H., MIN, S. L., CHO, Y., AND KIM, C. LRFU: A spectrum of policies that subsumes the least recently used and least frequently used policies. *IEEE Trans. Computers 50*, 12 (2001), 1352–1361.

[33] LEE, W., LEE, J., SEO, J., AND SIM, J. Infinigen: Efficient generative inference of large language models with dynamic KV cache management. In *18th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2024, Santa Clara, CA, USA, July 10-12, 2024* (2024), A. Gavrilovska and D. B. Terry, Eds., USENIX Association, pp. 155–172.

[34] LI, Z., ZHENG, L., ZHONG, Y., LIU, V., SHENG, Y., JIN, X., HUANG, Y., CHEN, Z., ZHANG, H., GONZALEZ, J. E., AND STOICA, I. AlpaServe: Statistical multiplexing with model parallelism for deep learning serving. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)* (Boston, MA, July 2023), USENIX Association, pp. 663–679.

[35] LIN, C., HAN, Z., ZHANG, C., YANG, Y., YANG, F., CHEN, C., AND QIU, L. Parrot: Efficient serving of llm-based applications with semantic variable. In *18th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2024, Santa Clara, CA, USA, July 10-12, 2024* (2024), A. Gavrilovska and D. B. Terry, Eds., USENIX Association, pp. 929–945.

[36] LIU, A., FENG, B., XUE, B., WANG, B., WU, B., LU, C., ZHAO, C., DENG, C., ZHANG, C., RUAN, C., ET AL. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437* (2024).

[37] LIU, Y., LI, H., CHENG, Y., RAY, S., HUANG, Y., ZHANG, Q., DU, K., YAO, J., LU, S., ANANTHANARAYANAN, G., MAIRE, M., HOFFMANN, H., HOLTZMAN, A., AND JIANG, J. Cachegen: KV cache compression and streaming for fast large language model serving. In *Proceedings of the ACM SIGCOMM 2024 Conference, ACM SIGCOMM 2024, Sydney, NSW, Australia, August 4-8, 2024* (2024), ACM, pp. 38–56.

[38] MEGIDDO, N., AND MODHA, D. S. ARC: A self-tuning, low overhead replacement cache. In *Proceedings of the FAST '03 Conference on File and Storage Technologies, March 31 - April 2, 2003, Cathedral Hill Hotel, San Francisco, California, USA* (2003), J. Chase, Ed., USENIX.

[39] NARAYANAN, D., SHOEYBI, M., CASPER, J., LEGRESLEY, P., PATWARY, M., KORTHIKANTI, V., VAINBRAND, D., KASHINKUNTI, P., BERNAUER, J., CATANZARO, B., ET AL. Efficient large-scale language model training on gpu clusters using megatron-lm. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (2021), pp. 1–15.

[40] NISHTALA, R., FUGAL, H., GRIMM, S., KWIATKOWSKI, M., LEE, H., LI, H. C., MCELROY, R., PALECZNY, M., PEEK, D., SAAB, P., STAFFORD, D., TUNG, T., AND VENKATARAMANI, V. Scaling memcache at facebook. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)* (Lombard, IL, Apr. 2013), USENIX Association, pp. 385–398.

[41] O'NEIL, E. J., O'NEIL, P. E., AND WEIKUM, G. The LRU-K page replacement algorithm for database disk buffering. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, Washington, DC, USA, May 26-28, 1993* (1993), P. Buneman and S. Jajodia, Eds., ACM Press, pp. 297–306.

[42] OpenAI. Openai developer platform. https://platform.openai.com/docs/overview.

[43] OpenAI. Chatgpt. https://chatgpt.com, 2024.

[44] OpenAI. Introducing openai o1. https://openai.com/o1/, 2025.

[45] OpenAI. Prompt caching. https://platform.openai.com/docs/guides/prompt-caching, 2025.

[46] Patel, P., Choukse, E., Zhang, C., Shah, A., Goiri, Í., Maleki, S., and Bianchini, R. Splitwise: Efficient generative LLM inference using phase splitting. In *51st ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2024, Buenos Aires, Argentina, June 29 - July 3, 2024* (2024), IEEE, pp. 118–132.

[47] Pu, Q., Li, H., Zaharia, M., Ghodsi, A., and Stoica, I. Fairride: Near-optimal, fair cache sharing. In *13th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2016, Santa Clara, CA, USA, March 16-18, 2016* (2016), K. J. Argyraki and R. Isaacs, Eds., USENIX Association, pp. 393–406.

[48] Rock, D. Your brain at work: Strategies for overcoming distraction, regaining focus, and working smarter all day long. *Journal of Behavioral Optometry 21*, 5 (2010), 130.

[49] Rodriguez, L. V., Yusuf, F. B., Lyons, S., Paz, E., Rangaswami, R., Liu, J., Zhao, M., and Narasimhan, G. Learning cache replacement with CACHEUS. In *19th USENIX Conference on File and Storage Technologies, FAST 2021, February 23-25, 2021* (2021), M. K. Aguilera and G. Yadgar, Eds., USENIX Association, pp. 341–354.

[50] Roy, R. B., Patel, T., and Tiwari, D. Icebreaker: warming serverless functions better with heterogeneity. In *ASPLOS '22: 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, 28 February 2022 - 4 March 2022* (2022), B. Falsafi, M. Ferdman, S. Lu, and T. F. Wenisch, Eds., ACM, pp. 753–767.

[51] Sajal, S. M., Hasan, R., Zhu, T., Urgaonkar, B., and Sen, S. Tracesplitter: a new paradigm for downscaling traces. In *EuroSys '21: Sixteenth European Conference on Computer Systems, Online Event, United Kingdom, April 26-28, 2021* (2021), A. Barbalace, P. Bhatotia, L. Alvisi, and C. Cadar, Eds., ACM, pp. 606–619.

[52] Sheng, Y., Cao, S., Li, D., Zhu, B., Li, Z., Zhuo, D., Gonzalez, J. E., and Stoica, I. Fairness in serving large language models. In *18th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2024, Santa Clara, CA, USA, July 10-12, 2024* (2024), A. Gavrilovska and D. B. Terry, Eds., USENIX Association, pp. 965–988.

[53] Smaragdakis, Y., Kaplan, S. F., and Wilson, P. R. EELRU: simple and effective adaptive page replacement. In *Proceedings of the 1999 ACM SIGMETRICS international conference on Measurement and modeling of computer systems, Atlanta, Georgia, USA, May 1-4, 1999* (1999), D. A. Menascé and C. Williamson, Eds., ACM, pp. 122–133.

[54] Touvron, H., Lavril, T., Izacard, G., Martinet, X., Lachaux, M., Lacroix, T., Rozière, B., Goyal, N., Hambro, E., Azhar, F., Rodriguez, A., Joulin, A., Grave, E., and Lample, G. Llama: Open and efficient foundation language models. *CoRR abs/2302.13971* (2023).

[55] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. Attention is all you need. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA* (2017), I. Guyon, U. von Luxburg, S. Bengio, H. M. Wallach, R. Fergus, S. V. N. Vishwanathan, and R. Garnett, Eds., pp. 5998–6008.

[56] Wang, X., Wang, Z., Liu, J., Chen, Y., Yuan, L., Peng, H., and Ji, H. Mint: Evaluating llms in multi-turn interaction with tools and language feedback. *arXiv preprint arXiv:2309.10691* (2023).

[57] Wu, B., Liu, S., Zhong, Y., Sun, P., Liu, X., and Jin, X. Loongserve: Efficiently serving long-context large language models with elastic sequence parallelism. *CoRR abs/2404.09526* (2024).

[58] Wu, Q., Bansal, G., Zhang, J., Wu, Y., Li, B., Zhu, E., Jiang, L., Zhang, X., Zhang, S., Liu, J., et al. Autogen: Enabling next-gen llm applications via multi-agent conversations. In *First Conference on Language Modeling* (2024).

[59] Xiao, G., Tian, Y., Chen, B., Han, S., and Lewis, M. Efficient streaming language models with attention sinks. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024* (2024), OpenReview.net.

[60] Yang, J., Yue, Y., and Rashmi, K. V. A large scale analysis of hundreds of in-memory cache clusters at twitter. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)* (Nov. 2020), USENIX Association, pp. 191–208.

[61] Yang, J., Zhang, Y., Qiu, Z., Yue, Y., and Vinayak, R. Fifo queues are all you need for cache eviction. In *Proceedings of the 29th Symposium on Operating Systems Principles* (New York, NY, USA, 2023), SOSP '23, Association for Computing Machinery, p. 130–149.

[62] Yao, J., Li, H., Liu, Y., Ray, S., Cheng, Y., Zhang, Q., Du, K., Lu, S., and Jiang, J. Cacheblend: Fast large language model serving for RAG with cached knowledge fusion. *CoRR abs/2405.16444* (2024).

[63] Ye, L., Tao, Z., Huang, Y., and Li, Y. Chunkattention: Efficient self-attention with prefix-aware kv cache and two-phase partition. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)* (2024), Association for Computational Linguistics, pp. 11608–11620.

[64] Yu, G., Jeong, J. S., Kim, G., Kim, S., and Chun, B. Orca: A distributed serving system for transformer-based generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2022, Carlsbad, CA, USA, July 11-13, 2022* (2022), M. K. Aguilera and H. Weatherspoon, Eds., USENIX Association, pp. 521–538.

[65] YU, L., AND LI, J. Stateful large language model serving with pensieve. *CoRR abs/2312.05516* (2023).

[66] ZHANG, D., WANG, H., LIU, Y., WEI, X., SHAN, Y., CHEN, R., AND CHEN, H. Fast and live model auto scaling with O(1) host caching. *CoRR abs/2412.17246* (2024).

[67] ZHANG, Y., YANG, J., YUE, Y., VIGFUSSON, Y., AND RASHMI, K. V. SIEVE is simpler than LRU: an efficient turn-key eviction algorithm for web caches. In *21st USENIX Symposium on Networked Systems Design and Implementation, NSDI 2024, Santa Clara, CA, April 15-17, 2024* (2024), L. Vanbever and I. Zhang, Eds., USENIX Association.

[68] ZHANG, Z., SHENG, Y., ZHOU, T., CHEN, T., ZHENG, L., CAI, R., SONG, Z., TIAN, Y., RÉ, C., BARRETT, C. W., WANG, Z., AND CHEN, B. H2O: heavy-hitter oracle for efficient generative inference of large language models. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023* (2023), A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine, Eds.

[69] ZHENG, L., YIN, L., XIE, Z., HUANG, J., SUN, C., YU, C. H., CAO, S., KOZYRAKIS, C., STOICA, I., GONZALEZ, J. E., BARRETT, C. W., AND SHENG, Y. Efficiently programming large language models using sglang. *CoRR abs/2312.07104* (2023).

[70] ZHONG, Y., LIU, S., CHEN, J., HU, J., ZHU, Y., LIU, X., JIN, X., AND ZHANG, H. Distserve: Disaggregating prefill and decoding for goodput-optimized large language model serving. In *18th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2024, Santa Clara, CA, USA, July 10-12, 2024* (2024), A. Gavrilovska and D. B. Terry, Eds., USENIX Association, pp. 193–210.

[71] ZHOU, Y., CHEN, Z., AND LI, K. Second-level buffer cache management. *IEEE Trans. Parallel Distributed Syst. 15*, 6 (2004), 505–519.

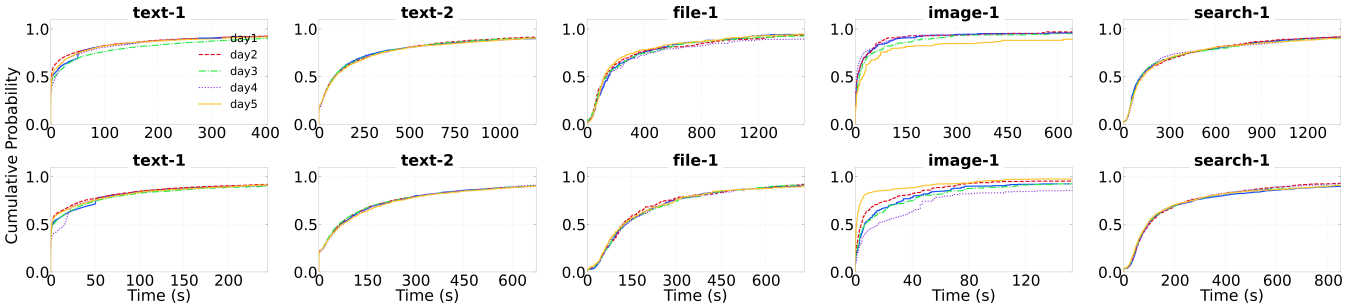


Figure 29: *The evolution of cumulative distribution function of the reuse probability in 5 workdays. The first row: distributions from 9.00 a.m. to 10.00 a.m. The second row: distributions from 10.00 a.m. to 11.00 a.m.*

A Appendix

A.1 Trace results across different days

Figure 29 shows the evolution of cumulative distribution (CDF) function of the reuse probability for each request category in future time window over 5 continuous workdays. We can observe that the reuse probability at 9:00 a.m. to 10:00 a.m. similar between workdays. Therefore, we can analyze the reuse log of last day offline to predict the reuse probability of each category in the next day. However, it is noted that the reuse probability of "image" type is harder to predict than other categories, we infer that this is because image types have coarser reusability granularity compared to text types, and there exists greater variability among different users.

LogCrisp: Fast Aggregated Analysis on Large-scale Compressed Logs by Enabling Two-Phase Pattern Extraction and Vectorized Queries

Junyu Wei[†], Guangyan Zhang[†,*], Junchao Chen[†], Qi Zhou[§]
[†]*Tsinghua University,* [§]*Alibaba Cloud*

Abstract

Cloud providers generate logs at massive scales, often requiring dense compression using log patterns. Meanwhile, aggregated analysis on logs is essential for various applications. However, performing aggregated analysis on highly compressed logs presents two fundamental challenges: 1) it is hard to extract a set of log patterns that have both a global description and high filtering effectiveness; 2) executing full-text queries on numerically encoded data is challenging.

This paper proposes a two-phase pattern extraction paradigm. Such a paradigm decouples messages within patterns into Sketch (global pattern structure) and Specs (local fine-grained pattern specifications). The Sketch is extracted in an offline phase to provide a comprehensive global description, while the Specs are customized in the online phase to enhance pattern filtering effectiveness. Additionally, this paper proposes an efficient prefix/suffix vectorized query algorithm for numerically encoded data, which leverages AVX SIMD instructions to convert full-text queries into high-performance range/point queries.

We implement and integrate all these techniques into a system called LogCrisp, which is evaluated using nearly 7TB of logs from both production environments and public datasets. Experimental results show that LogCrisp achieves an order of magnitude lower analysis latency, 3.8× higher ingestion speed, and an almost identical compression ratio, compared with state-of-the-art works.

1 Introduction

Large cloud providers continuously generate system logs, which can accumulate to the PB scale per day [26, 29, 33, 35]. Cloud logs capture critical system information and user access records. To support error diagnosis, security attack detection, and user behavior profiling [7, 12, 14, 38, 39], it is common to perform aggregated analysis (e.g. counting, summation, getting max/min) on these logs.

*Corresponding author: gyzh@tsinghua.edu.cn

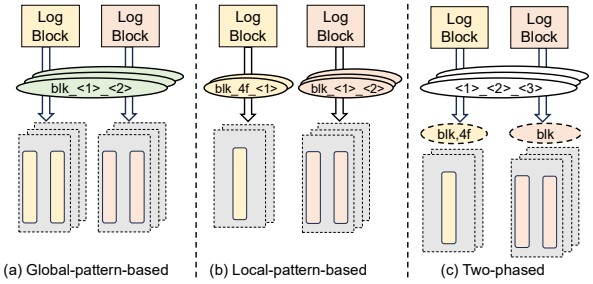


Figure 1: Comparison of different pattern extraction paradigms

In addition, since logs need to be retained for a relatively long time (say 180 days [35]) for audit purposes, existing log storage methods [17, 26, 29, 33–35] propose to compress logs densely using patterns within them. For example, in Hadoop logs, the chunk ID has the pattern of "blk_<hex,3>_<num,2>". During the ingestion process, each variable value (e.g. "blk_4ff_34") will be broken into a group of fragments (e.g. "4ff" and "34") according to the pattern. All fragments occurring at the same place will be encoded as a unit during the compression to achieve a high compression ratio. Patterns can also be used to filter out unrelated units during analysis. For example, when counting logs containing "blk_czf", there is no need to decompress any unit of chunk ID variable, since "czf" is not a three-character hexadecimal value.

However, performing aggregated analysis on highly compressed logs faces two fundamental challenges. The first is the dilemma between global description and filtering effectiveness of log patterns. Pattern-based log compression methods ingest logs as multiple log blocks and compress each block as a compressed file containing a set of units. They thus have a key design choice of extracting patterns. As shown in Figure 1, global-pattern-based methods [29, 35] extract or predefine a set of global patterns shared by all log blocks. On the other hand, local-pattern-based methods [17, 26, 33] extract a set

of patterns for each log block. In order to perform aggregated analysis on multiple log blocks, a global description of each fragment positions is needed. For example, to count how many chunks belong to "4f*" region, we need to perform a prefix query on the *first unit* of chunk ID variables according to the global pattern "blk_<hex,3>_<num,2>". However, global patterns are usually too general to describe the content of each unit precisely, and thus the query process needs to decompress more units because of the limited filtering effectiveness. According to previous work, the analysis latencies of global-pattern-based methods are usually longer than local-pattern-based methods by up to 10× [33].

We thus propose to **extract patterns in two phases** by decoupling messages within patterns clearly. We begin with an *off-line phase* that only extracts the minimum messages necessary to break variables into fragments. We call these messages as *Sketch*. For example, pattern "blk_<hex,3>_<num,2>" has the Sketch "<*>_<*>_<*>". Sketch contains the necessary global description for aggregation and leaves as many messages as possible to be customized for each log block. We then continue with an *on-line phase* when ingesting each log block. This phase extracts other messages of the pattern, such as the constant characters ("blk"), type messages ("hex", "num") and length messages ("3", "2"). We call these messages as *Spec*.

However, to enable two-phase pattern extraction, it is challenging to derive all fragment boundaries before extracting the whole pattern. We thus examine 11,954 fragment boundaries in all 3,259 local patterns extracted by the state-of-the-art method [33] and find over 98% of them are non-alphanumerical (NAU) characters. We thus choose to extract the Sketch according to all NAU characters heuristically. However, the opposite direction does not hold, namely some NAU characters may not be the boundaries and thus variable values originally correspond to the same local pattern may have multiple Sketches. To address this issue, a well-organized Sketch warehouse is maintained for each variable type, enabling efficient selection of the appropriate Sketch during data ingestion.

Besides, we further optimize the on-line ingestion speed by leveraging the total count of units according to the Sketch and generating Spec in a pre-allocated cacheline-aligned manner.

The second fundamental challenge in performing aggregated analysis on compressed logs is the dilemma between numerical encoding format and full-text query semantic. To execute arithmetical aggregation and improve the compression ratio, values in the unit that only contains numerical characters, which takes up 53% of all units, have to be encoded as integers. However, since logs are of text format originally, it is required to support full text queries on real-world logs, i.e. pre/suffix queries on some numerical variables [29, 33]. For example, it is a common practice to count how many chunks has a number starting with "3" (i.e., to query "blk_4FF_3*").

To overcome such a problem, we propose a **vectorized pre/suffix query algorithm** to execute pre/suffix queries directly on integer-encoded units with AVX SIMD instructions. Our key idea is to convert pre/suffix queries into range/point queries which are compatible to numerical encoding format. It is not hard to convert suffix queries since they correspond to querying specific modulo results for all target values. However, converting prefix queries to range queries is more complex, as the resulting range varies with each value. We prove that the corresponding range can be uniquely determined by a squeezing range whose endpoints are both powers of two. Based on this theorem, we: 1) calculate the squeezing range using vectorized shift operations; 2) determine the corresponding range for each value; 3) load each range with an AVX SIMD instruction of conditional blending [10]; 4) perform vectorized range queries accordingly.

Besides, to further reduce analysis latency, we use the AVX SIMD shuffle instruction [10] to construct the indexed bimtap [33], which records intermediate query results for subsequent aggregation and enables efficient merging across different units.

Finally, we implement and integrate all these techniques into a system called LogCrisp. We evaluate LogCrisp on 13 types of logs, nearly 7TB of data in total, including both production logs from our collaborator Alibaba Cloud and public logs used in previous works [18, 29, 33, 35]. We compare LogCrisp with the state-of-the-art works, i.e. CLP [29] and LogGrep [33]. We observe that LogCrisp achieves significantly lower aggregated analysis latency—15.32× and 4.65× faster than CLP and LogGrep, respectively. Additionally, LogCrisp improves ingestion speed by up to 3.8× over LogGrep, while maintaining a compression ratio comparable to CLP (about 1.1×) and LogGrep (about 96%).

The contributions of this paper are threefold:

- We propose a two-phase pattern extraction paradigm that delivers a global description for aggregated analysis while maintaining high filtering effectiveness.
- We propose a prefix/suffix query algorithm that, for the first time, enables vectorized aggregated analysis on highly compressed logs.
- We implement and evaluate LogCrisp, a compressed log storage system that outperforms state-of-the-art methods in analysis latency and ingestion speed.

2 Background and Related Work

Log storage systems target at a high compression ratio, a low analysis latency as well as a high ingestion speed. Table 1 shows four typical log storage systems as well as our work (called LogCrisp) in respect of their compression, ingestion and analysis performance. Besides, not all log storage systems have a global description of each variable, some systems thus can only perform keyword searching on logs.

	ZG [24]	ES [5]	CLP [29]	LG [33]	LC
Compression Ratio	Bad	Bad	Medium	Good	Good
Ingestion Speed	Good	Bad	Good	Medium	Good
Analysis Latency	Bad	Good	Medium	Good	Good
Global Description	No	Yes	Yes	No	Yes

Table 1: Summary of log storage systems. ZG, ES, LG and LC are the abbreviations for Zgrep, Elastic Search, LogGrep and LogCrisp respectively

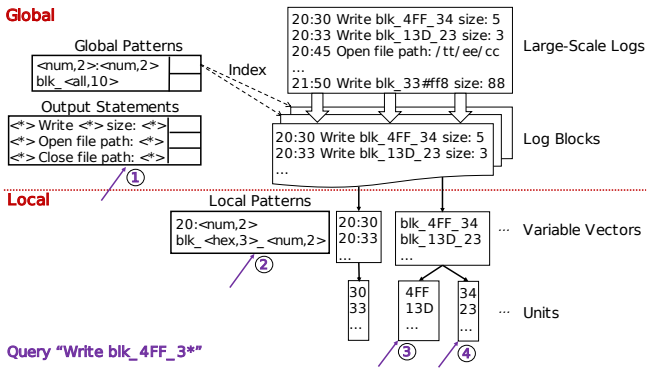


Figure 2: Conceptual diagram of pattern-based log storage

Due to the semi-structured characteristics of logs, pattern-based log storage methods [17, 26, 29, 33–35] are the most common solutions for log storage. Figure 2 shows the conceptual diagram of pattern-based log storage methods, which can be further categorized into global-pattern-based methods and local-pattern-based methods. Both of them need a log parsing process to extract output statements and log variables. In this section, we first introduce the log parsing process and then analyze these two types of pattern-based methods respectively. Finally, we discuss other works related with log storage.

2.1 Log Parsing

Since logs are generated by programs, each log entry can be parsed as an output statement and several variables. This parsing can be done by designating programming standard [6], analysis source code [8, 36] or using a log parser [11, 21]. For example, as shown in Figure 2, output statement *printf("%s Write block: %s with length: %d", time, blk, len)* will print many log entries, each containing three variable values, namely the timestamp, the block number and the write size.

Pattern-based log storage methods are all built based on such output statements and use them to extract log variables as well as locate the to-be-aggregated variables during the analysis. They regard all variable values occurring at the same place of the statement as belonging to the same variable and further extract a *pattern* for each variable to improve the compression ratio and accelerate the analysis.

2.2 Global-pattern-based Methods

Global-pattern-based methods extract or pre-define a global pattern for each variable, which is shared by all log blocks. This pattern is usually very general (e.g. "blk_<all,10>") since it has to describe variable values occurring at all log blocks.

CLP [29] is the state-of-the-art global-pattern-based method. During the ingestion process, CLP extracts variables based on the output statements and breaks them according to the pre-defined global pattern. Such process can be done alongside the ingestion process without any extra data copy. Then CLP reorganizes all fragments of each log entry as an encoded message. Finally, it compresses all encoded messages of the same log block together as a unit and constructs an inverted index to map the global pattern to the unit. During the analysis, it will first query on the output statements to locate to-be-aggregated variables and then query on the corresponding global patterns to decompress part of units based on the inverted index.

CLP achieves a relatively high compression ratio, a high ingestion speed as well as a global description of variables delivered by the global patterns. However, the analysis latency of CLP is sub-optimal in some scenarios [33] due to the limited filtering efficiency of global patterns. Besides, since CLP stores all fragments according to their original order, numerical fragments cannot be processed and aggregated with vectorized instructions.

2.3 Local-pattern-based Methods

Local-pattern-based methods propose to extract a local pattern for each variable within a log block. Compared with global pattern, such pattern will be more concrete and describe the content of each unit precisely (e.g. "blk_<hex,3>_<num,2>") since it only needs to describe variables within this block.

LogGrep [33] is the state-of-the-art local-pattern-based method. During the ingestion process, LogGrep first uses the output statement to extract all variables like CLP and then copies all variables of the same type into a variable vector. Afterward, LogGrep extracts a local pattern for each vector. Then it breaks each variable in the vector into fragments according to the local pattern and stores all fragments occurring at the same position within a unit. Finally, it compresses each unit individually. During the analysis process, LogGrep first locates to-be-aggregated variables like CLP (step ① in Figure 2) and filters units based on the local patterns (step ②). It then only decompresses and searches within a small part of units (step ③ and step ④).

LogGrep can partition log blocks into fine-grained units to achieve a low query latency without compromising the compression ratio. Such a design is also necessary to process numerical units with vectorized instructions together. However, due to the differences of local patterns, LogGrep does not have a global description. It thus can only support key-

word searching like Linux grep. Besides, when processing numerical units with vectorized instructions, full-text queries cannot be supported directly. LogGrep thus compromises to execute ordinary text query on numerical units. Thirdly, LogGrep's lack of prior knowledge about the patterns requires to execute full pattern extraction process for each log block. Consequently, LogGrep's ingestion speed is 2-5× slower than that of CLP.

2.4 Other Related Works

Except for CLP and LogGrep, there are still some related log storage methods. Besides, there are some works that can process directly on column-oriented storage in database or on compressed text data.

Log storage methods. Besides pattern-based methods, some other methods, such as Cowic [25] and LogArchive [16], are designed to compress log. They choose to compress by clustering similar logs together instead of breaking the variables and encoding similar fragments with tailored methods. As a result, their compression ratio is lower than pattern-based methods [35].

A common practice to execute aggregated analysis is using log management tools, such as ElasticSearch [5], Splunk [20], Scalyr [19] and Loki [22]. These tools can manage large-scale logs and are employed in production environments. However, they can not process compressed logs, which hinders them to achieve a low storage cost.

Processing directly on compressed data. Processing directly on compressed data includes two basic types: light-weight encoding methods [3, 15, 27, 28, 30, 31, 40, 41] and partial decompression methods [1, 2, 23, 29, 33] . Light-weight encoding methods do not combine with heavy byte-oriented compression method as its packing method, their compression ratios are lower than partial decompression methods [29, 33] and their supported operations are limited.

Pattern-based log storage methods can be categorized as partial decompression methods but they are different from other methods used in database [1, 2, 23] in respect that they are designed on compressed text logs. They thus need to support full-text queries on the original data.

3 Two-phase Pattern Extraction

A common global description shared by all log blocks can support aggregating fragments within the variable as well as improves the ingestion speed. However, to maintain the filtering effectiveness of patterns, local details for each log block are also needed.

We thus propose to decouple messages within the pattern and extract them in a two-phase manner. The primary challenge of this design is how to minimize the messages extracted globally so as to 1) ensure they are sufficient to describe each fragment's position; 2) maximize locally customized messages to enhance filtering effectiveness.

	Pattern Count	Boundary Count	NAU Boundary Rate
Hadoop	118	570	97.37%
HadoopL	97	358	99.16%
Hive	470	2117	98.87%
OpenStackC	170	850	100.00%
Spark	316	1416	99.44%
Thunderbird	1962	5691	98.84%
Windows	126	952	98.84%

Table 2: Proportion of non-alphanumerical boundaries

In this section, we first discuss the clear decoupling of messages within patterns and then present the two-phase pattern extraction paradigm. Additionally, we explore a cache-line-aware optimization based on this paradigm to further boost ingestion speed.

3.1 Pattern Message Decoupling

To locate and deliver a global description of fragments, we must at least know their boundaries. To investigate fragment boundary characteristics in local patterns, we extract 3,259 local patterns from 7 log types using LogGrep [33]. Among these patterns, which include 11,954 fragment boundaries in total, we find that over 98% of them are non-alphanumerical (NAU) characters, as shown in Table 2. This trend likely arises from two factors: 1) NAU characters often separate semantic components within log variables, making them natural boundaries; 2) State-of-the-art local-pattern extraction methods (e.g., [17, 26, 33, 35]) typically use NAU characters to split variables during pattern extraction.

Based on this observation, we define all NAU characters within a variable value as its Sketch. For example, in Figure 3, the value /pjhe/35/part-0009" has a Sketch /<*>/<*>/<*>-<*>". However, this observation is not bidirectional, namely not all NAU characters function as boundaries. Our method therefore extracts multiple Sketches for each variable type.

To overcome such problem, we organize all Sketches we found for a variable in a Sketch warehouse and represent each Sketch in a compactly: we calculate a 64-bit integer based on all NAU characters shown within the Sketch [32]. Such an integer serves as a fingerprint to merge common Sketches during the extraction as well as to determine the corresponding Sketch of an ingested variable value immediately. To accelerate such process, we build a hashing-based searching index on the Sketch warehouse.

3.2 Two-phase Extraction Paradigm

Pattern message decoupling enables simultaneous global description and local detail maintenance. We therefore propose to locate fragments using pre-extracted Sketches from an off-line phase and extract additional messages, referred to as

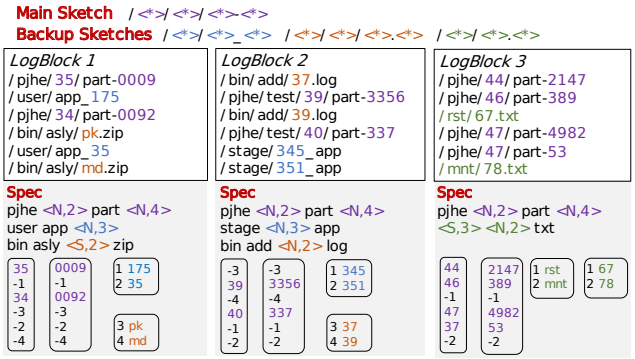


Figure 3: An example of pattern message decoupling

Specs, during ingestion. During analysis, units are filtered by combining both Sketch and Spec information.

Off-line phase. The extraction algorithm's pseudocode is detailed in Algorithm 1. First, we extract the Sketch from each value's NAU characters (line 5). We then merge identical Sketches to build the Sketch warehouse (line 6). The most common Sketch in the warehouse is designated as the main Sketch (line 8), with others classified as backup Sketches.

Algorithm 1 Sketch extraction algorithm

1: Variable value sample set $\mathbb{S}$
2: Main sketch Ψ
3: Sketch warehouse $\mathbb{B} = \emptyset$
4: **for all** s in $\mathbb{S}$ **do**
5: $\quad \mathcal{K} \leftarrow getSketch(s)$
6: $\quad \mathbb{B} \leftarrow \mathbb{B} \cap \mathcal{K}$
7: **end for**
8: Ψ = the most common sketch in $\mathbb{B}$
9: Output Ψ, $\mathbb{B}$

On-line phase. We extract other details, namely Specs, for each Sketch when ingesting logs during the online phase. After parsing variables with output statements, we first calculate the Sketch fingerprint of each incoming variable value based on all NAU characters in that value. We then search for this fingerprint in the corresponding Sketch warehouse. If a Sketch matches, we locate all fragments accordingly. If not, the incoming log entry is stored as an outlier.

We extract Specs in a piggybacked manner alongside ingestion after fragment location. Specifically: 1) (e.g., pjhe" in the Spec of LogBlock1" in Figure 3): We record the latest fragment and compare it with subsequent ingested fragments. If all fragments are constant by the end of ingestion, this value is treated as part of the pattern rather than creating a new unit. 2) for type messages (e.g., N" in <N,2>" in the Spec of LogBlock1" indicates 35" and "34" are numbers): We record type messages using a 6-bit vector [34], updating corresponding bits when new character types occur. 3) for length messages (e.g., 4" in <N,4>" in the Spec of LogBlock1" represents the maximum length of 0009" and "0092" as four): We track the maximum fragment length within a unit, updating it when a longer fragment is ingested.

Since each variable may have multiple Sketches, we store fragments of different Sketches in separate units. The main Sketch's units record the incoming order of variable values that conform to backup Sketches. Each value is assigned a number determined first by the global order of backup Sketches and then by its ingestion sequence. Each fragment unit of the main Sketch stores not only fragment values but also a placeholder. For positions corresponding to backup Sketch values, this placeholder holds the negative of the assigned number.

Used for filtering. During the analysis, if the to-be-aggregated fragment is designated, we directly execute aggregation on the corresponding unit in each log block. If the unit corresponds to constant characters, we return the constant value within the Spec rather than decompress units. If the to-be-aggregated fragment is not designated, we combine Specs in all log blocks with their corresponding Sketches to form patterns and then match the query keyword against these patterns to determine which unit to decompress, as shown in prior work [33]. If a fragment within the backup Sketches is designated or the keyword matches a variable value conforming to backup Sketches, we check one of the main Sketch units to determine its original incoming order based on the assigned number.

A concrete example. We provide an example of pattern decoupling in Figure 3. The variable from "LogBlock 1" includes six variable values, stored in four units: two for fragments in the main Sketch and two for backup Sketches. We also store placeholders for backup Sketch variables in the main Sketch's unit. In this example, the customization feature of Spec is demonstrated by: 1) *Selective Sketch Adoption*: Different log blocks can choose Sketches from the Sketch warehouse. For example, "LogBlock 3" has variable values conforming to the third backup Sketch, while the other two log blocks use values conforming to the first and second backup Sketches. 2) *Spec Variability Across Log Blocks*: Specs for the same Sketch differ in constant, type, and maximum length attributes across log blocks. For example, both "LogBlock 1" and "LogBlock 2" use the first and second backup Sketches, but their Specs are distinct.

3.3 Cache-friendly Ingestion Optimization

Based on two-phase extraction paradigm, pre-extracted Sketches offer position messages for fragments, allowing us to extract Specs in a pre-allocated and cache-friendly manner. We thus introduce two optimizations to further enhance the ingestion speed.

Cache-friendly position batching. During ingestion, pre-extracted Sketches locate fragments, but Specs are determined only at ingestion completion. Specs dictate compres-

sion strategies (e.g., units with constant fragments require no compression). Thus, we need to record fragment positions temporarily.

We represent each fragment position with a (offset, length) pair and batch positions for the same unit. Each batch is sized at 64 bytes (aligned with CPU cache lines), storing up to seven sequential position pairs and a 4-byte pointer to the next batch for the unit. This design enables the compressor to retrieve seven sequential positions for a unit in a single memory access by leveraging CPU cache locality.

Cache-friendly metadata maintaining. Each unit's metadata includes the head/tail of the position batch list, the latest ingested fragment, and type/length attributes for generating Specs. Prior approaches [33–35] use hash tables or tree-based indexes for metadata management, supporting dynamic unit insertion during ingestion.

By contrast, our method pre-allocates and stores metadata for all possible units (based on pre-extracted Sketches) in contiguous memory. This allows seamless, cache-friendly updates to all relevant metadata as each log entry is ingested.

4 Vectorized Query Processing

Pattern-based log storage methods break variables into fragments and group fragments into units. This design choice leads to a significant number of numerical units containing only numerical characters. Based on our statistics on 15,402 units from 7 log types [18], approximately 53% of units are numerical on average, accounting for about 60% of compressed space (exceeding 80% on some logs).

This observation motivates previous works to encode these units as integer vectors, aiming to improve compression ratios and enable efficient arithmetic aggregation. However, since logs are inherently in text format, performing pre/suffix queries on them is common practice—and such queries are naturally incompatible with integer-encoded units. Existing approaches either encode numerical units as plain strings [33] or convert integers back to numerical characters before pre/suffix queries [35].

We propose an efficient method to perform pre/suffix queries on integer-encoded units, fully leveraging the vectorization potential offered by numerical units. As evaluated in Section 6.2, this method is even more efficient than directly executing pre/suffix queries on plain strings. Additionally, we introduce a vectorized generation method for intermediate query results to further accelerate analysis.

4.1 Vectorized Processing of Pre/suffix Queries

Prefix queries. Our key idea is to transform prefix queries into multiple range queries. For example, checking if "502769" has a prefix of "502*" is equivalent to verifying whether it belongs to the range "[502000, 503000)". We term this range as the "*compared range*" of the encoded number. However, the key challenge is how to efficiently determine the correct compared range for each encoded number using vectorized operations.

We propose to calculate a "*squeezing range*" for each encoded number to derive its compared range. Given a parameter b (a power of 2), for any encoded number, there exists a unique integer m such that the range $[b^{m-1}, b^m)$ contains the number. We term this range as the "*squeezing range*" of the encoded number.

It can be proved as follow that when b is a power of two smaller than 10, each squeezing range $[b^{m-1}, b^m)$ contains at most one compared range:

Proof. Consider a positive integer a within a compared range. If its compared range is contained in a squeezing range $[b^{m-1}, b^m)$, then

$$a \in [b^{m-1}, b^m) \tag{1}$$

Multiplying a by 10 yields another compared range. If this new range were also contained in the same squeezing range, we would have.

$$10 * a \in [b^{m-1}, b^m) \tag{2}$$

For $b < 10$, Equations 1 and 2 cannot hold simultaneously. Thus, each squeezing range contains at most one compared range. □

Based on this theorem, we design our vectorized prefix query algorithm. The algorithm first performs SIMD right-shifting operations (*_mm256_srli_epi32*) on all encoded numbers, shifting them by log_2^b bits per step. A number is within the squeezing range $[b^{m-1}, b^m)$ if it becomes zero after m shifts. Using a conditional blending instruction (*_mm256_blendv_epi8*), we load the identified compared range (or [0, 0) if none is found) into corresponding slots of the SIMD register. After loading ranges for all numbers, SIMD comparison instructions check whether each encoded number falls within the queried range, determining if the prefix matches.

In the implementation, to avoid frequent loading of similar compared ranges into the registers, we use the 16 YMM registers [9] to keep all alternative compared ranges stored in the registers. We also maintain a map between squeezing ranges and compared ranges in a table. Additionally, the value of parameter b determines the total execution count of SIMD right-shifting operations. Since a larger b reduces the number of shifts, we choose $b = 8$, the largest power of 2 smaller than 10.

A concrete example is shown in Table 3. Here, we have four encoded numbers. By shifting each number until it becomes zero (after m shifts), we determine its squeezing range. Each squeezing range either contains no compared range (e.g.,"3307") or exactly one compared range. We load the corresponding range as the "*queried range*" and perform vectorized queries within it. Note that the queried range may differ

Encoded Number	Compared Range	Moves to Zero	Squeezing Range	Queried Range	Query Result
502769	[502000, 503000)	7 (>>21)	[262144, 2097152)	[502000, 503000)	Hit
246532	[502000, 503000)	6 (>>18)	[32768, 262144)	[50200, 50300)	Miss
3307	[5020, 5030)	4 (>>12)	[512, 4096)	[0,0)	Miss
45237	[50200, 50300)	6 (>>18)	[32768, 262144)	[50200, 50300)	Miss

Table 3: An example for vectorized prefix queries (query "502*" on integer-encoded units)

from the compared range if the number lacks the queried prefix (e.g., "246532"), but this does not impact the final result.

Suffix queries. Suffix queries are much simpler than prefix queries and only require determining whether the modulo result of an encoded number matches the queried suffix. For example, checking if "6927502" has a suffix of "*502" is equivalent to determining whether "6927502" modulo 1000 equals 502. Since the modulo number (e.g., 1000 in this example) remains constant for all encoded numbers, it is retained as an immediate value to accelerate calculations.

Our algorithm computes modulo results for all encoded numbers and performs vectorized point queries on these results. While the modulo operation itself is not vectorized, the algorithm matches the performance of executing suffix queries directly on plain strings. Vectorized optimization of suffix queries is reserved for future work.

Mask Code	Array Order	Description
00000000	0 1 2 3 4 5 6 7	Original order
00000001	7 0 1 2 3 4 5 6	Move 7 to the head
....	....	
01001101	1 4 5 7 0 2 3 6	Move 1, 4, 5 and 7 to the head in turn
....	....	
11111111	0 1 2 3 4 5 6 7	Original order

Table 4: The shuffle rule to construct index

4.2 Vectorized Construction of Intermediate Results

Since an aggregation can perform queries on multiple variables simultaneously, it needs to store intermediate query results so that subsequent queries can leverage previous results. We adopt the *Indexed Bitmap* proposed in prior work [34] to record intermediate results, consisting of a bitmap and an index array tracking all positions marked "1" in the bitmap. To further accelerate analysis, we propose a vectorized construction strategy for this structure using SIMD shuffle instructions.

A SIMD shuffle instruction takes a "Mask Code" and an original array as inputs; it then reorders elements in the array according to the mapping defined by the Mask Code in a shuffle matrix. In our case, the result bitmap serves as the Mask Code, and the original array contains 32-bit natural number sequences. A customized shuffle matrix is shown in Table 4, designed to move positions corresponding to "1" in the bitmap to the front of the array. For example, if the bitmap (Mask Code) is "01001101", indicating hits at the second, fifth, sixth, and eighth positions, the shuffle instruction moves indices "1", "4", "5", "7" to the front. The array order becomes "1 4 5 7 0 2 3 6", with the first four numbers forming a segment of the index array.

The vectorized construction process involves three steps: 1) Perform vectorized comparison operations on encoded data to generate a bitmap of search results using the scatter instruction (*_mm256_movemask_epi8*). 2) Using the bitmap and shuffle matrix, construct an index array segment via the shuffle instruction (*_mm256_permutevar8x32_epi32*). 3) Gather all segments to build the Indexed Bitmap using the gather instruction (*_mm256_storeu_si256*).

An example of this process is illustrated in Figure 4. The algorithm first performs a vectorized point check "(==2)" on 16-bit encoded data to obtain the corresponding bitmap. It then loads the natural number sequence and bitmap into the register, using the customized shuffle matrix to reorder elements. This process constructs an index fragment for every 8 numbers, with all indices corresponding to "1" in the bitmap gathered at the front. These indices are written back to form the Indexed Bitmap.

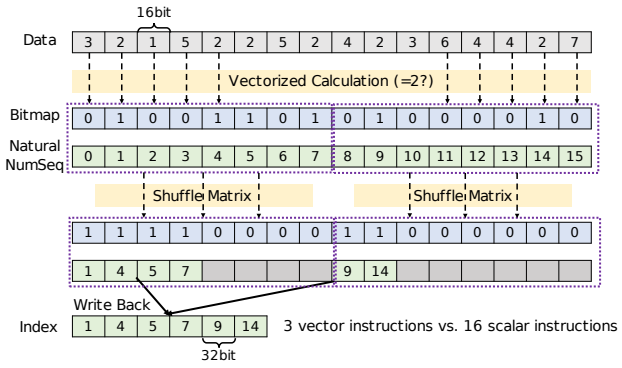


Figure 4: Vectorized construction of Indexed Bitmap

5 Implementation

By integrating all discussed techniques, we implement a log storage system named LogCrisp using approximately 15,000 lines of C++ code. LogCrisp simultaneously achieves high

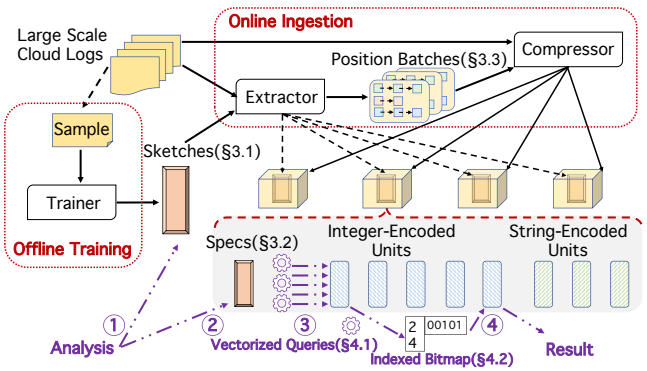


Figure 5: Overview of the LogCrisp system

compression ratios, fast ingestion speeds, and low analysis latency. Figure 5 presents an overview of LogCrisp. The workflow of LogCrisp can be summarized as follows:

Training. LogCrisp employs a *Trainer* to extract global information, including output statements and Sketches, from a log sample. It uses LogGrep's log parser to identify output statements. The specifics of Sketches are discussed in Section 3.1, and the extraction process is detailed in Section 3.2.

Compression. For each log block, LogCrisp uses an *Extractor* to: 1) Parse log variables based on output statements. 2) Locate fragments for encoding using pre-extracted Sketches. 3) Organize fragment positions into batches optimized for CPU cache lines (§3.3). Each compression stream corresponds to a batch group. The Extractor also extracts Specs for each variable type during ingestion (§3.2). A *Packer* then compresses fragments into units using zstd [13]: numerical units are encoded as integer vectors, while others use string vectors. Specs and compressed units are packaged into a zipped file.

Analysis. LogCrisp supports aggregated analysis on designated numerical units using query results from multiple units (e.g., `Count * where IP=xxx" and ERROR"`). Aggregation types include counting, summing, and min/max operations; query types include point, prefix/suffix, and range queries. LogCrisp supports: 1) Grep-like full-text queries, similar to prior works [29, 33, 34]; 2) Queries on specific units. Counting, point, and prefix/suffix queries work in both modes, while other aggregations and range queries require designated units.

The query process involves four steps: (1) Locate Variables: Match output statements to identify queried variables (like prior works [33]). (2) Filter Units (Steps ① and ②): Load zipped files and combine Sketches/Specs to filter relevant units (§3.2). (3) Execute Queries (Step ③): Perform vectorized queries on integer-encoded units and fixed-length queries on string-encoded units (§4.1). Store intermediate results in an Indexed Bitmap (§4.2). (4) Finalize Results(Step ④): Use the Indexed Bitmap to execute subsequent queries and generate final results.

6 Evaluation

We evaluate LogCrisp on a vast volume of logs and compare it with state-of-the-art works. Our evaluation answers the following questions:

- How does LogCrisp 's overall performance compare to state-of-the-art works? (§6.1)
- In LogCrisp, how much benefit does each key technology bring? (§6.2)
- How long is the training time of LogCrisp? How frequently must LogCrisp execute the training process? (§6.3)
- How does LogCrisp perform on different types of query strings? (§6.4)
- What percentage of query latency does each sub-operation in a single query account for? (§6.5)

The logs tested include three parts: 1) The four largest log types from the LogHub Dataset [18]; 2) All log types opened by the authors of CLP [37]; 3) Six types of real-world production logs collected from Alibaba Cloud. The total scale of these logs is nearly 7TB, with their names and sizes listed in Table 5.

We compare LogCrisp with two state-of-the-art methods, LogGrep and CLP:

- CLP [29]: the state-of-the-art global-pattern-based method, CLP is tested by running its open-source code [37] on all open logs. Due to our production environment using a self-developed OS incompatible with CLP, we exclude production log comparisons for CLP. Like LogCrisp, CLP uses zstd for packing.
- LogGrep [33]: the state-of-the-art local-pattern-based method, LogGrep is tested by running its open-source code [4] on both open and production logs. While LogGrep defaults to LZMA for packing, we changed it to zstd for fair comparison.

Since LogGrep does not support designating specific units, we standardize the comparison across all methods by testing counting analysis using grep-like full-text queries. We generated 22 query strings for production logs in collaboration with Alibaba Cloud engineers and 28 query strings for open logs to cover all execution paths of LogCrisp.

Before compression, a training process is required for each log type. For all open logs, sampling 1% of logs suffices to achieve over 99% Sketch matching rate. In practice, engineers can re-run the training when matching rates decline. Log ingestion uses a single thread for all logs except LogF (TB-scale), which employs 8-thread parallel ingestion with results normalized to a single thread.

Testbed. Experiments on open logs were conducted on a server with 2× Intel Xeon Silver 4210R (2.40GHz, 16 cores), 64GB RAM, and Ubuntu 11.3.0 kernel. Experiments on production logs used an Alibaba Cloud server with 2× Intel Xeon CPU E5-2682 v4 (2.50GHz, 32 cores), 188GB RAM, and a self-developed kernel based on Linux 3.10.0.

6.1 Overall Performance

We compare the overall analysis performance, ingestion speed, and compression ratio across all 13 log types. Detailed results are listed in Table 5. LogCrisp consistently optimizes analysis latency, compression ratio, and ingestion speed, outperforming CLP and LogGrep by 15.32× and 4.65× on average in analysis performance. It ingests logs 2.43× faster than LogGrep and achieves 95% of CLP's ingestion speed on average. Regarding compression ratio, LogCrisp matches 1.11× CLP and 96% of LogGrep's ratios on average.

Analysis latency. LogCrisp outperforms other systems in respect of aggregation analysis latency on all tested query strings on all datasets. We list the average analysis latency in the table.

To be concrete, on open datasets, the analysis latency of LogCrisp is 4.03× to 40.11× (average 15.32×) lower than that of CLP. We find LogCrisp outperforms CLP by 40.11× on average on Windows logs. This is because the inverted index on Windows is relatively simple since it does not have many variables. This is good news for compression ratio and ingestion speed but causes a high query latency by almost decompressing all compressed logs without any filtering. Compared with LogGrep, the analysis latency of LogCrisp is 4.30× to 10.90× (average 4.29×) lower. This is mainly because LogCrisp can achieve comparably high efficiency with the help of both Sketches and Specs and it can also achieve efficient vectorized queries on integer-encoded units. For example, LogCrisp can outperform LogGrep on Hadoop by 10.90×, since Hadoop includes many integer-encoded units, these units may cause a reduction on compression ratio, but can accelerate the analysis significantly by fully exploiting the vectorization technique.

As for the production logs, LogCrisp outperforms LogGrep by 2.50× to 7.27× (5.07× on average). Such result confirms the efficiency of LogCrisp to achieve fast aggregation analysis again.

To sum up, we find LogCrisp can outperform other compared systems and achieve significant performance gain in respect of analysis latency on both open and production logs.

Compression ratio. The compression ratio of LogCrisp is 89% and 95% of the highest compression ratio among three systems on average on LogHub dataset and CLP-opened dataset. LogGrep has the best compression ratio on these two datasets by 96% and 97% of the highest compression ratio on average.

We find LogCrisp achieves the highest compression ratio on HadoopL and Hive. Its compression ratio is comparable to LogGrep except Hadoop where LogGrep outperforms LogCrisp by 16%. The compression ratio of LogCrisp is higher than CLP on most logs except OpenStackC and Windows where CLP outperforms LogCrisp by 20% and 33% respectively. Our investigation reveals Hadoop includes many numerical values that are close to each other semantically, as a result, encoding them as strings has a lower storage overhead than encoding them as integers. OpenStackC includes many user-specific formats used by CLP. Windows includes few variables and many entries are in fact totally common, as a result, using global pattern is enough to achieve high compression ratio while more processing may incur extra overhead.

As for the production logs, the compression ratio of LogCrisp is comparable to LogGrep. The compression ratio of LogCrisp and LogGrep are 97% and 99% of the highest compression ratio on average respectively.

To sum up, introducing two-phase pattern extraction and encoding numerical units as integer vectors does not affect the compression ratio on most logs and the compression ratio of LogCrisp is comparable to other compatible systems on the tested logs.

Ingestion speed. The ingestion speed of LogCrisp is 91% and 95% of the highest ingestion speed on average on LogHub dataset and CLP-opened dataset respectively. CLP has the highest ingestion speed on these two datasets.

LogCrisp outperforms LogGrep on all logs significantly by up to 2.38×. LogCrisp outperforms CLP on 2 types of logs (Thunderbird, HadoopL) by up to 1.61×. These two logs has many output statements and variables, which incurs high overhead for CLP to build the inverted index. As for other logs, the ingestion speed of LogCrisp is within 99% of that of CLP on 2 types and lower than CLP on other 3 types by up to 79% (Windows). This is because many log entries in Windows are common, CLP does not need to build complex inverted index.

As for the production logs, the ingestion speed of LogCrisp is significantly higher than that of LogGrep by up to 3.72×.

To sum up, we find the ingestion speed of LogCrisp is comparable to or even outperforms global-pattern-based method, i.e. CLP, on most types of logs.

6.2 Individual Benefits of Proposed Techniques

We evaluate the impact of each proposed technique using all open logs in Table 5. The effectiveness of our vectorized algorithm correlates strongly with query type; we analyze its impact using representative Hadoop log queries: two point queries (Q1: "blk_1075557039_1816215", Q2: "134217728"), one prefix query (Q3: bytes: "1*"), one suffix query (Q4: "*28,"), and one range query (Q5: "bytes > 1342"). Similar trends are observed across other logs and queries.

	Source	Size (GB)	Compression Ratio			Ingestion Speed (MB/s)			Average Analysis Latency (s)		
			CLP	LogGrep	LogCrisp	CLP	LogGrep	LogCrisp	CLP	LogGrep	LogCrisp
Hadoop	LogHub	16.04	27.63	**33.48**	28.85	**97.16**	40.45	96.59	45.09	26.37	**2.42**
HadoopL	Opened by CLP	428.87	42.30	54.31	**54.76**	87.52	43.31	**89.33**	603.83	82.81	**56.70**
Hive	Opened by CLP	1.36	44.63	47.39	**47.76**	**116.19**	46.47	99.59	1.08	1.48	**0.25**
OpenStackC	Opened by CLP	32.93	**36.13**	33.93	30.71	**101.56**	52.11	100.35	60.86	15.90	**6.15**
Spark	LogHub	2.71	21.49	**29.10**	28.63	**77.04**	30.48	64.50	7.05	1.48	**0.36**
Thunderbird	LogHub	29.61	22.76	**34.41**	32.83	49.14	38.23	**58.87**	38.39	20.82	**9.53**
Windows	LogHub	26.09	**404.80**	339.74	304.50	**140.61**	67.63	111.31	62.98	4.58	**1.57**
LogA	Production Log	18.67	/	**23.17**	21.79	/	23.93	**69.79**	/	8.05	**3.23**
LogB	Production Log	45.82	/	17.72	**18.24**	/	17.85	**54.75**	/	20.15	**4.25**
LogC	Production Log	65.74	/	**14.17**	13.79	/	14.72	**41.25**	/	50.04	**10.21**
LogD	Production Log	2.12	/	**10.84**	10.14	/	17.61	**65.65**	/	4.02	**0.66**
LogE	Production Log	7.82	/	16.18	**16.82**	/	19.82	**69.63**	/	1.29	**0.18**
LogF	Production Log	6149.34	/	**14.46**	13.93	/	6.20	**10.73**	/	6199.90	**1268.54**

Table 5: Overall performance comparison. The best result across the three systems for each log type and metric is highlighted in bold.

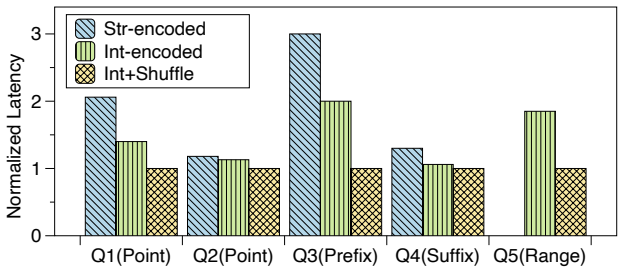


Figure 6: Impact of vectorized per/suffix queries and vectorized construction of indexed bitmap

Two-phase pattern extraction (§3.2). We implement a version called "LC-global" by only extracting a global pattern on our log sample. We use such pattern to ingest logs as well as filter units during the query. We test "LC-global" on all 7 open logs and plot the results of 6 types of logs in Figure 7. We do not put the result on HadoopL here since we do not observe significant impact of such technique, which is mainly because its patterns are relatively static on all log blocks and a single global pattern is enough to filter most units. However, on logs such as OpenStackC and Spark, global pattern will be too general to serve for filtering. After we decouple the Sketches and Specs within the pattern, we can eliminate the average analysis latency by 2.59× on average and up to 5.55× (Spark). Besides, two-phase pattern extraction can maintain or even improve (by up to 7%) the ingestion speed, since it can directly break the variables into fragments during the ingestion without complex pattern matching. It also has less effect on the compression, which only causes the compression ratio to drop by 9% on windows since it has a relatively large Sketch Warehouse. As a result, we need to record multiple assigned numbers for backup Sketches in the units of main Sketch.

Cache-friendly ingestion (§3.3). We implement a version called "LC-w/o-cache" by recording positions in a list and maintain metadata using a tree-based index like LogGrep. According to Figure 7, we find by introducing the cache-friendly design, we can improve the ingestion speed by 6% to 43% (19% on average). Such method will have a more significant improvement when each unit has many fragments such as on Spark and Thunderbird.

Vectorized pre/suffix queries (§4.1). We first encode all NUnits as plain strings and execute queries on them and mark this version as "Str-encoded". Then we change the encoding format of NUnit into integer vector, execute our proposed vectorized queries on them and mark this version as "Int-encoded". According to Figure 6, we find by encoding NUnits as integer vectors and exploiting vectorization on such format, we can reduce the latency of point and pre/suffix queries and support range queries additionally. Specifically, we can eliminate the latency of point queries, prefix queries, and suffix queries by 1.46×, 1.50×, and 1.22× respectively. The results on prefix and suffix queries demonstrate by exploiting vectorization properly, we can overcome the incompatibility between numerical encoding format and full-text queries semantic.

Vectorized construction of indexed bitmap (§4.2). We change the construction process of Indexed Bitmap in "Int-encoded" version to our proposed vectorized manner, and mark such version as "Int+Shuffle". According to Figure 6, we find the vectorized construction process of Indexed Bitmap can eliminate the latency by 1.4× for Q1, 2× for Q3 and 1.85× for Q5. We find these three queries have dense results, as a result, their construction processes of the Indexed Bitmap take up a significant part.

6.3 Analysis of Training Overhead

Since the training process to extract Sketches is crucial for delivering a global description in our design, we measure the training overhead in terms of the training time, miss rate as well as the training frequency.

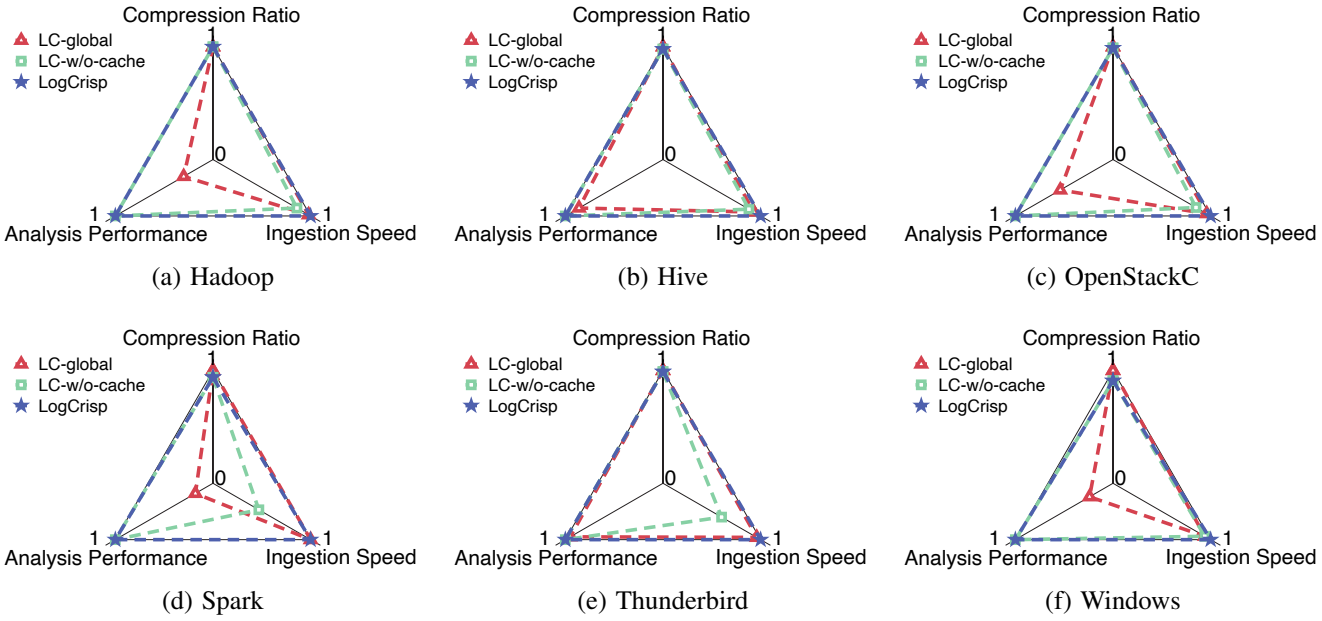


Figure 7: Impact of two-phase pattern extraction and cache-friendly ingestion (normalized to the best result in each dimension)

Training time. We evaluate the training time of different sample rates on an open log (Hadoop) as well as a production log (LogA). The results are shown in Figure 8. We find the training time of production log is relatively larger than that of open log due to the higher number of Sketches in production logs. Generally speaking, in our experiment (sampling 1% log), the relative training overhead, namely the training time divided by the compression time, is 10% (Hadoop) and 5.9% (LogA) respectively. The training overhead is higher on Hadoop since the compression time on Hadoop is shorter than that on LogA.

Miss rate. Miss rates for different sample rates on sampled logs are shown in Figure 8. Open logs exhibited higher sensitivity to sample rate than production logs. Given the sharp increase at 10% sampling, we chose 1% sample rate for experiments to balance training time and miss rate.

Training frequency. To evaluate training frequency, we sorted Hadoop and LogA logs by timestamp and split them into 64MB blocks. We trained on 1% of the first block and applied results to subsequent blocks, triggering retraining (with 1% sampling of all processed blocks) when miss rate exceeded 5%.

Figure 9 shows the miss rate per block and total training counts. For Hadoop (16GB total), we train for 3 times (5.3GB per training on average). For LogA (18GB total), we train for 6 times (3GB per training on average). We find the production logs change more frequently and thus the training process will be executed for more times.

6.4 Performance of Different Query Interfaces

We show the comparison between LogCrisp and other systems across different query interfaces. We evaluate 13 point queries, 7 prefix queries, 5 suffix queries and 3 range queries in total and plot their average analysis speed with maximum and minimum values as error bars in Figure 11.

For CLP, we find it performs better on point queries than prefix and suffix queries, which is due to the complex wildcard processing in CLP's algorithm. LogGrep exhibits nearly identical average performance across query types. As for LogCrisp, it performs better on point and range queries than those on prefix and suffix queries on average, since the integer-encoded units are more compatible with the former two types. But we find LogCrisp still outperforms LogGrep in prefix and suffix queries even though the string-encoded units used by LogGrep are compatible with such queries.

Besides, compared to other systems, LogCrisp exhibits a wider speed range. This is because compared to filtering using only global or local patterns, filtering with decoupled patterns varies more significantly for each queried string. Queried strings containing NAU characters can use the locating messages within Sketches to filter units while other strings can only rely on the Specs.

6.5 Breakdown of Query Latency

We plot a breakdown analysis of query latency for Q1-Q5 in Figure 10. We find for different queries, the Sketch loading and Spec loading latency remain the same. In practice, users

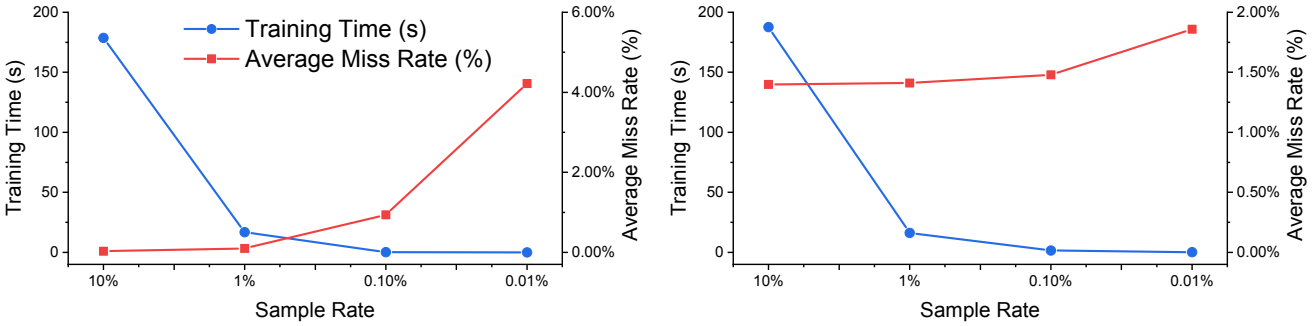


Figure 8: The training time and miss rate for different sample rate

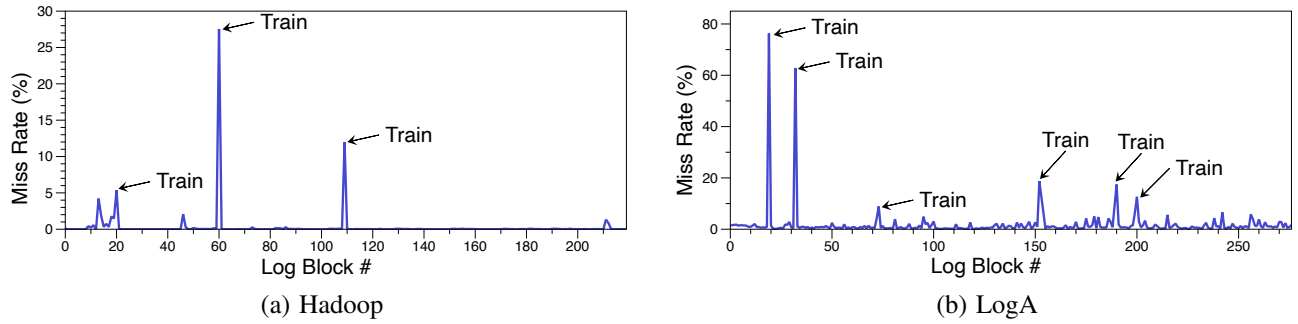


Figure 9: The miss rate on different blocks in the time series (showing the training frequency)

can load them once and execute multiple query strings based on them. We find the point queries and the suffix queries spend much time (0.84 - 1.12 seconds) to match on the patterns, as these queries need to match patterns across multiple log variables. The prefix queries and the range queries can directly find the related log variable based on the output statement and their matching latency is thus as low as only 0.03 seconds. We find the latency of the prefix queries and the range queries is much lower than that of the point queries and the suffix queries. This is because the point queries need to search within more units and the suffix queries need to calculate modulus result for each encoded number.

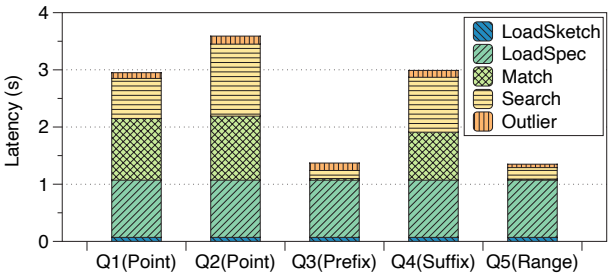


Figure 10: Query latency breakdown

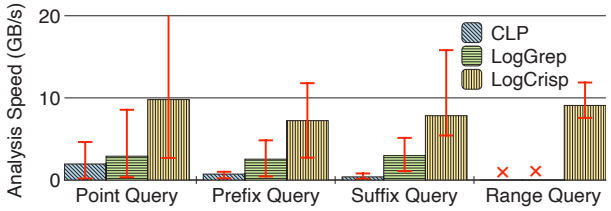


Figure 11: Query performance for different interfaces. CLP and LogGrep cannot support range query.

7 Conclusion

This paper presents LogCrisp, a log storage system designed to enable fast aggregated analysis on large-scale compressed logs. Our evaluation shows that by decoupling messages within log patterns, LogCrisp efficiently generates global descriptions while maintaining filtering effectiveness. The proposed paradigm enhances ingestion speed through hardware-optimized techniques (e.g., cache-line alignment). By leveraging vectorization properly, LogCrisp overcomes the semantic mismatch between numerical encoding and full-text queries, enabling vectorized queries on log data for the first time.

Acknowledgment

We thank all reviewers for their insightful comments and helpful suggestions. We are especially grateful to our shepherd, Dan Feng, for her detailed and patient guidance during our camera-ready preparation. This work was supported by the National Natural Science Foundation of China under Grant 62025203.

References

[1] Daniel Abadi, Peter Boncz, Stavros Harizopoulos, Stratos Idreos, Samuel Madden, et al. The design and implementation of modern column-oriented database systems. *Foundations and Trends® in Databases*, 5(3):197–280, 2013.

[2] Daniel Abadi, Samuel Madden, and Miguel Ferreira. Integrating compression and execution in column-oriented database systems. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 671–682, 2006.

[3] Rachit Agarwal, Anurag Khandelwal, and Ion Stoica. Succinct: enabling queries on compressed data. In *NSDI'15 Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, pages 337–350, 2015.

[4] LogGrep authors. Source code of loggrep. https://github.com/THUBear-wjy/LogGrep, 2022.

[5] Elasticsearch B.V. Elasticsearch 7.8.0. https://www.elastic.co/downloads/past-releases/elasticsearch-7-8-0, 2020.

[6] Wei Cao, Xiaojie Feng, Boyuan Liang, Tianyu Zhang, Yusong Gao, Yunyang Zhang, and Feifei Li. Logstore: A cloud-native and multi-tenant log database. In *Proceedings of the 2021 International Conference on Management of Data*, pages 2464–2476, 2021.

[7] Michael Chow, David Meisner, Jason Flinn, Daniel Peek, and Thomas F Wenisch. The mystery machine: End-to-end performance analysis of large-scale Internet services. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation*, pages 217–231. USENIX Association, 2014.

[8] Mihai Christodorescu, Nicholas Kidd, and Wen-Han Goh. String analysis for x86 binaries. In *Proceedings of the 6th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 88–95, 2005.

[9] Intel Corporation. Intel® avx-512 instructions. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-avx-512-instructions.html, 2022.

[10] Intel Corporation. Intel® instruction set extensions technology. https://www.intel.com/content/www/us/en/support/articles/000005779/processors.html, 2022.

[11] Min Du and Feifei Li. Spell: Streaming parsing of system event logs. In *Proceedings of the 16th International Conference on Data Mining*, pages 859–864. IEEE, 2016.

[12] Min Du, Feifei Li, Guineng Zheng, and Vivek Srikumar. DeepLog: Anomaly detection and diagnosis from system logs through deep learning. In *Proceedings of 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1285–1298. ACM, 2017.

[13] Facebook. z-standard compression tool. https://github.com/facebook/zstd, 2021.

[14] Bettina Fazzinga, Sergio Flesca, Filippo Furfaro, and Luigi Pontieri. Online and offline classification of traces of event logs on the basis of security risks. *Journal of Intelligent Information Systems*, 50(1):195–230, 2018.

[15] Paolo Ferragina and Giovanni Manzini. An experimental study of a compressed index. *Information Sciences*, 135(1):13–28, 2001.

[16] LogArchive group. Open source code of LogArchive. https://github.com/robertchristensen/log_archive_v0, 2019.

[17] Pinjia He, Jieming Zhu, Zibin Zheng, and Michael R Lyu. Drain: An online log parsing approach with fixed depth tree. In *Proceedings of 2017 IEEE International Conference on Web Services*, pages 33–40. IEEE, 2017.

[18] Shilin He, Jieming Zhu, Pinjia He, and Michael R. Lyu. Loghub: A large collection of system log datasets towards automated log analytics. *arXiv preprint arXiv:2008.06448*, 2020.

[19] Scalyr Inc. Scalyr home page. https://www.scalyr.com/, 2021.

[20] Splunk Inc. Spunk enterprise 8.0.3. https://www.splunk.com/en_us/download/previous-releases.html, 2020.

[21] Zhen Ming Jiang, Ahmed E Hassan, Parminder Flora, and Gilbert Hamann. Abstracting execution logs to execution events for enterprise applications (short paper). In *Proceedings of the 8th International Conference on Quality Software*, pages 181–186. IEEE, 2008.

[22] Grafana Labs. Loki documentation. https:grafana.com/docs/loki/latest/, 2021.

[23] Harald Lang, Tobias Mühlbauer, Florian Funke, Peter A Boncz, Thomas Neumann, and Alfons Kemper. Data blocks: Hybrid oltp and olap on compressed storage using both vectorization and compilation. In *Proceedings of the 2016 International Conference on Management of Data*, pages 311–326, 2016.

[24] Charles Levert. Zgrep linux manuel. https://linux.die.net/man/1/zgrep, 2020.

[25] Hao Lin, Jingyu Zhou, Bin Yao, Minyi Guo, and Jie Li. Cowic: A column-wise independent compression for log stream analysis. In *Proceedings of the 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 21–30. IEEE, 2015.

[26] Jinyang Liu, Jieming Zhu, Shilin He, Pinjia He, Zibin Zheng, and Michael R Lyu. Logzip: extracting hidden structures via iterative clustering for log compression. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering*, pages 863–873. IEEE, 2019.

[27] Giulio Ermanno Pibiri, Matthias Petri, and Alistair Moffat. Fast dictionary-based compression for inverted indexes. In *Proceedings of the Twelfth ACM International Conference on Web Search and Data Mining*, pages 6–14, 2019.

[28] Giulio Ermanno Pibiri and Rossano Venturini. Techniques for inverted index compression. *ACM Computing Surveys*, 53(6):1–36, 2020.

[29] Kirk Rodrigues, Yu Luo, and Ding Yuan. Clp: Efficient and scalable search on compressed text logs. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI'21)*, pages 183–198, 2021.

[30] Kunihiko Sadakane. Succinct representations of lcp information and improvements in the compressed suffix arrays. In *Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 225–232, 2002.

[31] Khalid Sayood. *Introduction to data compression*. Morgan Kaufmann, 2017.

[32] Stackoverflow. How to calculate fingerprint. https://stackoverflow.com/questions/51059782, 2018.

[33] Junyu Wei, Guangyan Zhang, Junchao Chen, Yang Wang, Weimin Zheng, Tingtao Sun, Jiesheng Wu, and Jiangwei Jiang. Loggrep: Fast and cheap cloud log storage by exploiting both static and runtime patterns. In *18th European Conference on Computer Systems (Eurosys'23)*, 2023.

[34] Junyu Wei, Guangyan Zhang, Junchao Chen, Yang Wang, Weimin Zheng, Tingtao Sun, Jiesheng Wu, and Jiangwei Jiang. Exploiting data-pattern-aware vertical partitioning to achieve fast and low-cost cloud log storage. *ACM Trans. Storage*, 20(2), February 2024.

[35] Junyu Wei, Guangyan Zhang, Yang Wang, Zhiwei Liu, Zhanyang Zhu, Junchao Chen, Tingtao Sun, and Qi Zhou. On the feasibility of parser-based log compression in large-scale cloud systems. In *19th USENIX Conference on File and Storage Technologies (FAST'21)*, pages 249–262, 2021.

[36] Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael I Jordan. Detecting large-scale system problems by mining console logs. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 117–132, 2009.

[37] YScope. clp-core. https://github.com/y-scope/clp-core, 2021.

[38] Ding Yuan, Soyeon Park, Peng Huang, Yang Liu, Michael M Lee, Xiaoming Tang, Yuanyuan Zhou, and Stefan Savage. Be conservative: Enhancing failure diagnosis with proactive logging. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI'12)*, pages 293–306, 2012.

[39] Ding Yuan, Jing Zheng, Soyeon Park, Yuanyuan Zhou, and Stefan Savage. Improving software diagnosability via log enhancement. *ACM Transactions on Computer Systems (TOCS)*, 30(1):1–28, 2012.

[40] Feng Zhang, Weitao Wan, Chenyang Zhang, Jidong Zhai, Yunpeng Chai, Haixiang Li, and Xiaoyong Du. Compressdb: Enabling efficient compressed data direct processing for various databases. In *Proceedings of the 2022 International Conference on Management of Data*, pages 1655–1669, 2022.

[41] Feng Zhang, Jidong Zhai, Xipeng Shen, Dalin Wang, Zheng Chen, Onur Mutlu, Wenguang Chen, and Xiaoyong Du. Tadoc: Text analytics directly on compression. *The VLDB Journal*, 30(2):163–188, 2021.

HotRAP: Hot Record Retention and Promotion for LSM-trees with Tiered Storage

Jiansheng Qiu [†]
qjc21@mails.tsinghua.edu.cn

Fangzhou Yuan [†]
yfz23@mails.tsinghua.edu.cn

Mingyu Gao [† ‡]
gaomy@tsinghua.edu.cn

Huanchen Zhang [† ‡ *]
huanchen@tsinghua.edu.cn

† Institute for Interdisciplinary Information Sciences, Tsinghua University ‡ Shanghai Qi Zhi Institute

Abstract

Tiered storage architectures are promising to improve cost efficiency by combining small and fast storage with slower but cheaper mediums. However, existing designs of Log-Structured Merge-trees (LSM-trees) on tiered storage cannot simultaneously support efficient read and write accesses. Keeping the upper and lower LSM-tree levels in the fast and slow storage respectively (i.e., tiering) allows efficient writes to the fast disks, but read-hot data may be stuck in the slow disks. Putting all the levels in the slow storage and using the fast disks as a cache (i.e., caching) can handle frequently read data efficiently, but LSM-tree compactions now need to happen in the slow disks.

We present HotRAP, a key-value store based on RocksDB that follows the tiering approach above, but enhances it to timely promote hot records individually from slow to fast storage and keep them in fast storage while they are hot. HotRAP uses an on-disk data structure (a specially-made LSM-tree) to track the hotness of keys in a fine-grained manner, and leverages two pathways to ensure that hot records reach fast storage with short delays. Our experiments show that HotRAP outperforms state-of-the-art LSM-trees on tiered storage by up to 1.6× compared to the second best under read-write-balanced YCSB workloads with common access skew patterns, and up to 1.5× under Twitter production workloads.

1 Introduction

Log-Structured Merge-trees (LSM-trees) [20, 30] are widely adopted to build key-value stores [6, 12, 16, 19, 27, 36] and database storage engines [1, 2, 18, 22, 24] because of their superior write performance on traditional storage mediums. When the underlying storage architecture evolves, the design of LSM-trees also needs to adapt to new hardware characteristics. Specifically, with the recent trend of *tiered storage*, a small fast tier, e.g., local solid-state drives (SSDs), is combined with a slower but cheaper tier, e.g., hard disk drives (HDDs) or cloud storage, to achieve better cost efficiency. Optimizing LSM-trees for tiered storage thus becomes a practical and important topic to explore.

Table 1: Trade-offs on read and write performance among different LSM-tree designs on tiered storage.

	Tiering	Caching	HotRAP
Write-heavy	✓	✘	✓
Read-heavy	✘	✓	✓

Intuitively, there are two basic approaches to porting LSM-trees to tiered storage. Since LSM-trees naturally organize into multiple levels, the *tiering* design[1] locates their upper levels in the fast tier while storing the lower levels with the majority of the records in the slow tier [17, 26, 31, 33, 39, 43]. Alternatively, the *caching* design directly uses the fast tier as a *cache* of the slow tier, with the entire LSM-tree located in the latter [8, 9, 21, 28, 35]. However, neither approach is ideal, as summarized in Table 1. The tiering design is inherently efficient for write operations (i.e., inserts, updates) because the append-only nature of the LSM-tree keeps the most recent writes automatically in the fast tier. However, read-hot records have few ways to be promoted to the upper levels, and are left in the slow tier with high access latency, thus compromising the read performance. The caching design, in contrast, can effectively migrate frequently read records to the fast tier cache. However, compactions in the caching design all happen in the slow tier, which constrains system performance under write-heavy workloads. Moreover, writes to these records would need to be performed twice in both tiers to keep data consistency, leading to extra time and energy overheads.

While slow tier compactions and duplicated writes in the caching design seem fundamental and difficult to avoid, we believe it is possible to introduce additional promotion mechanisms in the tiering design to improve its read performance. Several prior solutions such as LogStore [26] and Mir-

*Corresponding author.

[1] Note that this is different from the tiering compaction policy.

rorKV [33] adjust the placement of blocks/SSTables across storage tiers periodically based on access frequencies. Nevertheless, these approaches still exhibit several key limitations. First, they cannot efficiently utilize the precious fast tier capacity because they move data at a coarse granularity, where many cold records in the identified hot blocks/SSTables are altogether piggybacked to the fast tier (**limitation 1**). Second, even if we directly reduce the granularity to record-level, the huge number of individual records would require excessive metadata information for hotness tracking, easily exceeding the limited memory capacity [40,41] and requiring efficient *in-storage* structures (**limitation 2**). Third, the above solutions can only promote hot data to the fast tier through LSM-tree compactions. To deal with read-heavy workloads where compactions happen infrequently, several systems [26, 31] allow triggering compactions proactively, but they must wait for the hot records to accumulate in the slow tier before promoting them. Such a delay harms read performance because it could overstep the time window when a record is hot (**limitation 3**).

In this paper, we present HotRAP (Hot record Retention And Promotion), an LSM-tree design specifically optimized for tiered storage. HotRAP can accurately identify hot records in the LSM-tree, promote them timely from the slow disk (abbr. as *SD* hereafter) to the fast disk (abbr. as *FD* hereafter) at a fine granularity, and retain them in FD as long as they stay hot. We design HotRAP based on RocksDB [14]. Figure 1 shows its big picture.

HotRAP addresses the aforementioned three limitations in the following ways. First, HotRAP supports hotness tracking and data promotion/retention at the *record level* rather than at the block/SSTable level, thus preventing cold records from being piggybacked to FD (addressing **limitation 1**). It logs each record access in a specifically designed structure, called *RALT* (Recent Access Lookup Table). RALT is essentially a small LSM-tree, located in FD instead of the memory (addressing **limitation 2**). We choose LSM-tree to implement RALT in order to benefit from its low write latency on disks because inserting access records into it is on the critical path. RALT tracks the access history for each logged key and maintains a hotness score for the key. RALT then evicts low-score keys periodically from itself to stay under a size limit that can be automatically tuned according to the workload.

Also, besides waiting for LSM-tree compactions to retain/promote hot records, HotRAP introduces a small in-memory *promotion buffer* to buffer records read from SD and timely promotes the hot ones (via checking RALT) by flushing them to the top of the data LSM-tree (addressing **limitation 3**). More specifically, HotRAP provides the following two pathways for hot records to reside in FD: hotness-aware compaction, and promotion by flush. First, during compactions from FD to SD, HotRAP extracts records within the compaction range from the mutable promotion buffer and adds them to the input of the compaction. Then HotRAP checks the hotness of each record in the compaction output by sequentially scanning the corresponding key range in RALT. Hot records are written to FD instead of down to SD. Consequently, hot records in FD are retained in FD, and hot records in SD are promoted to FD. We call this path *hotness-aware compaction*. Second, when the mutable promotion buffer grows too big due to insufficient compactions, we turn it into an immutable promotion buffer and trigger *promotion by flush* to bulk-insert the hot records in the immutable promotion buffer (also identified via consulting RALT) to L_0 of the LSM-tree. Thus we effectively restrict the size of the promotion buffer. To prevent promotion by flush from overwriting records with a newer version (i.e., promoting a stale record to L_0 that has a higher level than the newly updated version), we perform extra checks and carry out a concurrency control mechanism to guarantee correctness.

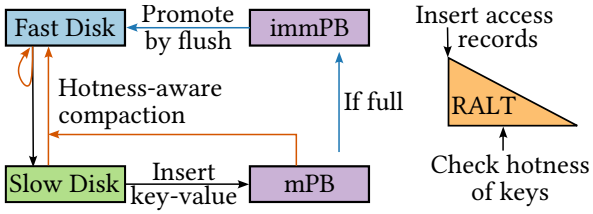


Figure 1: The high-level picture of HotRAP. RALT is a small LSM-tree in FD that tracks the hotness of keys. *mPB* and *immPB* stand for the mutable and immutable promotion buffers. Hot records are retained in FD during compactions. Records accessed in SD are inserted into the promotion buffer and then promoted by compaction or flush if they are hot.

We evaluate HotRAP using YCSB workloads and Twitter production traces [37] on AWS instances with fast local NVMe SSDs and slower cloud storage. We compare against the state-of-the-art LSM-trees on tiered storage, e.g., RocksDB using tiering/caching design, SAS-Cache [9] (caching design, based on RocksDB with secondary block cache [28]), and PrismDB [31] (tiering design, proactively triggering compactions). HotRAP achieves 5.2× speedup over the second best baseline with the tiering design for read-only YCSB workloads, 2.1× speedup with the caching design for write-heavy YCSB workloads, and 1.6× speedup over both designs for read-write-balanced YCSB workloads. HotRAP also achieves 1.5× speedup for Twitter production workloads. These results match the insights in Table 1. Our experiments also show that HotRAP adds $< 4\%$ overhead to the plain RocksDB under uniform workloads and is robust against hotspot shifts, where RALT quickly evicts stale access records and adds keys in the new hotspot into the hot set.

We make three primary contributions in this paper. First, we propose an on-disk data structure (i.e., RALT, a specially-made LSM-tree) for tracking the hotness of key-value records. It operates at the record level so that the system can efficiently utilize the limited space of FD. Second, we design two pathways in a tiered LSM-tree for timely promoting/retaining hot

records to/in FD. Finally, we implement HotRAP, a key-value store based on RocksDB that outperforms state-of-the-art LSM-trees on tiered storage because of its efficient hot record tracking and movement features. The code is available at https://github.com/hotrap/HotRAP.

2 Background & Related Work

2.1 Log-Structured Merge-tree (LSM-tree)

A Log-Structured Merge-tree (LSM-tree) consists of in-memory buffers (i.e., MemTables) and multiple levels $L_0, \ldots, L_n$ on disk. The capacity of level L_i is made T times larger than L_{i-1}, where T is the size ratio of the LSM-tree. Records are first inserted into the mutable MemTable. When it is full, it becomes immutable and then flushed to L_0 as an SSTable (i.e., a file format called Sorted String Table) in the background. When level L_{i-1} reaches its capacity, it will trigger the compaction process to merge its content into the next level L_i. There are typically two kinds of compaction policies: leveling and tiering. The leveling policy only allows one sorted run per level while the tiering policy allows multiple. This paper focuses on the leveling policy because it is RocksDB's default choice [27]. RocksDB also adopts partial compaction: each compaction picks an SSTables from L_{i-1} whose key range overlaps with a minimal number of SSTables in L_i to merge to the next level. Such a compaction strategy leads to a write amplification of $\approx \frac{nT}{2}$ [11].

A lookup first checks the MemTable and then searches the levels from top to bottom until a matching key is found. A block index in memory is used to determine which SSTable data block to search for a particular key in a sorted run. Per-SSTable Bloom filters are used to reduce the number of candidate SSTables to further save I/Os. RocksDB provides snapshot isolation via multi-version concurrency control (MVCC) to prevent compactions from blocking normal read operations. A snapshot in RocksDB, called a superversion, is created after a flush or compaction completes. An old snapshot is garbage collected when no active queries refer to it.

Although many LSM-trees such as RocksDB are initially designed for local SSDs, the multi-level nature of LSM-trees is a good fit for the tiering design. The upper levels contain the recently inserted and updated records and are therefore kept in the fast storage such as local SSDs because they are more likely to be accessed in the near future. To improve the system's cost efficiency, the majority of records in the lower levels are placed in HDDs [13, 14] or low-tier cloud storage based on HDDs [15]. HDDs exhibit higher latency than SSDs, but they are much cheaper. For example, the unit price for a 20TB Seagate Enterprise HDD Exos X20 today is 6.75× cheaper than a 7.68TB SAMSUNG Enterprise SSD PM9A3 [4, 5]. That means a tiered storage with a size ratio of 1:10 based on these hardware can reduce the storage cost by 77% compared to pure SSDs of the same capacity.

2.2 LSM-trees with the tiering design

LogStore [26] maintains histograms in memory to track the hotness of SSTables and retains/promotes hot SSTables in/to the faster storage. However, the granularity of SSTables is too coarse because there can be considerable cold data in the same SSTable that is considered hot.

MirrorKV [33] splits the LSM-tree into the key and value LSM-tree and caches the hottest key SSTables in the faster storage. Additionally, MirrorKV retains the hottest blocks (e.g., 10%) during compactions from L1 to L2. However, the granularity of blocks is still too coarse because small objects are prevalent in large-scale systems [25], and there can be many cold tiny records in a hot data block.

SA-LSM [43] accurately predicts cold data with survival analysis and demotes cold records from the faster storage to the slower storage. However, SA-LSM does not support promoting hot records back to the faster storage, and the training cost of the survival model is heavy.

PrismDB [31] estimates key popularity with the clock algorithm, and the clock bits are indexed with a hash table. Hot records are retained/promoted in/to the fast disk during compactions. However, the hash table can consume considerable memory. Also, the promotion speed of PrismDB is slow because it only promotes during compactions.

Although not an LSM-tree, 2-Tree [44] also uses tiered storage by maintaining two B-trees: one in memory for hot records and one on disk for cold records. However, it does not support tiered disk storage and the in-memory B-tree consumes huge memory.

2.3 LSM-trees with the caching design

Numerous prior studies aim to improve the performance of LSM-trees with in-memory caches [32, 34, 38, 42, 45]. However, the size of hot records can be far larger than the memory capacity. RocksDB, therefore, introduces the secondary cache on fast SSDs for data blocks not hot enough for the in-memory block cache [28]. SAS-Cache [9] further proposes several optimizations on the block cache, such as actively removing the cached blocks invalidated by compactions. However, the granularity of blocks is too coarse.

It is also a common practice to employ a separate key-value cache such as CacheLib [8] on top of the LSM-tree. However, the cache incurs extra disk I/O when updating, because it needs to update the key in both the cache and the LSM-tree. It is also challenging to ensure consistency between the cache and the LSM-tree. As a common problem of the caching design, write performance suffers because all compactions happen in the slow storage.

2.4 Memory footprint of hotness tracking

As discussed above, the granularity of SSTable/block is too coarse for retention and promotion. However, tracking the

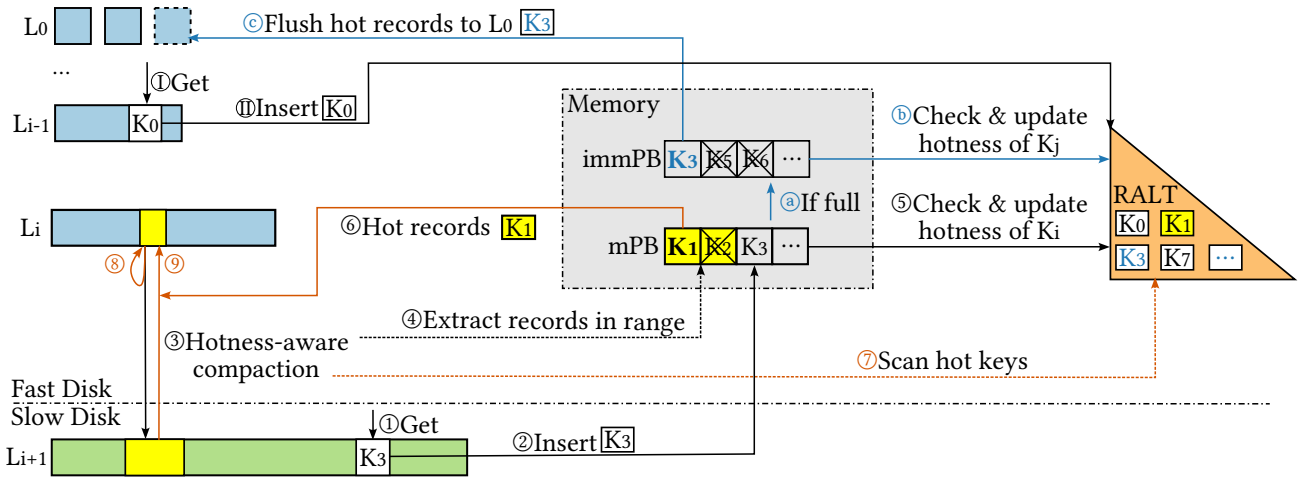


Figure 2: Overview of HotRAP. *mPB* and *immPB* stand for the mutable and immutable promotion buffers. Solid arrows are data flow. Dashed arrows are control flow. The accessed keys in SD are firstly inserted into the mutable promotion buffer (① to ②). A compaction can piggyback hot records in its range to FD (③ to ⑥). Hot records are retained in FD during compactions (⑦ to ⑧). If the mutable promotion buffer becomes full, hot records in it will be flushed to Level 0 (ⓐ to ⓒ).

access history of (potentially) hot records in memory can incur a large footprint. According to the Twitter trace [37], for example, 50% of the workloads have a value size smaller than 5× the key size. We take the AWS EC2 i4i.2xlarge instance for example, which is equipped with 64GB memory and an 1875GB local AWS Nitro SSD [3]. The configuration reflects the typical memory-disk ratio in the industry. If 1TB of its local SSD is used to store hot records, we need at least $1\text{TB}/(1+5) = 166.7\text{GB}$ memory to track those hot keys, which exceeds the physical memory of the instance.

3 Design & Implementation

3.1 Overview

The overview of HotRAP is shown in Figure 2. There are two components to facilitate retention and promotion: Recent Access Lookup Table (*RALT*) and the promotion buffer. RALT is responsible for tracking the hotness of records. It maintains a set of hot keys (without values) worth promoting and retaining. The promotion buffer consists of a mutable promotion buffer and a list of immutable promotion buffers. Records read from SD are inserted into the mutable promotion buffer. The mutable promotion buffer is turned into immutable when full. Immutable promotion buffers are flushed to disk in the background. The promotion buffers reside between the last level of FD and the first level of SD. To read a key, HotRAP first searches in MemTables and levels in FD, then in the mutable promotion buffer, and finally in levels in SD.

When a record in FD is accessed (①), its key will be inserted into RALT to record the access (Ⅱ). When a record in SD is accessed (①), HotRAP first inserts the key into the mutable promotion buffer (②), and the key's hotness information in RALT is updated later.

Hotness-aware compaction. During compactions from FD to SD (③), HotRAP extracts records within the compaction range from the mutable promotion buffer (④). For the example in Figure 2, K_1 and K_2 are extracted. HotRAP consults RALT about whether each key is hot, and updates the hotness score in RALT accordingly (⑤). Hot records (K_1) are added to the input of the compaction (⑥). Cold records (K_2) are dropped and future lookups to them would go to SD. Then HotRAP checks the hotness of each record in the compaction output, and writes hot ones to FD. It does so by constructing a RALT iterator in the compaction range. This iterator produces the hot records identified by RALT in the order of keys. Therefore, we can advance the compaction iterator and the RALT iterator in a sort-merge manner (⑦), to decide the records to be written into new SSTables in FD instead of SD. Consequently, hot records in FD are retained in FD (⑧), and hot records in SD are promoted to FD (⑨). The I/O incurred by the RALT iterator is small because RALT does not store values of HotRAP records. Compactions within SD are also hotness-aware: hot records are retained/promoted in/to the higher level in SD. Unlike the compactions from FD to SD, no records are extracted from the mutable promotion buffer.

Promotion by flush. For read-heavy workloads, there may not be enough compactions, and hotness-aware compaction alone may not effectively limit the size of the mutable promotion buffer. Therefore, when the size of the mutable promotion buffer grows to the target size of SSTables (64MiB by default), HotRAP converts it to an immutable promotion buffer, and a new mutable promotion buffer will be created (ⓐ). For the example in Figure 2, K_1 and K_2 have been handled by hotness-aware compaction. Therefore, only $K_3, K_5, \ldots$ are packed into an immutable promotion buffer. The immutable promotion

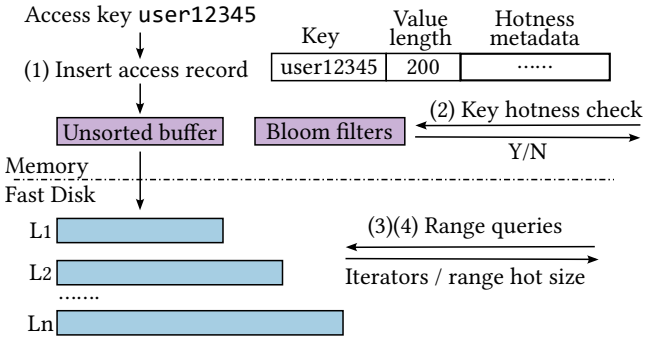


Figure 3: RALT structure. Suppose key `user12345`, whose value length is 200B, is accessed in HotRAP and an access record is inserted to RALT. The figure shows the RALT access record format, as well as the four supported operations (1) to (4). The HotRAP size is len(`user12345`) + 200 = 209 bytes. The physical size is $(9+4)+4+8=25$ bytes, where we use 4 bytes each for the length of the key and value, and 8 bytes for the hotness metadata used by our eviction policy.

buffer consults RALT whether the records are hot (ⓑ). Hot records (K_3) will be flushed to L_0 in FD (ⓒ), while cold records (K_5, K_6) are dropped. To avoid creating tiny SSTables in L_0 which will trigger compactions from L_0 to L_1 prematurely, if the total size of hot records is less than half of the target SSTable size, we insert them back into the mutable promotion buffer instead of flushing to L_0. The original records promoted by flush will remain in SD. Hotness-aware compaction can promote these duplicates to FD and merge them with previously promoted records, thus reducing disk usage.

This section is organized as follows. We first introduce RALT (§3.2, §3.3) and analyze its cost (§3.4). Then we detail the implementation of promotion by flush and its correctness (§3.5, §3.6). Next, we examine the influence of hotness-aware compaction on the benefit-cost scores (§3.7) and write amplification (§3.8). Finally, we analyze the relationship between the promotion buffer and the block cache (§3.9).

3.2 Recent Access Lookup Table (RALT)

RALT is a lightweight LSM-tree on FD, logging the accesses to records in HotRAP. Its structure is shown in Figure 3. Each access record in RALT consists of the key, the length of the value (instead of the value itself), and the scoring metadata tick and score. Keys are considered hot if their scores are greater than a threshold. We distinguish the size of the original key-value record in HotRAP (called the *HotRAP size*) and the size of the RALT access record itself (called the *physical size*). The total HotRAP size of hot records is called the *hot set size*. RALT has two parameters: the *hot set size limit* and the *physical size limit*, which constrain the total size of hot records and the disk usage of RALT itself, respectively. We set their values based on workload characteristics in §3.3.

RALT supports four operations, as shown in Figure 3:

(1) Inserting access records. When HotRAP inserts an access record into RALT, it first goes into an in-memory unsorted buffer. We use an unsorted buffer to improve performance because a sorted memory table does not benefit much: if a key is accessed again while its last access record is still in the buffer, it should be super hot and promoted very fast. When the unsorted buffer becomes full, it is sorted and flushed to FD. There are several levels on FD, following the leveling compaction policy. To avoid blocking reads, RALT maintains multiple versions of the LSM-tree structure.

(2) Checking the hotness of a key. Reading keys from SSTables is expensive. So for each SSTable, RALT stores bloom filters in memory that contain hot keys (keys with scores higher than the threshold). When checking whether a key is hot, RALT checks the bloom filters at each level and returns true if any bloom filter gives a positive result. We use 14-bit bloom filters to achieve a low overall false positive rate ($\ll 1\%$). The small number of false positives does not affect performance. Thus we do not perform a second verification.

(3) Scanning hot keys in a range. During hotness-aware compactions, HotRAP scans hot keys within the compaction range. To achieve this, RALT builds iterators for each level, similar to other LSM-trees. These iterators, which produce hot keys for their respective levels, are then merged into a single iterator that is returned to HotRAP.

(4) Calculating the hot set size in a range. RALT needs to calculate the hot set size in a given range, for HotRAP to select which SSTable to compact (details in §3.7). Similar to a normal LSM-tree, we have data blocks and index blocks for SSTables. For each data block, its first key and the sum of the HotRAP size of hot keys in all previous data blocks are added to the index block. For a range query, at each level, we read the two edge index blocks and calculate the difference of their stored sums, to obtain the total HotRAP size of hot keys in the range. The results of all levels are summed up and returned, as an estimation of the hot set size in the range.

The above result is slightly overestimated because of two error sources. First, the total HotRAP size of hot keys in the one or two edge data blocks is small, so we choose to tolerate the error and not read them. Second, there may be duplicate keys in different levels. If the number of levels is small and the size ratio between levels is large, we can expect the overestimation rate to be small. For example, if the size ratio is 10, the result in the second largest level may be about $1/10$ of the result in the largest level on average. So the overestimation is only about 10%. §3.7 discusses how we handle such overestimation in HotRAP.

3.3 Auto-tuning RALT size limits

In §3.2, RALT takes two parameters: the hot set size limit and the physical size limit, to constrain the total size of hot records and the space of RALT itself, respectively. However,

Algorithm 1 Algorithm for auto-tuning size limits

```
1:  Let the set of tracked keys T ← ∅
2:  for each access k_i do
3:      c_{k_i} ← c_max
4:      if k_i ∉ T then
5:          T ← T ∪ {k_i}
6:          t_{k_i} ← 0
7:      else
8:          t_{k_i} ← 1
9:      end if
10:     if accessed data amount reaches R then
11:         forall k_i ∈ T do c_{k_i} ← max(c_{k_i} − 1, 0)
12:     end if
13:     if hot set size limit or physical size limit is exceeded then
14:         Evict unstable keys with low scores
15:         If not enough, evict stable keys with low scores
16:         % Update size limits.
17:         t ← HotRAP size of stable records
18:         p ← Physical size of stable records
19:         r ← Average ratio between physical and HotRAP size
20:         Hot set size limit ← min(t + D_hs, R_hs)
21:         Physical size limit ← p + rD_hs
22:     end if
23: end for
```

the hot set size is determined by the distribution of the workload, which is usually unknown to the user and may further dynamically vary over time. Therefore, an automatic tuning approach for these size limits is necessary.

We define a key as hot if the expected amount of data accessed (in terms of HotRAP size) between two accesses to the key is below a threshold R. We aim to automatically adjust the hot set size limit and the physical size limit so that hot keys are identified as hot with a high probability. Our algorithm is outlined in Algorithm 1.

We maintain a counter c and a tag t in each RALT access record. Records are deemed *stable* if $c > 0$ and $t = 1$, and *unstable* otherwise. Accessing a key sets $c = c_{\max}$ (Line 3). The first access sets $t = 0$ (Line 5 to 6). A hit on an existing key sets $t = 1$, making the key stable (Line 8). We lazily update counters and tags during compactions. Every time the amount of data accessed (in terms of HotRAP size) reaches R, we decrease all counters by 1, eventually making cold records no longer stable (Line 11). The maximum counter value $c_{\max}$ ensures that keys not reaccessed can be evicted after accessing at most $c_{\max} \cdot R$ data.

When the hot set size or physical size exceeds the limit, RALT evicts 10% of access records, and all access records are merged into a single sorted run. 10% is a good trade-off between the I/O cost and the stability of HotRAP's hit rate. We maintain a score for each access record with exponential smoothing [23]. During evictions, we first evict unstable records with low scores (Line 14). If the size is still too large, stable records with low scores will also be evicted (Line 15).

After each eviction, we update the two limits (Lines 17 to 21). R_{hs} is the upper bound of the hot set size to cap the write amplification of retention. We set $R_{hs} = 0.85 \times$ last level size in FD, allowing a maximum additional SD write amplification of about 3.3 as analyzed in §3.8. D_{hs} is the maximum HotRAP size of unstable records. Essentially, we set the new size limit (HotRAP size at Line 20, or physical size at Line 21) to be the size of stable records plus the maximum size of unstable records, subject to constraints of R_{hs}.

Analysis. Our algorithm ensures that if we set D_{hs} and c_{max} properly and accesses are identically distributed and independent, then almost all hot keys will become stable, while the size of stable cold keys is bounded. We have formally proved these properties, but we omit the proof due to the space limit. Intuitively, because of the tag t, we need to access an originally untracked key at least two times with less than D_{hs} other data accessed in between, to make it stable. Suppose each key k has an access probability of p_k. While the difference between $\sum_{k \text{ is hot}} p_k$ and $\sum_{k \text{ is cold}} p_k$ may not be large enough due to the very large number of cold keys, the difference between $\sum_{k \text{ is hot}} p_k^2$ and $\sum_{k \text{ is cold}} p_k^2$ is typically large enough to separate them apart.

We set $R =$ FD size, $D_{hs} = 0.05 \times R$, $c_{max} = 5$. With these parameters, the probability that keys with access probability $> \frac{1}{0.5 \times \text{FD size}}$ become stable is $\approx 99.8\%$. The HotRAP size of cold keys with access probability $< \frac{1}{6 \times \text{FD size}}$ is smaller than $0.1 \times$ FD size. Therefore, these parameters should work well under most workloads. The experiments are shown in §4.6.

3.4 Cost analysis of RALT

Disk and memory usage. Suppose the HotRAP record size is 200B, and the key length is 24B with 5% hot keys. The disk usage of RALT is about $5\% \times (24+16)/200 = 1\%$ of the data size. Bloom filters and index blocks are cached in memory so that checking key hotness and calculating range hot size do not incur random disk I/O. The memory usage of the bloom filters in RALT is $5\% \times 14/8/200 = 0.0438\%$ of the data size. An index block record is less than 40B, and a data block in RALT is 16KiB. Therefore, the memory usage of the index blocks is $5\% \times 40/16/1024 = 0.0122\%$ of the data size. In total, the memory usage is only 0.056% of the data size. By storing RALT on disk instead of in memory, we can save $1 - 0.056\%/1\% \approx 95\%$ of memory.

I/O cost. Suppose RALT has N_L levels, and the size ratio is T. Since we evict 10% of data each time, the total write amplification is $\frac{T}{2}N_L + \frac{1}{10\%}$. The total read amplification is $\frac{T}{2}N_L + \frac{2}{10\%}$ because we need two full scans to calculate the score threshold. In our experiments in §4.4, $T = 10, N_L \approx 2$, the read amplification is about 30 and the write amplification is about 20. The experiment results show that RALT only accounts for 5.2%–9.7% of the total I/O, because a RALT record does not contain the value in a HotRAP record, therefore the total I/O of RALT is small compared to HotRAP.

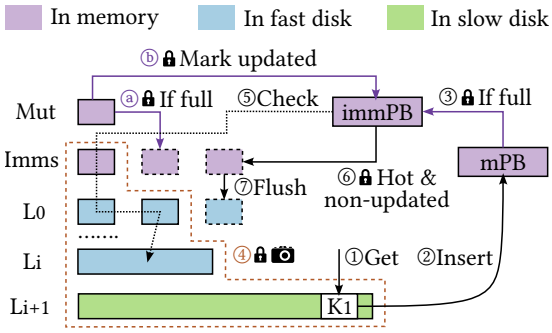


Figure 4: Concurrency control of promotion by flush. Processes with lock icons are protected by the DB mutex lock. ⑤ ensures that no newer versions exist in the snapshot of the LSM-tree. ⓐ and ⓑ mark all updated keys in immutable promotion buffers. Records with updated keys are excluded in ⑥. The snapshot is taken (④) after the creation of the immutable promotion buffer (③), therefore, a key updated before ⑥ is either found out by ⑤ or by ⓐ and ⓑ.

3.5 Checks before promotion buffer insertion

The promotion buffer resides between FD and SD. Therefore, before promoting a record into the promotion buffer, it is crucial to verify the absence of a newer version of this record in SD, which would otherwise be shielded by the inserted older record in the promotion buffer. For a point read that wants to insert the latest version just retrieved from SD into the promotion buffer, it is known that no newer version exists in the same superversion (i.e., the LSM-tree's snapshot). However, it is still possible that a newer version of the record is compacted into SD before the previous record has finished being inserted into the promotion buffer.

To address this issue, HotRAP marks SSTables as being or having been compacted when setting up compaction jobs. When searching for the value of a key in SD levels, SD SSTables whose range contains the key are recorded. Before inserting the key-value pair into the promotion buffer, HotRAP checks if any of the recorded SSTables is being or has been compacted. If there is, HotRAP aborts the insertion. The abort rate is low because of the small number of compaction jobs. Our experiments show that this check only aborts less than 1% of insertions into the promotion buffer.

3.6 Concurrency control of promotion by flush

Now we discuss the details of promotion by flush in Figure 4. As mentioned before, a read to SD record (①) inserts it to the mutable promotion buffer (②), which may be later made immutable when full (③). At this point, a snapshot is taken by incrementing the reference count of the caller's superversion (④). We pass the immutable promotion buffer's reference and the superversion's reference to a background thread called *Checker*, which handles the rest part. This minimizes the impact on foreground reads. *Checker* then picks out hot records in the immutable promotion buffer by consulting RALT. These hot records are candidates to be flushed to L_0.

Proper concurrency control is needed to avoid the issue that flushing a hot record to L_0 may shield a newer version of this key in the FD levels. *Checker* looks for newer versions of the hot records in the snapshot's immutable MemTables and the levels in FD (⑤). *Checker* only checks bloom filters for levels in FD for speed. The key is marked updated[2] if any filter returns a positive result. Hot records without possible newer versions are packed into a new immutable MemTable (⑥)[3]. Those records will eventually be flushed into L_0 (⑦).

However, there is still a corner case: newer versions can be flushed into L_0 in the normal access path when HotRAP is looking for newer versions during step ⑤. To address this issue, when a mutable MemTable becomes immutable during normal accesses (ⓐ), for every record in it, HotRAP checks whether the same key exists in immutable promotion buffers. If so, HotRAP marks the key as updated[2] (ⓑ), and the record will not be packed into the immutable MemTable at step ⑥.

We create immutable promotion buffers with the DB mutex (the only major lock in RocksDB) held at step ③. Since flushes are protected by the DB mutex, there are only two possible cases and both are correct: (1) An immutable MemTable is created before an immutable promotion buffer is created. The newer versions of records in the immutable MemTable will be detected by the check of ⑤. (2) An immutable promotion buffer is created before an immutable MemTable is created. The newer versions of records in the immutable MemTable will be marked updated by ⓐ and ⓑ.

3.7 Cost-benefit analysis of compaction

RocksDB calculates a score for each SSTable and picks the one with the highest score to compact into the next level. By default, the score is defined as $FileSize/OverlappingBytes$, where *OverlappingBytes* represents the total size of the target level's SSTables whose key ranges overlap with the picked SSTable. This is essentially a cost-benefit trade-off score: *FileSize* is the benefit, and $(FileSize + OverlappingBytes)$ is the cost, in which *FileSize* is optimized out.

However, the cost-benefit score needs some adjustment to better support HotRAP. During a cross-tier compaction from FD to SD in HotRAP, hot records in the chosen SSTable will be retained in the source level. Therefore, the benefit becomes (*FileSize* - *HotSize*), and the cost is still (*FileSize* + *OverlappingBytes*). Their ratio is the new cost-benefit score.

HotRAP estimates the HotSize of an SSTable by querying RALT about the hot set size in the corresponding range (§3.2). Recall that the obtained HotSize is an overestimation.

[2] By inserting the key into the *updated* field attached to the immutable promotion buffer.

[3] Or inserted back into the mutable promotion buffer if there are too few of them.

Therefore, it is possible, although very unlikely, that all benefit values are zero. In such cases, HotRAP falls back to choosing the oldest SSTable for compaction.

3.8 Write amplification of retention

Suppose the fraction of cold data in the last level of FD is p. Since each compaction from FD to SD only compacts p of the selected data to SD and writes the remaining $(1-p)$ of data back to FD, $\frac{1}{p}\times$ more compactions are needed to compact the same amount of data. Therefore, suppose the size ratio of the LSM-tree is T, FD and SD respectively have n_{FD} and n_{SD} levels, the write amplification in FD is $\frac{T}{2}n_{FD}+\frac{1-p}{p}$ and the write amplification in SD is $\frac{T}{2p}+\frac{T}{2}(n_{SD}-1)$, which are $\frac{1-p}{p}$ and $\frac{T}{2p}-\frac{T}{2}$ larger than a normal LSM-tree, respectively.

Write amplification can be lowered by tuning the level size ratios. Specifically, we can shrink the first level of SD to make the size ratio between the last level of FD and the first level of SD be pT. To keep the size of the LSM-tree in SD unchanged, we can add an extra level after the last level with a size ratio of $\frac{1}{p}$. The size ratio between other levels in SD remains T. In this way, the write amplification in SD is $\frac{T}{2}n_{SD}+\frac{1}{2p}$, which is only $\frac{1}{2p}$ larger than a normal LSM-tree.

3.9 The promotion buffer and block cache

The promotion buffer and the block cache complement each other. When a record is in the promotion buffer, its readers are served by the promotion buffer. After being flushed into FD, its readers are served by the block cache. Therefore, the promotion buffer serves reads to SD, while the block cache mainly serves reads to FD. Our design is orthogonal to Leaper [38] and Range Cache [32], i.e., their optimization techniques can also be applied to HotRAP, and by promoting hot records to FD with our techniques, records evicted from their cache will be read from FD instead of SD.

The main function of the promotion buffer is to batch records from SD before flushing them to L0, to avoid many tiny SSTables in L0. Serving readers is only the secondary function of the promotion buffer to avoid accessing SD when the record is already in the promotion buffer but must wait for more records to be batch-flushed into FD. Therefore, we fix its size to the target size of SSTables (64MiB by default). As a result, the promotion buffer contributes less to the hit rate when the hotspot is larger. However, we do not need to increase the promotion buffer size for a larger hotspot (either due to higher hotspot percentage or larger dataset), because the record's readers will be served by the larger block cache after it is flushed to FD.

Table 2: Disk performance on our EC2 instances.

	Fast Disk	Slow Disk
Type	AWS Nitro SSD	gp3
16 threads rand 16K read IOPS	≈83000	10000 [4]
Sequential read bandwidth	≈1.4GiB/s	300MiB/s[4]
Sequential write bandwidth	≈1.1GiB/s	300MiB/s[4]

Table 3: Read-write ratios of YCSB workloads in our tests.

Notation	Meaning	Read-write ratio
RO	read-only	100% read
RW	read-write	75% read, 25% insert
WH	write-heavy	50% read, 50% insert
UH	update-heavy	50% read, 50% update

4 Evaluation

4.1 Experimental setup

Testbed. We evaluate HotRAP on AWS EC2 i4i.2xlarge instances running Debian 12. Each instance has 8 vCPU cores, 64GiB memory, and a 1875GB local AWS Nitro SSD. We use local SSDs as FD and gp3 as SD. Their performance characteristics are shown in Table 2.

Sizes of tiers. We set the space ratio between tiers to 10: the initial expected used size of SD and FD is set to 100GB and 10GB respectively, and the memory budget is 1GB. HotRAP is also evaluated with a larger dataset in §4.7.

Compared systems. We compare HotRAP with PrismDB [31], SAS-Cache [9], and the following three variants of RocksDB: RocksDB-FD, RocksDB-tiering, and RocksDB-CL. RocksDB-FD stores all data in FD, which is used to indicate the upper-bound performance that HotRAP can achieve. RocksDB-tiering tunes its size ratios between levels so that the total size of FD levels becomes 10GB, which is the same as HotRAP. PrismDB is also tuned to use about 10GB of FD expectedly. RocksDB-CL caches records on FD using CacheLiB [8], while SAS-Cache caches blocks on FD. Among the baseline systems, RocksDB-tiering and PrismDB use the tiering design like HotRAP, while RocksDB-CL and SAS-Cache use the caching design.

Configurations. All experiments run with 16 threads. To minimize the impact of the file system cache, direct I/O is used in all systems. We set the initial hot set size limit and RALT physical size limit to 50% and 15% of the FD size respectively. HotRAP is configured with a 256MiB block cache, while other systems are configured with 64MiB more block cache to compensate for the memory usage of RALT. The 256+64=320MiB block cache uses about 1/3 of our 1GB memory budget, which is recommended by the RocksDB tuning guide [29]. Other configurations are set following the RocksDB tuning guide [29], e.g., 16KiB block size, 10-bit bloom filters, and 6 maximum number of background jobs.

[4] The maximum sustainable IOPS and throughput of gp3.

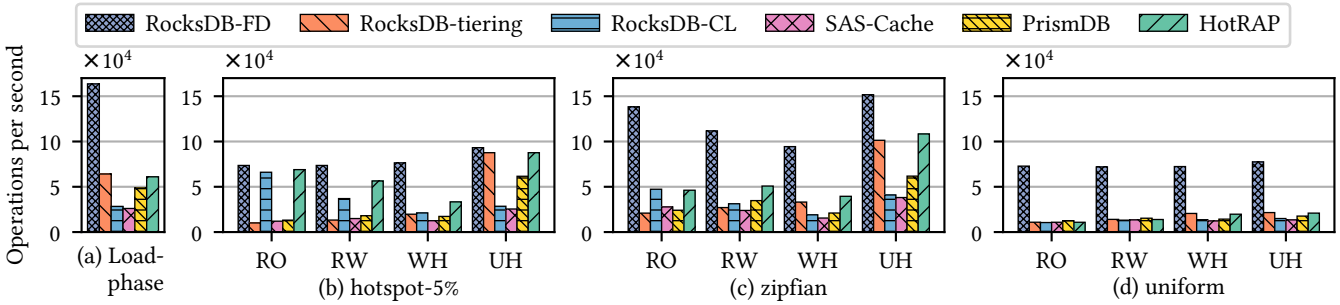


Figure 5: Throughput comparison with 1KiB record size.

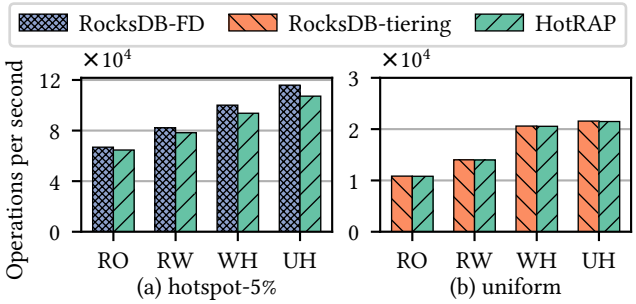


Figure 6: Throughput comparison with 200B record size.

4.2 Performance on YCSB workloads

To assess the performance of HotRAP under various key distributions and read-write ratios, we evaluate HotRAP with the YCSB workloads [10] shown in Table 3. *RO* (read-only) tests the effectiveness of promotion by flush. *RW* (read-write) and *WH* (write-heavy) test the effectiveness of hot-awareness compaction. *UH* (update-heavy) is the worst case for HotRAP. With UH, the key distributions of reads and updates are the same. Therefore, newer versions of read-intensive records are frequently inserted into the database and flushed into FD, making the proactive promotion of HotRAP barely needed.

We test three skewness types: hotspot-5%, Zipfian, and uniform. In the hotspot-5% distribution, 5% of records are accessed by 95% operations uniformly. The other 5% operations uniformly access the other 95% of records. In the Zipfian distribution, the access probability of the k-th hottest record is proportional to $1/k^s$ [7]. In our experiments, $s = 0.99$. In the uniform distribution, the access probability of all records is the same. We evaluate two record sizes, 1KiB (≈24B key and 1000B value) and 200B (≈24B key and 176B value).

All workloads have a load phase and a run phase. The load phase loads 110GB of records into the LSM-tree. The run phase executes read/write operations. For workloads with 1KiB records, the run phase executes 2.2×10^8 operations. For 200B records, the run phase executes 1.1×10^9 operations.

Figures 5 and 6 compare the average throughput (over the final 10% for the run phase) of evaluated systems under different read-write ratios and skewness types, with 1KiB and 200B records, respectively. Since the trends are similar, we only show a representative subset in Figure 6 to save space: hotspot-5% to show near-optimal efficiency compared to RocksDB-FD, uniform to show low overhead compared to RocksDB-tiering.

During the load phase, HotRAP's behavior is the same as RocksDB-tiering; thus, their performance is the same. We next focus on the run phase. The performance of HotRAP when running non-write-heavy hotspot-5% workloads is close to that of the ideal RocksDB-FD, because HotRAP promotes almost all hot data into FD and achieves about 95% hit rate. Compared to other systems, HotRAP achieves 5.2× speedup over the second best baseline with the tiering design under read-only (RO) workloads, 2.1× speedup over the caching design under write-heavy (WH) workloads, and 1.6× speedup over both designs for read-write-balanced (RW) workloads. Additionally, HotRAP matches the performance of RocksDB-CL for read-only (RO) workloads, and RocksDB-tiering for update-heavy (UH) workloads. On the other hand, HotRAP is only 4.0% slower than RocksDB-tiering under uniform workloads, showing the overhead of HotRAP is low when promotion has no benefits. For the Zipfian distribution, there is a noticeable gap in Figure 5 between HotRAP and the upper-bound RocksDB-FD, because the hit rate is lower (79%) compared to hotspot-5%. There is also a gap between HotRAP and RocksDB-FD under write-heavy (WH) workloads, because compactions saturate SD[5]. But HotRAP still outperforms other designs in these scenarios.

For the baseline designs, SAS-Cache shows negligible improvements over RocksDB-tiering, as its block-level caching is too coarse. PrismDB also only has small improvements over RocksDB-tiering due to its inefficient promotion mechanism. While RocksDB-CL performs comparably to HotRAP under read-only (RO) workloads, its performance degrades under other workloads (RW, WH, UH) due to SD being overwhelmed by compactions.

Among different read-write ratios, we notice that systems using the tiering design exhibit significantly higher throughput

[5]As hot records progressively migrate to FD, read latency reductions create proportional write throughput gains due to the workload's fixed 1:1 read-write ratio. This synchronized acceleration ultimately triggers excessive compactions that saturate SD's throughput ceiling at 300 MiB/s.

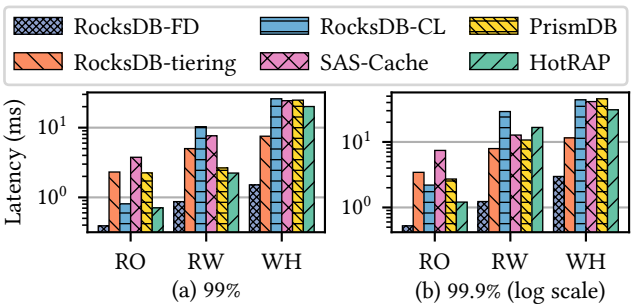


Figure 7: *Get* tail latency comparison under hotspot-5% workloads with 1KiB record size.

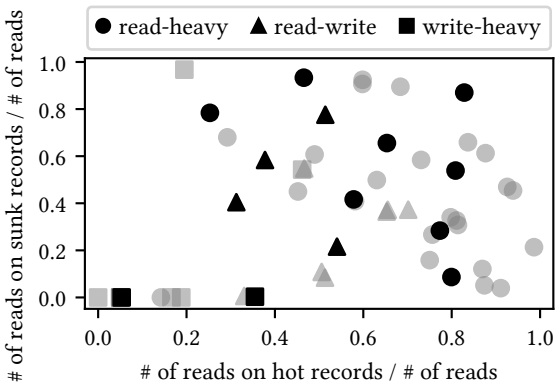


Figure 8: Characteristics of Twitter production traces. Each point stands for a cluster's trace. Dark black points are our selected traces for evaluation in Figure 9.

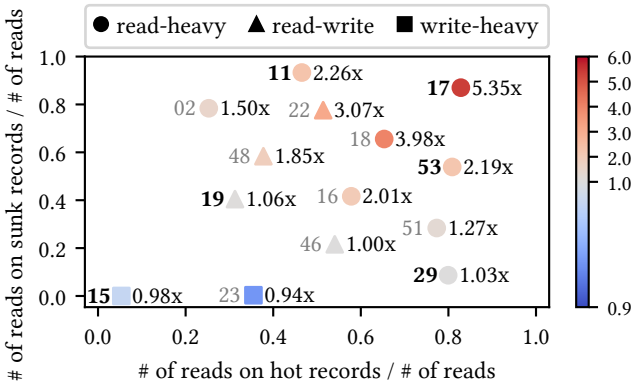


Figure 9: Speedup of HotRAP over RocksDB-tiering on Twitter production traces. Numbers on the two sides of a point are the cluster ID and the speedup. Traces with bold cluster IDs are selected for further analysis in Figure 10.

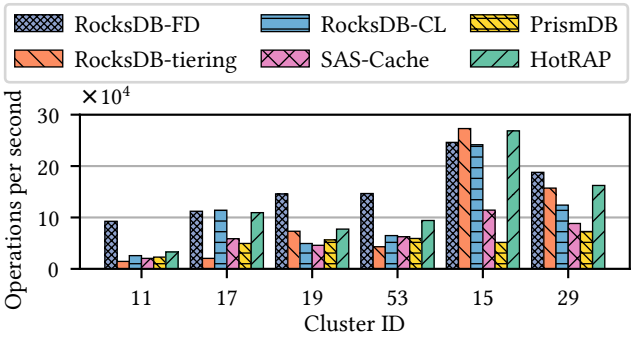


Figure 10: Throughput comparison under several Twitter production traces.

under update-heavy (UH) workloads. This is because the updated data have a skewed distribution, and thus update operations can be considered as promotions to FD. However, PrismDB performs relatively poorly in this case, because it suffers from lock contention and inefficient random writes to FD. RocksDB-CL and SAS-Cache barely benefit from the update-heavy workloads because they use the caching design and all levels are stored in SD.

Figure 7 shows the tail latencies (again over the final 10% of the run phase) under hotspot-5% workloads with 1KiB record size. For read-only (RO) workloads, HotRAP achieves lower tail latency than other systems except RocksDB-FD. It is because HotRAP has a higher FD hit rate and thus reduces accesses to SD, so there are proportionally fewer long-latency accesses affected by the SD tail latency among all FD and SD accesses. In contrast, under write-heavy workloads (WH), HotRAP has a higher tail latency than RocksDB-tiering. We believe this is because the throughput of HotRAP is higher than RocksDB-tiering, therefore compactions in HotRAP are more frequent, which deteriorates the tail latency.

4.3 Performance on real-world Twitter traces

To show the performance of HotRAP under real-world workloads, we evaluate HotRAP with the Twitter production traces [37]. We pre-process every trace into two phases: the load phase and the run phase. In the load phase, we ignore reads and keep inserting about 110GB of data. In the run phase, we execute 5×10^8 operations. We augment small traces by repeating operations until reaching 110GB of data. For example, if a trace has 40GB of data, we repeat each operation 3 times, e.g., an operation "`INSERT user1`" becomes "`INSERT 0user1, INSERT 1user1, INSERT 2user1`".

Traces are categorized based on their read proportions: read-heavy has >75% reads; read-write has >50% and ≤75% reads; write-heavy has ≤50% reads. In some traces, it is common for a frequently read key to be also frequently updated. Proactively promoting this key has little benefit because its newer version will be automatically inserted into FD. We define that a read is performed on a *sunk* record if the data amount written since the last update of its key exceeds 5% of the DB size, so that the latest version is now likely sunk to SD when being read. For example, suppose we first update keys A, C, D, B, then we read key A. If the total size of records C, D, B surpasses 5% of the DB size, then we say A is sunk when being read. Similarly, we define that a read is performed on a *hot* record if the data amount read since the last read of

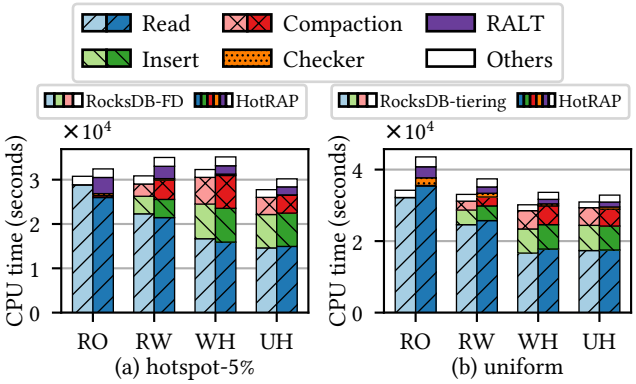


Figure 11: CPU time breakdown with 200B record size.

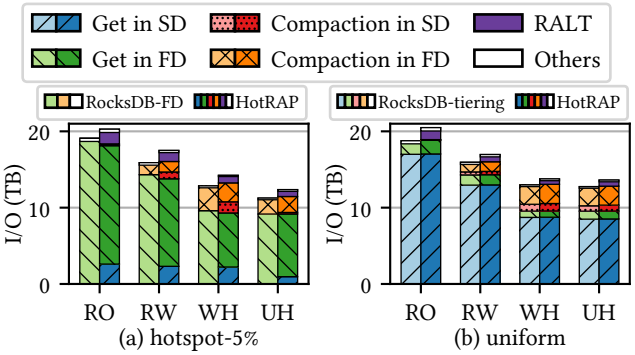


Figure 12: I/O breakdown with 200B record size.

this record is less than 5% of the DB size, so that the record is likely to be identified as hot. If there are many reads on such sunk and hot records, then promoting them to FD is necessary.

Figure 8 shows the categories and proportion of reads on sunk/hot records in the Twitter traces. We select some representative traces to evaluate the speedup of HotRAP over RocksDB-tiering in Figure 9. HotRAP performs better when the proportions of reads on sunk and hot records are higher, e.g., achieving up to 5.35× speedup under the trace of cluster 17. On the other hand, HotRAP is not significantly slower than RocksDB-tiering under traces with low proportions of reads on sunk and hot records, showing its low overhead.

We further analyze several traces with high (11, 17), medium (19, 53), and low (15, 29) proportions of sunk record reads, and present their throughput in Figure 10. HotRAP is almost always the best among compared systems, achieving up to 1.5× speedup over the second best. As the proportion of sunk record reads decreases, almost all systems perform better, but HotRAP's relative speedup diminishes. At medium and high sunk record read proportions, if we increase the amount of hot record reads, HotRAP throughput would significantly increase, while PrismDB and SAS-Cache perform moderately better, and RocksDB-tiering does not benefit from it at all. Although RocksDB-CL performs well under workloads with few writes like cluster 17, it underperforms under other workloads.

4.4 Cost breakdown

Figures 11 and 12 show CPU time and I/O breakdowns for the run phase of YCSB workloads with 200B record size. The size of RALT here exceeds our 1GB memory budget, and thus needs to be stored in FD instead of memory. The results show that RALT accounts for only 3.7%–11.2% of total CPU time and 5.2%–9.7% of total I/O. The I/O cost of RALT[6] is background FD I/O. Since the throughput of FD is abundant, the I/O cost of RALT barely affects the overall system performance.

[6]Measured by maintaining the number of bytes read/written in the file system interface wrapper.

Table 4: Costs with/without hotness-aware compaction under the RW hotspot-5% workload with 1KiB record size.

Version	Promoted	Compaction	Hit rate	Disk usage
HotRAP	5.5GB	2092.5GB	94.8%	170.0GB
no-hot-aware	35.8GB	2841.8GB	72.0%	178.6GB

In hotspot-5% workloads, HotRAP incurs more CPU time and I/O on compactions than RocksDB-FD, because retention increases write amplification. Interestingly, HotRAP uses less CPU time and I/O on reads than RocksDB-FD. We believe it is because hot records are promoted to higher levels, so there are fewer levels to probe to read those records. In uniform workloads, HotRAP consumes more CPU time than RocksDB-tiering because most accesses are in SD and thus the records are inserted into the promotion buffer. However, few records are promoted into FD due to the hotness checking (⑤ & ⓑ in Figure 2), therefore they have similar compaction I/O.

4.5 Effectiveness of individual techniques

Hotness-aware compaction. We disable the hotness-aware compaction mechanism in HotRAP and call the design *no-hot-aware*. Table 4 shows that *no-hot-aware* incurs higher promotion costs and achieves a lower final hit rate compared to HotRAP. The reason is that although *no-hot-aware* still promotes records into FD by flush, the promoted records are compacted into SD again during subsequent compactions. Consequently, hot records have to be promoted repeatedly, with much more promotion traffic. Additionally, HotRAP incurs less disk usage than *no-hot-aware*. It is because hotness-aware compaction gradually promotes duplicate data (left in SD by promotion by flush) into FD and merges them with previously promoted records.

Promotion by flush. We disable the promotion by flush mechanism and call the design *no-flush*. Figure 13 shows that the hit rate increases very slowly without promotion by flush, especially for read-heavy workloads. In contrast, the hit rate of

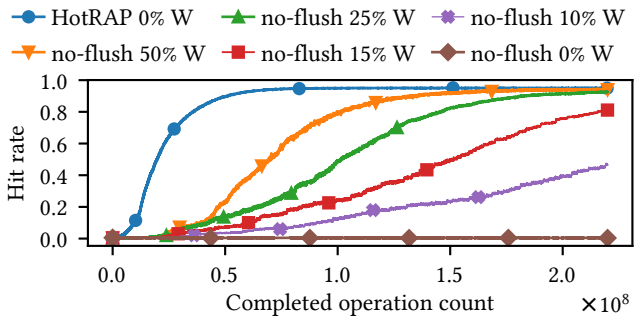


Figure 13: Effectiveness of promotion by flush. $x\%$ W means $x\%$ of operations are writes and $(1-x)\%$ are reads.

Table 5: Promotion costs with/without hotness checking under the RO uniform workload with 1KiB record size.

Version	Promoted	Retained	Compaction
HotRAP	1.0GB	15.7MB	34.8GB
no-hotness-check	200.6GB	4.3GB	5870.3GB

HotRAP increases rapidly even under the read-only workload.

Hotness checking before promotion. HotRAP promotes only hot records to reduce the overhead introduced by promotion. To show its effectiveness, we remove the hotness checking and promote all accessed records, and call the design *no-hotness-check*. Table 5 shows that under uniform workloads, *no-hotness-check* promotes 204.1× more records and thus incurs 167.7× more compaction I/O than HotRAP.

4.6 Performance on dynamic workload

To show that HotRAP can adapt to changes in the access pattern, we evaluate it under a dynamic workload. The details of the dynamic workload and the results are shown in Figure 14.

The first stage has a uniform distribution, so there are few stable records, and the hot set size stays low. At the second stage, the key distribution becomes hotspot-2%. With the auto-tuning method, HotRAP gradually increases the hot set size limit until all hotspot keys are added to the hot set. Eventually, the hot set size stabilizes around the hotspot size. After the hotspot expands from 2% to 4% and from 4% to 6%, the FD hit rate temporarily drops because new hot keys are not yet promoted. Then HotRAP gradually increases the hot set size limit until all new hot keys are added to the hot set, thus recovering the hit rate. After the hotspot expands from 6% to 8%, the hotspot size exceeds the max hot set size (≈ 7GB). Therefore, the performance of HotRAP is relatively low. When the hotspot shifts, HotRAP reacts adaptively by evicting old hot keys after they become unstable, and gradually adding new hot keys to the hot set. Both the throughput and the hit rate recover eventually. After the hotspot shrinks from 5% to 3% and from 3% to 1%, the throughput and the hit rate do not drop because new hot keys are already considered hot

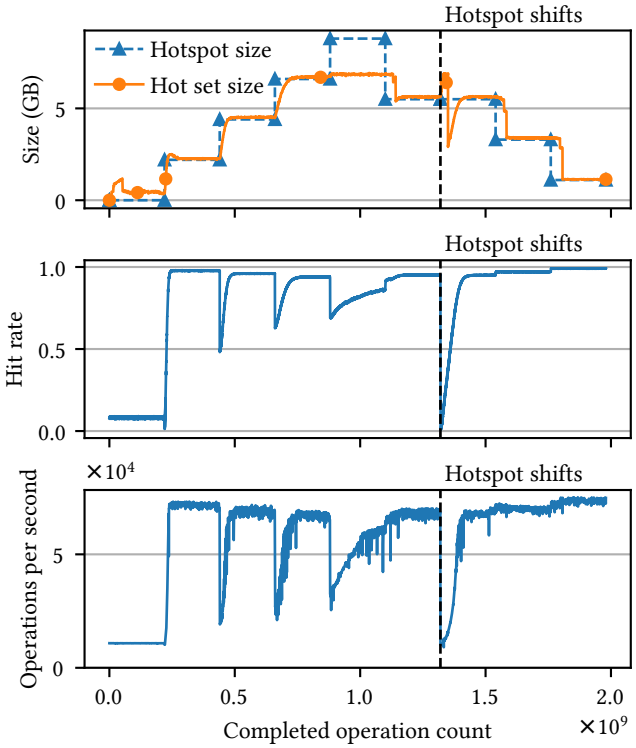


Figure 14: HotRAP under dynamic workload. The run phase consists of nine stages, whose data distributions are first uniform, then hotspot-2% → 4% → 6% → 8% → 5% → 5% → 3% → 1%. Each stage executes 2.2×10^8 read operations. The two 5% hotspots in the 6th and 7th stages are non-overlapping. When the hotspots increase from 2% to 8%, the new hotspot completely contains the old hotspot. When they decrease, the new one is completely contained by the old one.

and have already been promoted. Nevertheless, HotRAP in this case could decrease the hot set size limit after old access records become unstable.

In summary, the results show that the auto-tuning mechanism enables HotRAP to find the most suitable hot set size limit under a dynamic workload with hotspot expanding, shrinking, and shifting.

4.7 Large dataset

Figure 15 compares the average throughput (over the final 10% of the run phase) of RocksDB-FD, RocksDB-tiering, and HotRAP under different read-write ratios and skewness types, with 1.1TB datasets and 1KiB record size. HotRAP is configured with a 2GiB block cache while other systems are configured with 1GiB more block cache to compensate for the memory usage of RALT. The run phase of HotRAP executes 2.2×10^9 operations. The run phase of RocksDB-FD and RocksDB-tiering executes 1.1×10^9 operations to save test time since their performance barely changes during the run phase. The results are similar to Figure 5, showing the

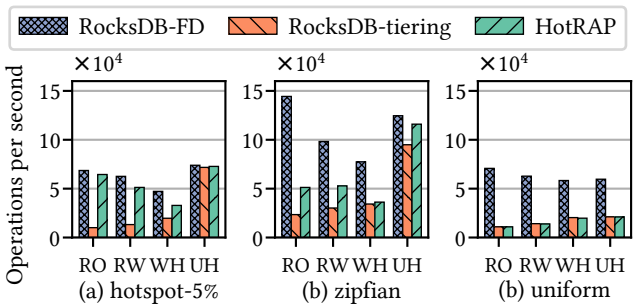


Figure 15: Throughput comparison with 1KiB record size and 1.1TB datasets (1TB in SD and 100GB in FD).

Table 6: Comparison with Range Cache under the read-only Zipfian YCSB workload with 1KiB record size. *OPS* means *Operations Per Second*.

System	OPS	FD IOPS	SD IOPS
RocksDB-tiered	20927	1311	9998
Range Cache	29217	1019	9995
HotRAP	46156	14632	9998
HotRAP + Range Cache	46180	8443	9986

scalability of HotRAP.

4.8 Comparison with Range Cache

Range Cache [32] addresses the block-cache invalidation problem by introducing a separate in-memory key-value cache on top of the LSM-tree. Since Range Cache is not open-source and this paper focuses on point query performance, we simulate Range Cache by enabling the row cache in RocksDB. Table 6 shows the experimental results collected during the final 10% of the run phase. Range Cache improves throughput compared to RocksDB-tiered by finer-grained caching. However, the in-memory cache is insufficient to hold all hot records due to the limited memory capacity; therefore, many accesses to SD still occur. HotRAP outperforms Range Cache by promoting all hot records to FD, which is possible due to the large capacity of FD. Combining Range Cache with HotRAP further alleviates the load of FD by caching some hot records in memory. The throughput is not improved because the performance remains bottlenecked by SD.

5 Discussion

Scan workloads. HotRAP behaves the same as RocksDB-tiering under scan workloads: scanned records are not inserted into RALT or the promotion buffer, and there is no metadata lookup for RALT during scans. It is our future work to speed up range scans by maintaining the set of promoted ranges so that a range scan does not need to read SD if it is in a promoted range. Details will be revealed in our future paper.

Limitation of the auto-tuning algorithm. The algorithm keeps unstable records of HotRAP size D_{hs} to detect hot records. It assumes that hot records are accessed randomly, so every hot record has a probability > 0 of being detected. However, if a hot record is accessed only once every $> D_{hs}$ accesses, it will not be captured. In extreme cases such as "sequential flooding", the algorithm cannot work unless D_{hs} is increased. But this is uncommon in real-world datasets (e.g., the Twitter traces in §4.3).

Portability. The key HotRAP techniques (RALT and promotion by flush) can be easily applied to other modern LSM-tree implementations. (1) RALT is implemented as a standalone library with a minimum interface (insert, key hotness check, iterate, and range hot size) to ensure cross-system portability. (2) Promotion by flush leverages the multi-version concurrency control (MVCC) of the LSM-tree structure to minimize the impact on foreground tasks. The MVCC capability should be available in modern LSM-tree implementations.

6 Conclusion

We introduced HotRAP, an LSM-tree-based key-value store on tiered storage. Unlike previous solutions, HotRAP adopts an efficient on-disk hotness tracker, along with a fine-grained record-level retention and promotion mechanism that offers two pathways for hot records to be stored in the fast tier. These techniques allow HotRAP to efficiently manage data across tiers to fully utilize the fast tier under various workloads.

Acknowledgments

We thank our shepherd, Suzhen Wu, and the anonymous reviewers for their constructive comments. This work was partially supported by the Shanghai Qi Zhi Institute Innovation Program (SQZ202406 & SQZ202314).

References

[1] Apache cassandra, 2009. https://cassandra.apache.org.

[2] Scylladb, 2015. https://github.com/scylladb/scylladb.

[3] Amazon ec2 instance types, 2024. https://aws.amazon.com/ec2/instance-types/.

[4] Samsung pm9a3 2.5" u.2 7.68tb pcie 4.0 x4 nvme 1.4 v-nand tlc enterprise solid state drive, 2024. https://www.newegg.com/p/N82E16820147858?Item=9SIA12KJA14195.

[5] Seagate exos x20 st20000nm007d 20tb 7200 rpm 256mb cache 3.5" internal hard

drive, 2024. https://www.newegg.com/seagate-exos-x20-st20000nm007d-20tb/p/N82E16822185011?Item=N82E16822185011.

[6] Tikv is a highly scalable, low latency, and easy to use key-value database, 2024. https://tikv.org/.

[7] Zipf's law, 2024. https://en.wikipedia.org/wiki/Zipf%27s_law.

[8] Benjamin Berg, Daniel S Berger, Sara McAllister, Isaac Grosof, Sathya Gunasekar, Jimmy Lu, Michael Uhlar, Jim Carrig, Nathan Beckmann, Mor Harchol-Balter, et al. The cachelib caching engine: Design and experiences at scale. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 753–768, 2020.

[9] Zhang Cao, Chang Guo, Ziyuan Lv, Anand Ananthabhotla, and Zhichao Cao. Sas-cache: A semantic-aware secondary cache for lsm-based key-value stores. In *The 38th International Conference on Massive Storage Systems and Technology (MSST 2024)*, 2024.

[10] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154, 2010.

[11] Niv Dayan, Tamar Weiss, Shmuel Dashevsky, Michael Pan, Edward Bortnikov, and Moshe Twitto. Spooky: granulating lsm-tree compactions correctly. *Proceedings of the VLDB Endowment*, 15(11):3071–3084, 2022.

[12] Siying Dong, Mark Callaghan, Leonidas Galanis, Dhruba Borthakur, Tony Savor, and Michael Strum. Optimizing space amplification in rocksdb. In *CIDR*, volume 3, page 3, 2017.

[13] Siying Dong, Andrew Kryczka, Yanqin Jin, and Michael Stumm. Evolution of development priorities in key-value stores serving large-scale applications: The {rocksdb} experience. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 33–49, 2021.

[14] Siying Dong, Andrew Kryczka, Yanqin Jin, and Michael Stumm. Rocksdb: Evolution of development priorities in a key-value store serving large-scale applications. *ACM Transactions on Storage (TOS)*, 17(4):1–32, 2021.

[15] Siying Dong, Shiva Shankar P, Satadru Pan, Anand Ananthabhotla, Dhanabal Ekambaram, Abhinav Sharma, Shobhit Dayal, Nishant Vinaybhai Parikh, Yanqin Jin, Albert Kim, et al. Disaggregating rocksdb: A production experience. *Proceedings of the ACM on Management of Data*, 1(2):1–24, 2023.

[16] Google. Leveldb, 2011. https://github.com/google/leveldb.

[17] Jorge Guerra, Himabindu Pucha, Joseph Glider, Wendy Belluomini, and Raju Rangaswami. Cost effective storage using extent based dynamic tiering. In *9th USENIX Conference on File and Storage Technologies (FAST 11)*, 2011.

[18] Dongxu Huang, Qi Liu, Qiu Cui, Zhuhe Fang, Xiaoyu Ma, Fei Xu, Li Shen, Liu Tang, Yuxing Zhou, Menglong Huang, et al. Tidb: a raft-based htap database. *Proceedings of the VLDB Endowment*, 13(12):3072–3084, 2020.

[19] Junsu Im, Jinwook Bae, Chanwoo Chung, Sungjin Lee, et al. Pink: High-speed in-storage key-value store with bounded tails. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 173–187, 2020.

[20] HV Jagadish, PPS Narayan, Sridhar Seshadri, S Sudarshan, and Rama Kanneganti. Incremental organization for data recording and warehousing. In *VLDB*, pages 16–25, 1997.

[21] Hiwot Tadese Kassa, Jason Akers, Mrinmoy Ghosh, Zhichao Cao, Vaibhav Gogte, and Ronald Dreslinski. Improving performance of flash based key-value stores using storage class memory as a volatile memory extension. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 821–837, 2021.

[22] Cockroach Labs. Cockroachdb, 2015. https://github.com/cockroachdb/cockroach.

[23] Justin J Levandoski, Per-Åke Larson, and Radu Stoica. Identifying hot and cold data in main-memory databases. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pages 26–37. IEEE, 2013.

[24] Yoshinori Matsunobu, Siying Dong, and Herman Lee. Myrocks: Lsm-tree database storage engine serving facebook's social graph. *Proceedings of the VLDB Endowment*, 13(12):3217–3230, 2020.

[25] Sara McAllister, Benjamin Berg, Julian Tutuncu-Macias, Juncheng Yang, Sathya Gunasekar, Jimmy Lu, Daniel S Berger, Nathan Beckmann, and Gregory R Ganger. Kangaroo: Caching billions of tiny objects on flash. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 243–262, 2021.

[26] Prashanth Menon, Thamir M Qadah, Tilmann Rabl, Mohammad Sadoghi, and Hans-Arno Jacobsen. Logstore: A workload-aware, adaptable key-value store on hybrid storage systems. *IEEE Transactions on Knowledge and Data Engineering*, 34(8):3867–3882, 2020.

[27] Meta. Rocksdb: A persistent key-value store for flash and ram storage, 2012. https://github.com/facebook/rocksdb/.

[28] Meta. Secondarycache (experimental), 2022. https://github.com/facebook/rocksdb/wiki/SecondaryCache-(Experimental).

[29] Meta. Setup options and basic tuning, 2022. https://github.com/facebook/rocksdb/wiki/Setup-Options-and-Basic-Tuning.

[30] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. The log-structured merge-tree (lsm-tree). *Acta Informatica*, 33:351–385, 1996.

[31] Ashwini Raina, Jianan Lu, Asaf Cidon, and Michael J Freedman. Efficient compactions between storage tiers with prismdb. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pages 179–193, 2023.

[32] Xiaoliang Wang, Peiquan Jin, Yongping Luo, and Zhaole Chu. Range cache: An efficient cache component for accelerating range queries on lsm-based key-value stores. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*, pages 488–500. IEEE, 2024.

[33] Zhiqi Wang and Zili Shao. Mirrorkv: An efficient key-value store on hybrid cloud storage with balanced performance of compaction and querying. *Proceedings of the ACM on Management of Data*, 1(4):1–27, 2023.

[34] Fenggang Wu, Ming-Hong Yang, Baoquan Zhang, and David HC Du. Ac-key: Adaptive caching for lsm-based key-value stores. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 603–615, 2020.

[35] Kan Wu, Zhihan Guo, Guanzhou Hu, Kaiwei Tu, Ramnatthan Alagappan, Rathijit Sen, Kwanghyun Park, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. The storage hierarchy is not a hierarchy: Optimizing caching on modern storage devices with orthus. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 307–323, 2021.

[36] Xingbo Wu, Yuehai Xu, Zili Shao, and Song Jiang. Lsm-trie: An lsm-tree-based ultra-large key-value store for small data items. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 71–82, 2015.

[37] Juncheng Yang, Yao Yue, and K. V. Rashmi. A large scale analysis of hundreds of in-memory cache clusters at twitter. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 191–208. USENIX Association, November 2020.

[38] Lei Yang, Hong Wu, Tieying Zhang, Xuntao Cheng, Feifei Li, Lei Zou, Yujie Wang, Rongyao Chen, Jianying Wang, and Gui Huang. Leaper: A learned prefetcher for cache invalidation in lsm-tree based storage engines. *Proceedings of the VLDB Endowment*, 13(12):1976–1989, 2020.

[39] Ting Yao, Yiwen Zhang, Jiguang Wan, Qiu Cui, Liu Tang, Hong Jiang, Changsheng Xie, and Xubin He. Matrixkv: Reducing write stalls and write amplification in lsm-tree based kv stores with matrix container in nvm. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 17–31, 2020.

[40] Huanchen Zhang, David G Andersen, Andrew Pavlo, Michael Kaminsky, Lin Ma, and Rui Shen. Reducing the storage overhead of main-memory oltp databases with hybrid indexes. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1567–1581, 2016.

[41] Huanchen Zhang, Hyeontaek Lim, Viktor Leis, David G Andersen, Michael Kaminsky, Kimberly Keeton, and Andrew Pavlo. Surf: Practical range query filtering with fast succinct tries. In *Proceedings of the 2018 International Conference on Management of Data*, pages 323–336, 2018.

[42] Jianshun Zhang, Fang Wang, and Chao Dong. Halsm: A hotspot-aware lsm-tree based key-value storage engine. In *2022 IEEE 40th International Conference on Computer Design (ICCD)*, pages 179–186. IEEE, 2022.

[43] Teng Zhang, Jian Tan, Xin Cai, Jianying Wang, Feifei Li, and Jianling Sun. Sa-lsm: optimize data layout for lsm-tree based storage using survival analysis. *Proceedings of the VLDB Endowment*, 15(10):2161–2174, 2022.

[44] Xinjing Zhou, Xiangyao Yu, Goetz Graefe, and Michael Stonebraker. Two is better than one: The case for 2-tree for skewed data sets. *memory*, 11:13, 2023.

[45] Yuanhui Zhou, Jian Zhou, Shuning Chen, Peng Xu, Peng Wu, Yanguang Wang, Xian Liu, Ling Zhan, and Jiguang Wan. Calcspar: A contract-aware lsm store for cloud storage with low latency spikes. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 451–465, 2023.

Mitigating Resource Usage Dependency in Sorting-based KV Stores on Hybrid Storage Devices via Operation Decoupling

Qingyang Zhang[1], Yongkun Li[1]*, Yubiao Pan[2], Haoting Tang[1], Yinlong Xu[1,3]

[1]*University of Science and Technology of China* [2]*Huaqiao University*
[3]*Anhui Provincial Key Laboratory of High Performance Computing*

Abstract

LSM-tree-based key-value (KV) stores mainly employ sorting-based operations (e.g., flush and compaction) to manage the KV pairs on disk. Through analysis and experiments with RocksDB, we identify that the sorting operations cause critical issues of operation coupling, including intertwined resource consumption within an operation, interdependencies and contention among operations. These coupling problems lead to dependency in resource usage and are particularly exacerbated on hybrid storage devices, causing significant resource fragmentation and increased write stalls. Existing approaches to mitigating write stalls rely on either fixed differentiated data management or superficial scheduling of data sorting operations, but they fail to fundamentally address the resource usage dependency caused by operation coupling.

In this paper, we propose DecouKV, designed to alleviate resource usage dependency and enhance resource utilization on hybrid storage devices through operation decoupling. Specifically, DecouKV decouples data sorting operations into CPU-intensive index merge tasks and I/O-intensive data append and data flush tasks by separating indexes from data files, managing indexes with a mergeable skip list-based structure and managing data with append-only files. Furthermore, we propose an elastic scheme for tuning level capacity and introduce a parameterized queue-based task scheduling strategy to maximize resource utilization. We implement DecouKV and conduct experimental evaluations. Compared to RocksDB, as well as state-of-the-art systems such as MatrixKV, PrismDB, SplitDB and ADOC, DecouKV improves CPU utilization by 25.4%–32.3%, increases throughput by 2.3–4.9×, and reduces tail latency by 74.3%–91.4% under write-intensive workloads. DecouKV also achieves a modest throughput improvement of 1.2–2.3× under read-intensive workloads.

1 Introduction

Key-value (KV) stores manage data in the form of KV pairs, enabling fast lookup, insertion, and deletion operations. As a result, KV stores are widely used in distributed systems [1,23], graph storage systems [5,18,31,51], and OLTP/OLAP databases [3, 7, 9, 15, 19, 24, 26, 36, 40, 52]. Modern KV stores typically employ the Log-Structured Merge-tree (LSM-tree) [38], a sorting-based data structure, as their core storage mechanism (e.g., LevelDB [22], RocksDB [20], Cassandra [3], X-Engine [25]). KV pairs are first cached in memory, and then persistently stored on disk through sorting-based flush and compaction operations. By converting random writes into sequential writes, LSM-tree-based KV stores are able to provide efficient write operations and good scalability.

With the continuous advancement of hardware performance, LSM-tree-based KV stores are increasingly deployed on fast devices, such as NVMe SSDs [10, 32, 45] and Persistent Memory (PM) [17, 29, 30, 47], also known as Non-Volatile Memory (NVM). However, these high-performance devices are often expensive [45], leading to the growing popularity of hybrid storage systems that combine fast devices with slower but inexpensive devices such as SATA SSDs and HDDs (e.g., [10, 29, 42, 47]). In such solutions, the common strategy is to store hot data on fast devices while storing large volumes of cold data on slower devices (e.g., [10, 29, 47]). This approach delivers high performance with minimal use of expensive fast devices, ensuring cost efficiency.

LSM-tree and its derivative optimizations [12, 17, 29, 33, 35, 47, 49] primarily adopt a sorting-based multi-level tree structure. These designs rely on heavyweight and unavoidable data sorting operations, such as flush and compaction, to manage KV pairs. However, sorting-based KV stores suffer from significant *operation coupling issues*, which manifest in three main aspects: (i) both flush and compaction operations consume a certain amount of CPU and I/O resources, leading to intertwined resource consumption; (ii) a sorting operation at a certain level can cause the size of the next level to exceed its threshold and trigger a subsequent sorting operation, resulting in interdependencies among operations; (iii) concurrent two sorting operations compete for system resources, leading to resource contention between these operations. Operation coupling issues lead to severe resource usage dependency, making it difficult to isolate CPU and I/O resources and schedule them effectively. These issues are exacerbated on hybrid storage devices due to the performance disparity between fast and slow devices, which causes fluctuating resource consumption. This results in alternating performance bottlenecks over time and increased write stalls.

Recent related works propose two main strategies for miti-

*Yongkun Li is the corresponding author.

gating write stalls. The first strategy leverages the high bandwidth of fast devices to perform differentiated data management across fast and slow devices (e.g., [29, 47]). However, data management is predefined and fixed, which remains misaligned with the fluctuating resource consumption on hybrid storage devices, leading to resource underutilization. The second strategy focuses on superficial scheduling of data sorting operations (e.g., [4, 48]), attempting to reduce latency fluctuations by adjusting the frequency, timing, and placement of sorting operations. Nevertheless, intertwined resource consumption still persists. Superficial scheduling fails to identify the resource-bound nature of sorting operations, and scheduling sorting operations of the same resource-intensive can even exacerbate resource bottlenecks.

To address these issues, we propose DecouKV, which mitigates resource usage dependency in sorting-based KV stores on hybrid storage devices via *operation decoupling*. DecouKV first decouples the indexes of KV pairs from data files, as in data sorting operations, CPU resource consumption primarily originates from index sorting, while I/O resource consumption is primarily associated with data reading and writing. This decoupling transforms data sorting operations into CPU-intensive index merge tasks and I/O-intensive data append and flush tasks by managing data with append-only files (AOFs) and indexes using a mergeable skip list-based structure called IndexTable, thereby reducing intertwined resource consumption. Additionally, DecouKV introduces two queues, the Index Merge Queue and the Data Flush Queue, to manage IndexTables. By auto-tuning queue parameters, DecouKV proactively schedules decoupled tasks. Specifically, DecouKV minimizes competition between resource-intensive tasks and optimizes resource utilization by coordinating tasks with different resource demands. Finally, DecouKV relaxes level amplification factor limit and uses elastic capacity to reduce interdependencies among data sorting operations.

We implement DecouKV and compare its performance against RocksDB, one of the most widely adopted systems in the industry, as well as two state-of-the-art systems representing the two categories of optimization strategies mentioned before: MatrixKV and ADOC. Experimental results show that under write-intensive workloads, DecouKV improves CPU utilization by 25.4%–32.3%, increases throughput by 2.3–4.9×, and reduces tail latency by 74.3%–91.4%. DecouKV also achieves a modest throughput improvement of 1.2–2.3×, under read-intensive workloads. In summary, our main contributions are summarized as follows.

- We analyze the data sorting operations of sorting-based KV stores and identify operation coupling as the root cause of resource usage dependency and resource fragmentation on hybrid storage devices.
- Based on the analysis above, we propose a decoupling solution, decoupling data sorting operations into CPU-intensive tasks and I/O-intensive tasks to reduce intertwined resource consumption within an operation.

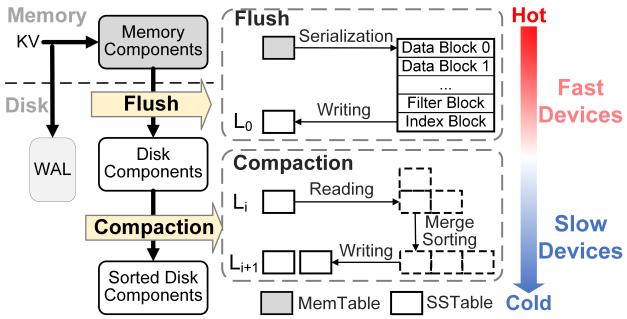


Figure 1: Architecture of LSM-tree.

- We design an elastic strategy for tuning the level capacity of LSM-trees, relaxing the level amplification factor to reduce interdependencies among data sorting operations.
- We develop an efficient task scheduling strategy to minimize competition between resource-intensive tasks and optimize resource utilization by coordinating tasks with different resource demands.

The source code of DecouKV is available online [50].

2 Background and Motivation

2.1 KV Stores on Hybrid Storage Devices

Architecture of LSM-tree-based KV stores. As shown in Figure 1, an LSM-tree consists of memory components (MemTables) and disk components organized into Sorted String Tables (SSTables) across multiple levels. Each level has a fixed storage capacity, and the amplification factor between levels is typically set to 10. The system uses background flush and compaction operations, defined here as data sorting operations, to manage data movement between components. These operations are triggered when the component capacity exceeds predefined thresholds. When the MemTable reaches its capacity limit, a flush operation serializes the data (generating metadata like Bloom filters and index blocks) and persists it to disk as SSTables at level 0. When a specific level reaches its capacity limit, the system performs compaction operations, which involve data reading, merge sorting, and data writing. These operations consume significant CPU and I/O resources, leading to resource competition and system performance degradation.

Deployment of LSM-trees on hybrid storage devices. Recently, LSM-tree-based KV stores have been extended to cost-effective hybrid storage devices, which consist of fast and slow devices. Fast devices, such as NVMe SSDs and Persistent Memory (PM), typically provide high bandwidth and low latency. Although Intel Optane DCPMM [2] has been discontinued, the trend in hardware development continues to prioritize performance improvements (like CXL SSDs [27]). However, their high cost remains a significant barrier—for example, the price of Intel Optane DCPMM per terabyte is more than 10 times higher than that of SATA SSDs [45]. Meanwhile, slower devices such as traditional SSDs and HDDs

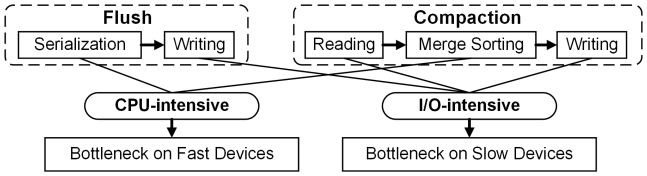


Figure 2: Intertwined resource consumption within an operation.

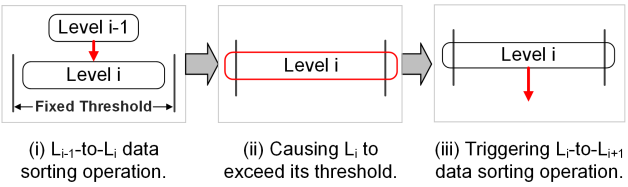
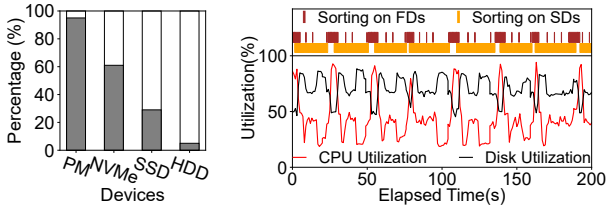


Figure 3: Interdependencies among operations.



(a) CPU Percentage. (b) Resource fluctuations.

Figure 4: Resource utilization of RocksDB on hybrid devices.

remain indispensable due to their large capacity and lower cost. Thus, hybrid storage devices become a popular solution offering a balance between performance and cost.

In hybrid devices scenarios, a common approach is to store hot data (e.g., MemTable and lower levels of the LSM-tree) on fast devices (FDs) and cold data (e.g., higher levels) on slow devices (SDs). For example, NoveLSM [29] uses PM as an extension of DRAM and deploys the MemTable on PM. MatrixKV [47] places the level 0 of the LSM-tree on PM through a matrix management approach. SpanDB [10] treats NVMe SSD as a performance device, deploying the low levels of the LSM-tree on it. PrismDB [42] identifies hot and cold data, storing hot data on PM and cold data on slow devices. Overall, by strategically balancing the use of fast and slow devices, these systems achieve high throughput while reducing storage costs.

2.2 Problems of Operation Coupling

Typically, both RocksDB and its derivative optimizations adopt a multi-level, sorting-based design. As mentioned earlier, these designs rely on heavyweight and unavoidable data sorting operations, such as flush and compaction, to manage KV pairs. Below, we first analyze three categories of operation coupling issues. Then, we discuss how these issues exacerbate the underutilization of CPU and I/O resources in sorting-based KV stores deployed on hybrid storage devices, which ultimately results in resource fragmentation and increased write stalls.

2.2.1 Operation Coupling Issues

Specifically, the operation coupling issues in LSM-tree manifest in three key aspects:

Intertwined resource consumption within an operation. Data sorting operations require substantial CPU cycles and I/O bandwidth simultaneously. As shown in Figure 2, the serialization (generating metadata like Bloom filters and index blocks) in the flush operation consumes substantial CPU cycles, categorizing it as a CPU-intensive task, whereas data writing primarily consumes I/O bandwidth, classifying it as an I/O-intensive task. Similarly, in the compaction operation, merge sorting is CPU-intensive, while data reading and writing are I/O-intensive. Within a single data sorting operation, the intertwined nature of CPU-intensive and I/O-intensive tasks leads to intertwined resource consumption.

Interdependencies among operations. In sorting-based KV stores, data sorting operations are strictly triggered by fixed thresholds, creating dependencies between adjacent levels. As shown in Figure 3, an L_{i-1}-to-L_i data sorting operation may increase L_i's size, causing it to exceed its threshold and trigger an L_i-to-L_{i+1} data sorting operation. Such interdependencies mean that operations are passively triggered, leaving the system unable to choose the level and timing of data sorting based on resource availability. This results in resource conflicts and waste.

Resource contention between operations. Under high write loads, data sorting operations often occur simultaneously across multiple components of the sorting-based KV stores, leading to competition for system resources. In resource-constrained systems, resource contention reduces the efficiency of sorting operations, causing severe write stalls.

2.2.2 Exacerbation on Hybrid Storage Devices

Furthermore, the coupling problem becomes further exacerbated on hybrid storage devices. Due to the performance disparity between fast and slow devices, their capabilities for handling I/O requests differ significantly, leading to varying performance bottlenecks for data sorting operations on different devices. We validate the analysis through experiments with RocksDB deployed on hybrid storage devices.

Varying bottlenecks on different devices. First, we measure the percentage of CPU time consumed by data sorting operations on different devices (refer to Section 4 for detailed configurations). Figure 4(a) shows that on fast devices (PM, NVMe SSD), CPU time accounts for over 60% of compaction time, making that the CPU becomes the performance bottleneck for these devices. On slow devices (SATA SSD, HDD), the CPU accounts for less than 30% of the compaction time, with a large proportion of time spent on I/O wait. As a result, I/O bandwidth remains the performance bottleneck for slow devices.

Alternating bottlenecks on hybrid storage devices. Next, we test RocksDB's performance on hybrid storage devices, following the common approach introduced in Section 2.1, where levels 0 and 1 are stored on fast devices (PM) and other levels on slow devices (SATA SSD). We import 100M update requests into a 100GB database, and record sorting times on fast and slow devices, as well as CPU and disk utilization over time. We present a representative 200-second subset, with similar patterns observed throughout the experiment. Figure 4(b) shows that under write-intensive workloads, resource utilization fluctuates significantly, with low averages and severe resource fragmentation. Disk and CPU utilization exhibit opposing patterns: during sorting on fast devices, CPU utilization is high (94%) but disk utilization is low (48%), indicating a CPU bottleneck; during sorting on slow devices, disk utilization is high (88%) while CPU utilization is low (21%), indicating an I/O bottleneck. These bottlenecks alternate between data sorting operations on two devices, hindering the full utilization of CPU and I/O resources and ultimately increasing write stalls.

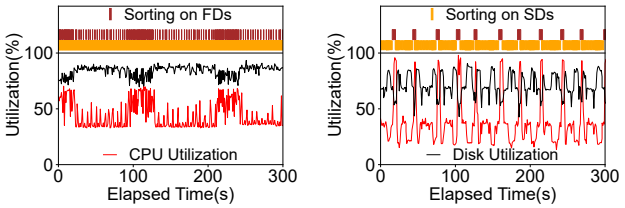
Overall, data sorting operations consume intertwined CPU and I/O resources, and the disparity I/O bandwidth between devices leads to different resource bottlenecks on different devices. Additionally, the fluctuating resource consumption is misaligned with the fixed hardware capabilities. Due to interdependencies among operations, the system cannot proactively trigger operations to utilize resource fragments efficiently (e.g., triggering sorting on fast devices when the CPU is idle).

2.3 Limitations of Existing Solutions

There are two main strategies from recent related works for mitigating write stalls: (i) fixed differentiated data management and (ii) superficial scheduling of data sorting operations.

Fixed differentiated data management. This category of work [10, 29, 42, 47] argues that the performance disparity between fast and slow devices necessitates differentiated management strategies, rather than applying the same approach to both. These works typically modify data sorting operations on specific devices, mitigating intertwined resource consumption to some extent. However, severe interdependencies still exist among the data sorting operations. Data management is predefined and fixed, which remains misaligned with the fluctuating resource consumption on hybrid storage devices, leading to resource underutilization.

Taking MatrixKV [47] as an example, which places L_0 of the LSM-tree on persistent memory and employs a matrix container for differentiated management. As shown in Figure 5(a), the amplitude of resource utilization fluctuation is reduced compared to RocksDB (though still exceeding 50%). The result indicates that fine-grained compaction reduces the CPU and I/O resources consumed in each operation, mitigating intertwined resource consumption to some extent. However, the issue of interdependencies among operations



(a) Differentiated data management. (b) Superficial operations scheduling.

Figure 5: Resource utilization fluctuations of existing solutions.

persists. Frequent sorting on fast devices keeps CPU utilization at a high level (100–120s in Figure 5(a)). When the data on fast devices exceeds the threshold, a large number of threads shift to sorting on slow devices, causing CPU utilization to drop significantly due to I/O bottlenecks. As a result, the fluctuating resource consumption on hybrid storage causes the average resource utilization to remain low.

Superficial scheduling of data sorting operations. Some works [4, 13, 48] aim to improve performance by superficial scheduling data sorting operations. After detecting the system state, they superficially adjust the frequency, timing, and location of these operations to reduce latency fluctuations, alleviating interdependencies among operations to some extent. However, intertwined resource consumption within operation still persists. These works fails to identify the resource-bound nature of sorting operations, as well as the varying performance bottlenecks on hybrid storage devices. Moreover, scheduling sorting operations of the same resource-intensive can even exacerbate resource bottlenecks.

Taking ADOC [48] as an example, which adjusts parameters to control the components and rates at which data sorting operations occur to reduce data blockage. The experiment on hybrid storage devices (Figure 5(b)) shows more pronounced fluctuations in its resource utilization. Compared to RocksDB in Figure 4(b), sorting on fast devices causes a significantly higher CPU utilization, approaching 100%. At the same time, disk utilization decreases significantly, indicating a more severe CPU bottleneck. This occurs because, when data blockage occurs on components in the fast devices, ADOC schedules more sorting threads for sorting on fast devices. Meanwhile, sorting threads on slow devices are reduced. Resource contention between these operations leads to a more severe CPU bottleneck, while the bandwidth of slow devices is underutilized. Conversely, when data blockage occurs on slow devices, ADOC achieves higher disk utilization but lower CPU utilization, indicating a more severe I/O bottleneck. As a result, the bottlenecks significantly alternate due to the intertwined resource consumption within operation.

Although these efforts have made progress in alleviating the write stall issue in LSM-trees, they cannot fundamentally solve the problem of resource fragmentation and underutilization on hybrid storage devices caused by operation coupling. To address this issue comprehensively, decoupling the operations within the sorting-based KV stores is necessary.

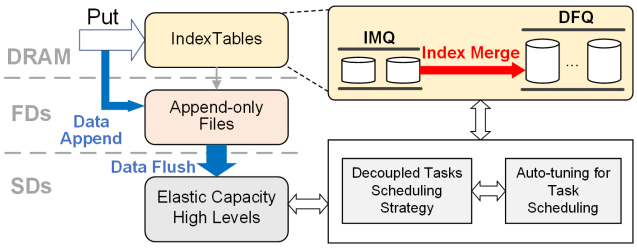


Figure 6: DecouKV overview.

3 DecouKV Design

3.1 Design Goals and Challenges

DecouKV focuses on decoupling data sorting operations into resource-independent CPU-intensive tasks and I/O-intensive tasks, enabling the system to flexibly schedule tasks based on current resource utilization. The design goals and challenges of DecouKV are as follows:

Decoupling operations. DecouKV needs to decouple operations to achieve relative isolation while maintaining correctness and consistency, thereby facilitating subsequent task scheduling and resource utilization. Specifically, DecouKV needs to address the three operation coupling issues discussed in Section 2.2.1.

Managing decoupled components. Decoupled operations need to achieve the original objectives of coupled operations. Coupled sorting operations improve data orderliness, addressing two key issues: unordered indexes degrade query performance, and unordered data on disk leads to random access patterns. Decoupled operations must similarly resolve these two issues.

Efficient scheduling of decoupled tasks. DecouKV needs to efficiently schedule decoupled tasks to maximize CPU and I/O resource utilization. This requires accurately assessing resource states and making appropriate scheduling decisions to improve resource efficiency.

3.2 System Overview

This section introduces DecouKV, designed to mitigate resource usage dependency in sorting-based KV stores deployed on hybrid devices via operation decoupling. Now, we begin by outlining the architecture of DecouKV.

In sorting-based KV stores, data sorting operations involve CPU-intensive sorting tasks mainly for indexes and I/O-intensive read/write tasks mainly for data. Therefore, to isolate these tasks, DecouKV first decouples indexes from data files. Next, suitable structures are required to manage the separated indexes and data while maintaining their isolation. The structure managing indexes must satisfy the following requirements: (i) fast insertion and query speeds, as these are fundamental for indexes, and (ii) support for independent sorting tasks without impacting access during merging. The structure managing data must meet these criteria: (i) fast insertion speed, and (ii) an unordered design to avoid excessive CPU consumption for data sorting. As shown in Figure 6, DecouKV manages data on fast devices using append-only files (AOFs) and manages their indexes in DRAM using IndexTables, a mergeable structure based on skip list.

In this way, DecouKV decouples data sorting operations into three types of tasks: CPU-intensive *index merge* tasks and I/O-intensive *data append* and *data flush* tasks, reducing intertwined resource consumption. Specifically, index merge tasks focus solely on sorting indexes in DRAM without involving data read/write operations, making them CPU-intensive. Data append and flush tasks primarily consume disk I/O without sorting, making them I/O-intensive.

Additionally, DecouKV employs two DRAM-based queues—Index Merge Queue (IMQ) and Data Flush Queue (DFQ)—to manage IndexTables. By autotuning queue parameters, DecouKV actively schedules the decoupled tasks. Specifically, DecouKV minimizes competition among the same resource-intensive tasks and optimizes resource usage by coordinating tasks with different resource demands. Finally, DecouKV introduces elastic capacity design for high levels (HLs) of the LSM-tree on slow devices, relaxing the strict level amplification factor to reduce interdependencies among operations. The subsequent sections detail the design of these decoupling mechanisms.

3.3 Decoupled Components

DecouKV manages indexes with DRAM-based IndexTables and data with AOFs. Below, we describe their structures.

IndexTable. DecouKV employs IndexTables, a mergeable skip list-based structure, to store indexes in DRAM (Figure 6). Each IndexTable uses a skip list to manage indexes, enabling efficient lookups and subsequent index merge tasks. DecouKV leverages the index to precisely locate data in fast devices, with each index entry containing a key and the corresponding data address in fast devices (comprising an 8-byte file number and an 8-byte offset). Unlike RocksDB's MemTable, IndexTables avoid storing values, supporting more entries within the same memory footprint. To ensure efficient index insertion and memory utilization, the initial capacity of the IndexTable, denoted as *max_index_size*, is set to a relatively small value (8 MB). Note that to minimize the impact of I/O operations on the indexes after decoupling, DecouKV employs asynchronous insertion. By recording the data address, DecouKV allows insertion to proceed without waiting for the data append to fast devices to complete. Once the IndexTable reaches *max_index_size*, it becomes non-insertable, and a new IndexTable is created. Queries traverse IndexTables sequentially, from newest to oldest, until the required key is located. When the number of IndexTables exceeds a threshold, DecouKV triggers an index merge operation, which will be discussed in the next subsection.

Append-only file (AOF). DecouKV stores KV pairs as AOFs on fast devices, with the corresponding indexes in DRAM-based IndexTables. Although AOFs are unordered files prone

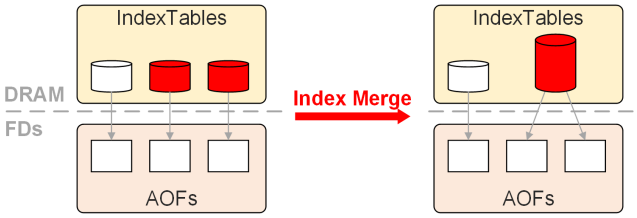


Figure 7: Index merge. The red cylinders represent the IndexTables before and after merging, and the arrows indicate the correspondence between the IndexTables and AOFs.

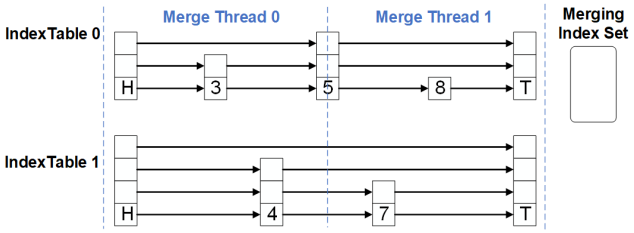
to random access, DecouKV leverages the high bandwidth and concurrency of fast devices to achieve random access performance comparable to sequential access. Consequently, AOFs maintain unordered data without compromising performance. Furthermore, the original Write-Ahead Logs (WALs) of the LSM-tree are eliminated as DecouKV can recover data with AOFs during system failures. Additionally, AOFs also eliminate the extra CPU overhead associated with generating Bloom filters for the data on fast devices, and reduce the memory overhead required for caching these Bloom filters during query processing. Because of the excellent random access performance of fast storage devices, the impact on read performance caused by removing the Bloom Filter in AOF is effectively mitigated.

3.4 Decoupled Tasks

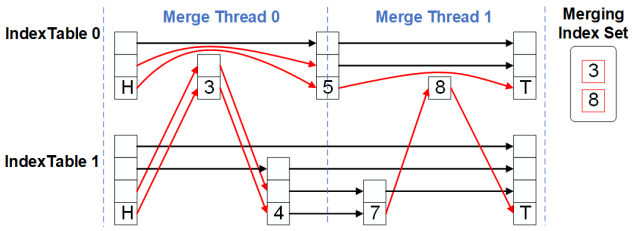
Based on IndexTables and AOFs, DecouKV decouples data sorting operations into CPU-intensive index merge tasks and I/O-intensive data append and data flush tasks. In the following, we explain how to handle these decoupled tasks to achieve the objectives of the original coupled tasks outlined in Section 3.1.

Index merge. Obviously, the number of IndexTables directly impacts query performance. To address the first issue mentioned in Section 3.1—unordered indexes degrade query performance—index merge tasks reduce the number of unordered IndexTables. As illustrated in Figure 7, DecouKV employs index merge tasks to consolidate two red IndexTables into a larger one. Note that index merge tasks occur entirely in DRAM without affecting the data in AOFs. Therefore, index merge is a CPU-intensive task. Initially, each IndexTable corresponds to one AOF; after merging, one IndexTable may correspond to multiple AOFs.

Index merge effectively merges two skip lists. Recent studies have introduced many optimizations for merging skip list structures [17,30]. Specifically, by altering the pointers within skip lists, the two skip lists can be merged efficiently without blocking query requests. DecouKV further implements multithreaded parallelism based on these optimizations, making full use of CPU resources. As shown in Figure 8, DecouKV uses multiple threads to perform parallel index merge. In Figure 8(b), the red arrows are pointers that are currently being modified. Meanwhile, DecouKV maintains a Merging Index



(a) Initial state.



(b) Multithreaded parallel merging.

Figure 8: Illustration of multithreaded parallel index merge. The figure shows two threads performing parallel merging. The arrows represent skip list pointers.

Set (a lock-free set implemented using CAS) to store atomic pointers to the index nodes being merged (3 and 8 in the figure). In this way, after accessing IndexTable 0, the query operation will first check the Merging Index Set and then access IndexTable 1 to ensure the correctness of the query. Once the node insertion is complete, the thread removes the corresponding pointer from the Merging Index Set.

Data append and data flush. DecouKV employs the data append task, an I/O-intensive operation, to write KV pairs into the AOFs on fast devices. Additionally, DecouKV performs the data flush task to flush KV pairs into slow devices. Notably, to avoid the performance gap between random and sequential access on slow devices, DecouKV organizes KV pairs in an ordered data structure (SSTables) on slow devices. When the size of a merged IndexTable reaches the threshold *data_flush_trigger_size*, DecouKV triggers a data flush operation to consolidate the IndexTable and its corresponding AOFs into the slow devices. Specifically, the data flush task sequentially scans all indexes in the IndexTable, retrieves the data from the AOFs and writes them into the SSTables on slow devices. Finally, the system deletes the outdated IndexTable and its corresponding AOFs. Note that while the data flush task inevitably introduces some CPU overhead, I/O remains the primary performance bottleneck on slow devices. Furthermore, since the indexes in the IndexTable are already organized as sorted skip lists, the data flush task is still predominantly classified as an I/O-intensive operation.

3.5 Scheduling Strategy of Decoupled Tasks

In this section, we discuss the scheduling of decoupled CPU-intensive and I/O-intensive tasks to maximize system resource utilization. DecouKV employs two distinct queues to evaluate

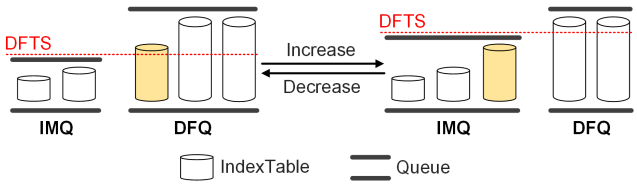


Figure 9: Auto-tuning DFTS. The height of the cylinder represents the size of the IndexTable. The yellow cylinder is the IndexTable that has been moved during auto-tuning DFTS.

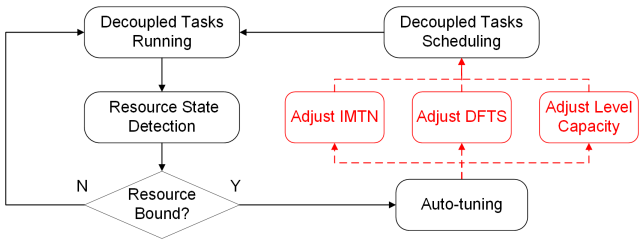


Figure 10: Control flow of task scheduling.

system resource status and uses corresponding parameters to manage task scheduling effectively.

Index Merge Queue (IMQ). DecouKV enqueues small non-insertable IndexTables into the IMQ, awaiting index merge. Since index merge is a CPU-intensive task, the length of the IMQ (L_{IMQ}) reflects the availability of CPU resources. When CPU resources become bottlenecked, the number of IndexTables waiting for merge increases, causing the IMQ to grow. The threshold for triggering an index merge task is referred to as *index_merge_trigger_num* (IMTN). We define $Score_{IM} = \frac{L_{IMQ}}{IMTN}$ to represent CPU resource pressure. When $Score_{IM}$ exceeds a certain value (1.5 is optimal in our experiments), DecouKV identifies the system as CPU-bound.

DecouKV can adjust CPU resource demand by modifying the IMTN parameter. By increasing IMTN, the IMQ can accommodate more IndexTables, thereby reducing the frequency of index merge triggers and alleviating the pressure on CPU resources. Conversely, decreasing IMTN increases the demand for CPU resources.

Data Flush Queue (DFQ). For IndexTables that reach the threshold size of *data_flush_trigger_size* (DFTS), DecouKV places them into the DFQ awaiting data flush. Similarly, the length of the DFQ (L_{DFQ}) reflects the pressure of I/O resources. Since data flush is an I/O-intensive task, when I/O resources become bottlenecked, the DFQ length increases. We define $Score_{DF} = L_{DFQ}$ to represent I/O resource pressure. When $Score_{DF}$ exceeds a certain value (3 is optimal in our experiments), DecouKV identifies the system as I/O-bound.

DecouKV adjusts the DFTS to schedule different resource-intensive tasks. As shown in Figure 9, increasing the DFTS causes some IndexTables to move from the DFQ back to the IMQ, thereby increasing CPU demand and alleviating I/O pressure. Conversely, decreasing the DFTS alleviates CPU pressure and increases I/O demand.

3.6 Elastic Capacity High Levels

For the high levels (HLs) stored on slow devices, DecouKV still uses leveled compaction. Due to memory limitations and the random access performance of slow devices, we cannot leverage the same decoupling design as lower levels. Therefore, we chose to use elastic capacity to mitigate the coupling issue at high levels by relaxing the strict amplification ratio between levels. In Section 2.2, we mentioned three key aspects of the operation coupling issue, with the second point discussing how interdependencies among operations stem from fixed thresholds. Elastic capacity is mainly optimized to address this problem. For example, a large number of HL_0 compactions may lead to an IO-bound situation, which also causes HL_1 to exceed its threshold and trigger further compaction, leading to an even worse IO-bound situation. In this case, DecouKV relaxes the capacity threshold for HL_1, postpones the sorting for that part, and schedules CPU-intensive index merges to improve CPU utilization.

However, relaxing the amplification ratio between levels in other KV stores may temporarily lead to increased size in certain levels, potentially introducing some challenges. (i) Increased write amplification: When the size of L_i increases while the size of L_{i-1} remains relatively smaller, the compaction between L_{i-1} and L_i will involve more data from L_i, causing additional write amplification. (ii) Higher read overhead: With the larger size of L_i, the cost of read operations accessing L_i will increase.

Fortunately, the decoupling design in DecouKV, as described in Section 3.3, successfully addresses both challenges. For the first one, the DFTS parameter (which directly influences the file size written to HL_0) is adjustable. This allows DecouKV to control the size of files written to HL_0. By adjusting the capacity of the high levels according to DFTS, DecouKV ensures that the size ratio between adjacent levels does not become excessively large, thereby controlling write amplification. Additionally, the use of the index merge operation reduces some of the write amplification caused by duplicate keys. For the second one, DecouKV stores hotter data in fast devices, while cold data in the high levels is accessed less frequently. Moreover, data written to higher levels is pre-sorted, resulting in larger sizes for high levels but fewer overall levels. Subsequent experiments demonstrate that even under read-intensive workloads, DecouKV achieves a slight improvement in throughput.

3.7 Auto-tuning for Task Scheduling

As shown in Figure 10, first, DecouKV monitors the IMQ and DFQ to assess the system resource state. It then determines whether the system is in a resource-constrained state based on the scores defined in Section 3.5. Following this, DecouKV performs auto-tuning according to the actual system state. By adjusting IMTN, DFTS, and the level capacity, DecouKV schedules resource-intensive tasks accordingly. This ensures

efficient utilization of system resources and reduces fragmentation. Detailed parameter settings and tuning methods are presented below.

The parameter setting in DecouKV is based on the default values of similar parameters in RocksDB. As a widely adopted industrial storage engine, the parameters in RocksDB are fine-tuned through numerous experimental optimizations. We also validate the efficiency of the selected values via experiments. Specifically, the default value of IMTN in our evaluation is 2. Since IMTN directly impacts read performance, we set it according to the default number of MemTables, which similarly affects read performance. The default value of DFTS is 32MB (4 * 8MB), which is approximately the size of 4 merged IndexTables according to the number of files in L_0 (default 4). When the system is CPU-bound, DecouKV increases the IMTN to reduce CPU resource demand. Simultaneously, DecouKV decreases the DFTS to minimize the triggering of CPU-intensive index merge operations while scheduling more I/O-intensive data flush tasks. Conversely, when the system is I/O-bound, DecouKV reduces the IMTN and increases the DFTS to lower I/O resource demand, thereby scheduling more CPU-intensive tasks. Additionally, proportionally adjusting the capacity of the high levels based on DFTS further regulates I/O resource pressure on slow devices. Note that DecouKV adjusts IMTN by increasing 2 (or decreasing 2) and adjust DFTS by multiplying (or dividing) by 2 according to the state, since the number of IndexTables typically increases incrementally, while the data size often increases exponentially (e.g., two IndexTables of 16MB each are merged into a 32MB IndexTable).

Naturally, these parameters are capped and cannot grow indefinitely. If both CPU and I/O are bound (indicated by congestion in both IMQ and DFQ), DecouKV recognizes that the incoming data writes exceeds the system's processing capacity and slows down data write through write stalls. Finally, when neither queue experiences congestion over a certain period (set to 60 seconds in our implementation), DecouKV identifies that system resources are idle. In this case, it reduces the IMTN, DFTS, and the capacity of the high levels simultaneously to schedule both CPU- and I/O-intensive tasks effectively. In this way, DecouKV improves data orderliness to enhance read performance.

4 Evaluation

In this section, we evaluate and compare DecouKV with RocksDB v9.3.0 and the two categories of solutions mentioned in 2.3 using hybrid storage devices. Among the works following the fixed differentiated management approach, MatrixKV stands out as a typical representative. In the superficial scheduling direction, ADOC represents the state-of-the-art. Therefore, we select RocksDB, MatrixKV and ADOC as the basic benchmarks for comparison. Besides, some state-of-the-art works on hybrid storage, such as PrismDB [42] and SplitDB [6], mainly focus on different scenarios like workload skewness or read-intensive workloads. To ensure a comprehensive evaluation, we also include them in our comparative analysis under YCSB workloads. The test data are obtained from three experimental runs, with minimal variation in the results. The average values are presented.

Hardware environment. The experiments are conducted on a server equipped with two 20-core Intel Xeon Gold 5218R processors and 128 GB of DRAM. The server runs Ubuntu 20.04.6 LTS with the Linux kernel 5.15. For storage devices, we use 128 GB Intel Optane DCPMM and INTEL SSDPE2KE032T8 NVMe SSDs for fast devices, and 960 GB Intel S4520 SSDs with SATA 3.0 interfaces for slow devices.

Configuration of key-value stores. For all KV stores, we use the recommended configuration [21], setting the Memtable size and SSTable size to 64MB, configuring Bloom filters to 10 bits per key, and allocating a block cache size of 1GB. By default, four CPU cores are assigned, with four background compaction threads, following the optimization tuning guide. We also use direct I/O to avoid the impact of the operating system's page cache. For fairness, we limit the amount of fast devices allocated to each KV store to 10% of the dataset size (e.g., 10 GB for a 100 GB dataset). Specifically, for DecouKV, we use the configuration described in Sec 3, deploying AOFs on fast devices while controlling space usage. The initial size of the IndexTable is set to 16MB. For RocksDB and ADOC, we place L_0, L_1, and the WAL on fast devices, slightly increasing the sizes of L_0 and L_1 to ensure an equal allocation of fast devices. For MatrixKV, PrismDB and SplitDB, we follow the configuration from their paper, placing their differentiated data structures on fast devices and allocating an equivalent amount of space.

First, we evaluate the resource utilization, throughput, average latency, and tail latency of various fundamental KV operations using the following workloads: (i) inserting 100 GB KV pairs, (ii) updating 100 GB KV pairs, (iii) reading 100 GB KV pairs, and (iv) scanning 10 GB KV pairs with a maximum scan length of 100. Prior to each experiment, we clear the entire database to avoid interference. By default, we perform the microbenchmark evaluations using a uniform workload with a KV size of 1KB (24B for the key and 1000B for the value). We also conduct the same operations under a zipfian distribution and observe similar results, so we omit the results in the interest of space.

4.1 Microbenchmark

Resource utilization. We evaluate the variation in CPU and disk utilization using psutil [44] during insert and update operations over a 1500-second period. We calculate the average CPU utilization and fluctuation range during this time, as shown in Figure 11. For the insert operation, compared to RocksDB, MatrixKV, and ADOC, DecouKV achieves a 32.3%, 25.4%, and 27.3% increase in average CPU utilization, respectively, while significantly reducing the fluctuation range. This improvement is a key factor contributing to the

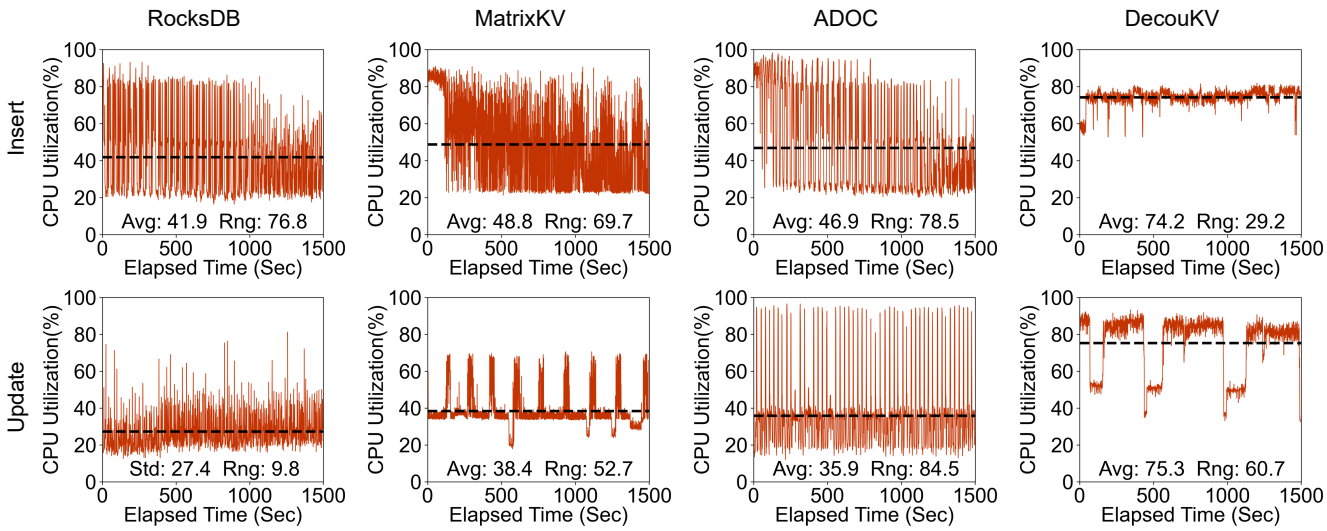


Figure 11: The red solid line represents the real-time CPU utilization over a 1500-second period, measured at a granularity of one second. The black dashed line indicates the average CPU utilization over this period. In the figure, "Avg" denotes the average value, and "Rng" represents the range, reflecting the extent of fluctuation in utilization.

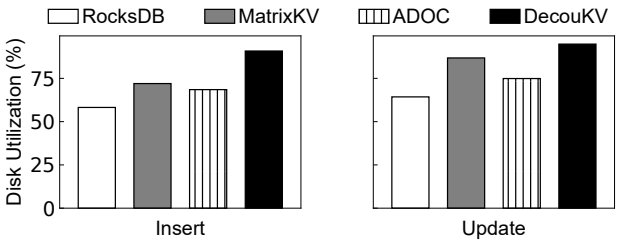
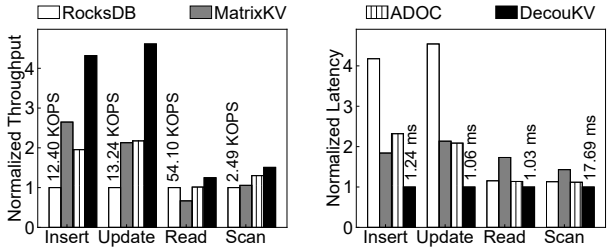


Figure 12: Disk utilization.



(a) Throughput. (b) Average latency.

Figure 13: Performance under different operations.

significant write performance gains achieved by DecouKV. For the update operation, compared to RocksDB, MatrixKV, and ADOC, DecouKV improves average CPU utilization by 47.9%, 36.9%, and 39.4%, respectively. Due to the capacity limitations of fast devices, DecouKV experiences resource fluctuations. However, upon examining the fluctuation intervals, we observe that DecouKV maintains relatively stable resource utilization over extended periods. Figure 12 presents the average disk utilization over the same period. Compared to RocksDB, MatrixKV, and ADOC, DecouKV achieves a 17.6%–31.6% increase in disk utilization for the insert operation and an 8.0%–19.9% increase in disk utilization for the update operation. Additionally, we observe that MatrixKV exhibits high disk utilization, indicating that fine-grained compaction can mitigate intertwined resource consumption and improve disk utilization. However, due to interdependencies among operations, it still suffers from significant resource fluctuations. ADOC reduces blocking by scheduling data sorting operations, improving average resource utilization compared to RocksDB. Nonetheless, on hybrid storage devices, it experiences even more severe resource fluctuations due to intertwined resource consumption. In summary, DecouKV improves average resource utilization and reduces fragmentation by mitigating resource usage dependency.

Throughput. Figure 13(a) presents the throughput results for various operations. To facilitate comparison, we normalize the experimental results using the throughput of RocksDB as the reference and annotate the absolute values in the figure. Compared to RocksDB, DecouKV achieves 4.3× the throughput for inserts and 4.6× the throughput for updates. Additionally, DecouKV achieves 1.2× the throughput for reads and 1.5× the throughput for scans. Under read-intensive workloads, the background threads remain idle and DecouKV is adapted to allow a very limited number of overlapping IndexTables and files, resulting in a higher degree of data order and fewer levels in the LSM-tree (e.g., HL_0 is empty in most cases). Meanwhile, benefiting from the excellent random access performance of fast storage devices, the impact of removing the Bloom Filter in AOF is effectively mitigated. As a result, DecouKV's read performance is not significantly affected and even shows a slight improvement.

Compared to MatrixKV and ADOC, DecouKV also demonstrates a significant advantage under write-intensive workloads. Specifically, DecouKV achieves 1.6× and 2.2× the

Latency (ms)	Insert		Update	
	90%	99%	90%	99%
RocksDB	9.454	37.557	11.619	16.132
MatrixKV	5.571	12.532	10.010	22.219
ADOC	6.995	17.881	7.545	21.075
DecouKV	**1.579**	**3.227**	**2.275**	**14.480**

Table 1: Tail Latency.

Workload	Characteristics
A (Update Heavy)	50%updates, 50%reads
B (Read Mostly)	5%updates, 95%reads
C (Read Only)	100%reads
D (Read Latest)	5%inserts, 95%reads
E (Scan Mostly)	5%inserts, 95%scans
F (Read-Modify-Write)	50%read-modify-write, 50%reads

Table 2: YCSB core workloads.

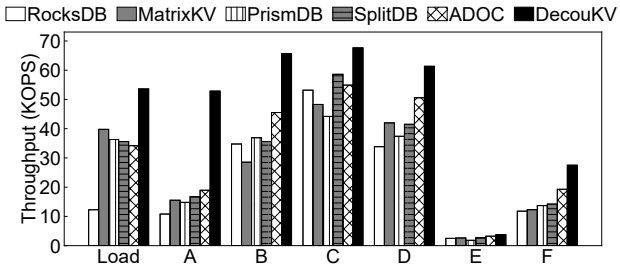
throughput of MatrixKV and ADOC, respectively, for insert operations, and 2.2× and 2.1× the throughput for update operations. Additionally, DecouKV shows good performance in both read and scan operations.

Average latency. Figure 13(b) presents the experimental results normalized to DecouKV's average latency, with DecouKV's latency values annotated in the figure. Compared to RocksDB, DecouKV reduces the average latencies for inserts and updates by 76.0% and 77.9%, respectively. In comparison to MatrixKV and ADOC, DecouKV achieves a reduction in average latencies for inserts and updates ranging from 38.7%–56.9%. Additionally, DecouKV achieves average latencies for reads and scans comparable to those of other KV stores.

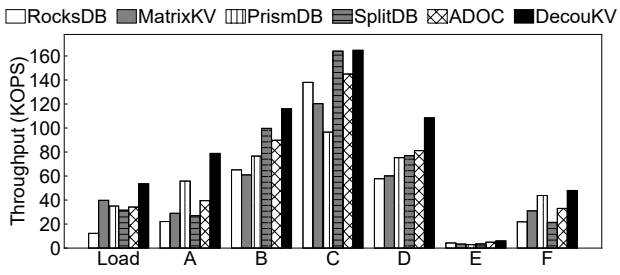
Tail latency. Table 1 presents the P90 and P99 tail latency results for insert and update operations. We observe that compared to RocksDB, MatrixKV, and ADOC, DecouKV reduces the P90 tail latency of the insert operation by 71.6%–83.3% and the P99 tail latency by 74.3%–91.4%. For the update operation, DecouKV reduces the P90 tail latency by 69.8%–80.4%. These improvements are attributed to the stable resource utilization and the reduction in resource bottlenecks in DecouKV. However, DecouKV does not show a significant advantage in P99 tail latency, which corroborates the fluctuations in resource utilization shown in Figure 11.

4.2 YCSB Benchmark

In this section, we evaluate DecouKV using the YCSB-C [39, 43], a C++ version of YCSB [11] benchmark. YCSB (Yahoo! Cloud Serving Benchmark) is a widely used testing tool that generates workloads based on real-world data characteristics. We conduct our evaluation using the six workload configurations shown in Table 2. Each workload runs 100 million operations on a randomly loaded 100GB database, with a KV size of 1KB (24B for the key and 1000B for the value). All other settings remain consistent with those used in



(a) Uniform workloads.



(b) Zipfian workloads.

Figure 14: Throughput under YCSB workloads.

previous experiments. Figure 14(b) and Figure 14(a) present the throughput of RocksDB, MatrixKV, PrismDB, SplitDB, ADOC, and DecouKV under the YCSB uniform and zipfian workloads.

Write-intensive workloads (load, YCSB-A, and -F). For write-intensive workloads, DecouKV demonstrates significant throughput improvement. Compared to RocksDB, DecouKV achieves a 2.3–4.9× improvement. In comparison to MatrixKV and ADOC, DecouKV shows a smaller improvement under the load workload, with a 1.4× and 1.6× increase, respectively. This indicates that MatrixKV and ADOC, as optimized solutions for addressing write stall issues, effectively reduce the occurrence of write stalls under pure write workloads, leading to superior performance. However, for read-write mixed write-intensive workloads like YCSB-A and YCSB-F, the resource bottlenecks caused by frequent writes also impact read performance. DecouKV successfully mitigates resource fragmentation, reducing its impact on both read and write performance, thus providing an advantage with a 1.5–3.4× improvement. Compared to PrismDB and SplitDB, DecouKV demonstrates significant advantages under uniform workloads, achieving improvements of 1.5–3.6×. Under heavily skewed Zipfian workloads, PrismDB performs particularly well in write-intensive cases (YCSB-A, YCSB-F). This is because PrismDB employs a slab-based structure and in-place update strategy on the fast devices. When the workload is highly skewed, a large portion of write requests are handled on the fast devices and are not pushed to the slow device, thus achieving high throughput. However, DecouKV still achieves strong performance overall.

Additionally, according to our previous analysis, the sys-

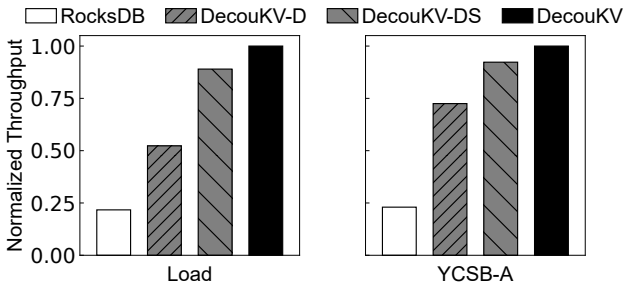


Figure 15: Normalized throughput of different DecouKV versions.

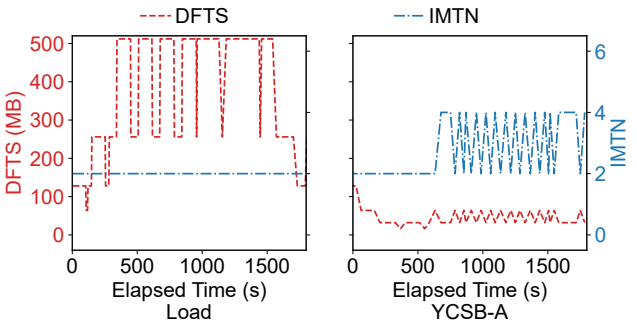


Figure 16: Tuning actions for load and YCSB-A workloads.

tem often enters a resource-constrained state under write-intensive workloads. In such cases, DecouKV relaxes certain parameters, resulting in the creation of multiple overlapping files (e.g., IndexTables or SSTables in HL_0), which could impact read performance. However, in our experiments, we observe that DecouKV's read performance is not significantly affected. This is because, under write-intensive workloads, with the scheduling of decoupled tasks in DecouKV, the burst in CPU usage is short-lived, thereby preventing excessive IndexTables (in the evaluation, the number of IndexTables never exceeds 6, which is even lower than the number of files in L_0 in RocksDB, even during write stalls). As shown in Figure 16, the maximum value of IMTN is only 4, further validating this result. Regarding overlapping SSTables in HL_0, we set the number of files to one in order to trigger the compaction of data from HL_0 to HL_1, giving HL_0 compaction a higher priority. In the evaluation, even under write-intensive workloads, the number of files in HL_0 does not exceed two. Under read-intensive workloads, HL_0 is empty in most cases.

Read-intensive and scan-intensive workloads (YCSB-B, -C -D, and -E). For read-intensive and scan-intensive workloads, DecouKV also demonstrates modest improvements. Compared to RocksDB, MatrixKV, and ADOC, DecouKV achieves a 1.2–1.9× improvement under the zipfian workload and a 1.2–2.3× improvement under the uniform workload. These gains primarily stem from more efficient resource utilization and a reduction in the number of high levels on slow devices. Compared to PrismDB and SplitDB, DecouKV also exhibits slight advantages under uniform workloads, achieving a 1.1–1.8× improvement. For highly skewed zipfian workloads, DecouKV shows no advantage, whereas SplitDB performs remarkably well. SplitDB employs a cascading skiplist index–which constructs fast paths between adjacent LSM-tree levels to reduce lookup latency–and a data promotion and demotion mechanism that migrates hot data from slower SSDs to faster NVM and cold data to SSDs, thereby optimizing read performance under skewed workloads.

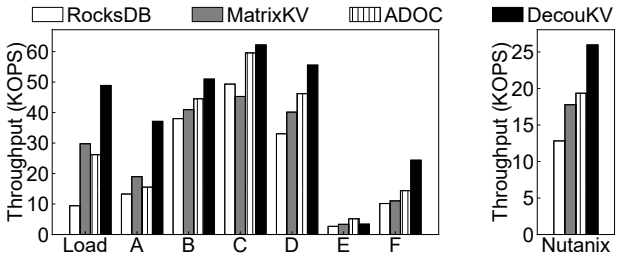
4.3 Performance Breakdown

In our design, we consider three primary techniques to optimize system performance: decoupling, scheduling strategies, and elastic capacity for high levels. In this section, we evaluate the performance contribution of each technique to DecouKV. However, task scheduling relies on decoupling, while elastic capacity high levels depends on scheduling strategies. Consequently, we evaluate three versions of DecouKV against RocksDB: DecouKV-D (decoupling only, with the queue parameters set to the average of the auto-tuning results from the experiment), DecouKV-DS (decoupling + scheduling strategies), and DecouKV (decoupling + scheduling strategies + elastic capacity high levels, i.e., the complete version of DecouKV). Figure 15 shows the throughput of different DecouKV versions during insert and update operations, normalized to DecouKV as the baseline. In the following, we will explain the contribution of each component.

Decoupling technique. By comparing the performance of DecouKV-D with RocksDB, we can roughly estimate the contribution of the decoupling technique. The decoupling technique contributes significantly to both the load and YCSB-A workload, with a greater contribution observed under the YCSB-A workload. This is because DecouKV's decoupling technique primarily targets data on fast devices, and a substantial portion of the updates in the YCSB-A workload can be completed solely on fast devices.

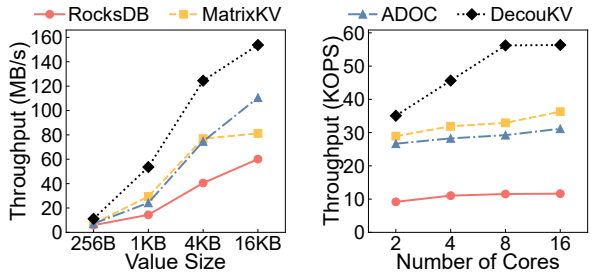
Scheduling strategies. Similarly, by examining the differences between DecouKV-DS and DecouKV-D, we can roughly assess that the scheduling queue technique has a greater contribution to the load workload. To further understand this, we record the variations in the parameters DFTS and IMTN during load and YCSB-A workload in the 2000s, as shown in Figure 16. After decoupling, during the load workload, DFTS is significantly increased, indicating a clear performance bottleneck in I/O. DecouKV mitigates this bottleneck by increasing DFTS to schedule more CPU-intensive operations, thus reducing the demand for I/O resources. In contrast, during the YCSB-A workload, DFTS and IMTN are not significantly adjusted, suggesting that the CPU and I/O resources are well balanced after decoupling, and no excessive scheduling is required. Therefore, the scheduling queue technique contributes more to load workload.

Elastic capacity high levels. The performance difference between DecouKV and DecouKV-DS demonstrates that the elastic capacity for high levels technique provides greater ben-



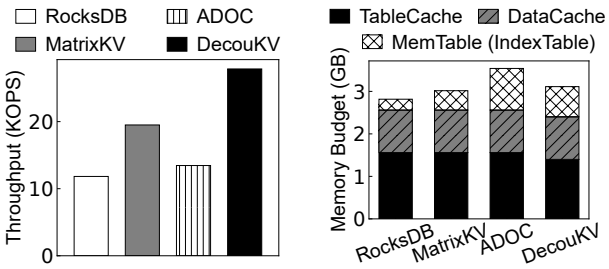
(a) YCSB workloads with 500 million KV pairs. (b) Nutanix.

Figure 17: Performance under other workloads.



(a) Throughput w/ varying V-size. (b) Throughput w/ varying core#.

Figure 18: Performance impact of varying value sizes/core counts.



(a) Throughput on NVMe SSDs. (b) Memory budget comparison.

Figure 19: Comparison of throughput after replacing fast devices with NVMe SSDs and memory budget.

efits for load workload compared to YCSB-A workload. This is primarily because the load workload has a more significant impact on the high levels than the YCSB-A workload. Moreover, the elastic capacity high levels technique further mitigates the interdependencies among operations, minimizing the impact of excessive data sorting tasks on other operations during the write-intensive load phase.

4.4 Performance under Other Workloads

YCSB workloads with a huge size dataset. We conduct performance evaluations on a larger YCSB uniform workload, which includes a 500GB dataset (i.e., 500 million key-value pairs), as shown in Figure 17(a). Compared to RocksDB, DecouKV demonstrates performance improvements of 5.2×, 2.8×, and 2.4× under write-intensive load, YCSB-A, and YCSB-F workloads, respectively, consistent with the results obtained with the 100GB dataset. In comparison to MatrixKV and ADOC, DecouKV achieves performance improvements of 1.6–2.4× under the same workloads, even slightly outperforming the results from the 100GB dataset. This is attributed to the slight performance degradation observed in MatrixKV and ADOC with larger datasets, whereas DecouKV maintains its performance advantage due to more efficient utilization of system resources.

Nutanix production workloads. We also evaluate the throughput performance of various KV systems using a real-world workload from Nutanix, which closely resembles a write-intensive workload with 57% updates, 41% reads, and 2% scans. The results, shown in Figure 17(b), demonstrate that DecouKV outperforms other systems, achieving a throughput improvement of 1.3–2.0×.

4.5 Understanding DecouKV Performance

Impact of KV size. We evaluate the variation in throughput performance of the load under different value sizes. As shown in Figure 18(a), DecouKV demonstrates the most significant advantage when the value size is 1KB and 4KB. It is known that as the value size increases, the demand for I/O bandwidth also increases. Conversely, when the value size decreases, the load's demand for CPU resources becomes more significant. When the value size is too large (16KB) or too small (256B), severe imbalances in the demands for CPU and I/O resources results in performance degradation, leading to resource shortages of a single type. Consequently, the benefits of decoupling decrease, and ADOC, which smooths the data flow through scheduling, still performs well.

Impact of core number. We evaluate the impact of the number of CPU cores on load performance. As shown in Figure 18(b), when the number of CPU cores is low (e.g., 2 cores), the performance improvement of DecouKV is modest due to limited CPU resources. As the number of cores increases, DecouKV 's performance advantage becomes more pronounced, indicating that DecouKV can better leverage the additional CPU resources. However, when CPU resources become excessive (e.g., 16 cores), the performance of DecouKV no longer increases, as it becomes constrained by disk I/O.

Performance on different devices. We evaluate the throughput performance of DecouKV on different fast devices. Specifically, we use the INTEL SSDPE2KE032T8 NVMe SSD as the fast device and leverage io-uring to fully exploit the asynchronous I/O capabilities of the NVMe SSD. As shown in Figure 19(a), DecouKV demonstrates a throughput improvement of 1.4–2.4× over RocksDB, MatrixKV, and ADOC when deployed on hybrid storage devices consisting of an NVMe SSD and a SATA SSD.

Memory budget. We record the maximum memory usage during the experiments, with YCSB-A requiring the highest memory. As shown in Figure 19(b), about 23% (0.7GB) of memory is used for the IndexTable–more than RocksDB's MemTable (0.3GB). Approximately 45% (1.4GB) is used

for DecouKV's Tablecache, less than RocksDB's (1.56GB), because there's no need to cache the AOF's Filter Block and Index Block. The memory usage of DecouKV is comparable to that of MatrixKV and ADOC, with only a slight increase over RocksDB, while achieving higher performance.

Recovery time. We also test the recovery capability of DecouKV in the event of system crashes. Specifically, we verify that data persisted in the AOFs on fast devices before a crash can be used to rebuild the index upon reboot. The results show that DecouKV successfully recovers after multiple crash tests. In terms of performance, the recovery speed of DecouKV and RocksDB is nearly identical, with recovery times of 8.76s and 8.12s, respectively, for a 10GB database. This is because the high sequential read bandwidth provided by the fast devices enables rapid index rebuilding.

5 Related Work

Performance optimization of LSM-tree-based KV stores. A lot of research focus on optimizing the performance of LSM-tree-based KV stores. One line of investigation addresses the issue of write stalls. SILK [4] aims to mitigate the tail latencies caused by write stalls by adjusting the priority of background threads and rate-limiting their operations. ADOC identifies data overflow in various components of LSM trees as the direct cause of write stalls and fine-tunes parameters such as the number of threads and batch size to schedule flush and compaction operations more effectively. Another line of research focuses on reducing write amplification through KV separation. WiscKey [35] is the first to propose storing values in a separate append-only log, while Bourbon [12] extends WiscKey by indexing values through a learned approach. HashKV [8] and BadgerDB [16] have optimized the additional overhead introduced by garbage collection in the context of KV separation. Other studies focus on optimizing read and scan performance, including systems such as LSM-trie [46], UniKV [49], ElasticBF [34], and DiffKV [33].

Deploying LSM-tree-based KV stores on new devices. Recent studies also focus on new types of storage media, with some research targeting hybrid storage solutions, similar to the scenario discussed in this paper. NoveLSM extends memory components to PM and employs in-place updates to reduce compaction overhead. SLM-DB [28] improves upon this by constructing a global B+-tree index on PM and using optional compaction to minimize write amplification. MatrixKV identifies L_0-L_1 compaction as the root cause of PM device write stalls and addresses this issue by using matrix containers for differentiated management of L_0 on PM. Fine-grained matrix compression is employed to reduce write stalls.

Besides, some works [6, 42, 45] mainly focus on different scenarios like workload skewness or read-intensive workloads. PrismDB modifies the hierarchical structure to reduce compaction overhead and uses a cold-hot awareness approach to improve read performance. SplitDB employs a cascading skip list index and a data promotion and demotion mechanism that migrates hot data from slower SSDs to faster NVM and cold data back to SSDs, thereby optimizing read performance under skewed workloads. PRISM [45] analyzes the advantages of each storage device and adopts hybrid management for heterogeneous storage. However, abandoning the tree structure leads to some loss in scan performance and complicates garbage collection management. Other studies focus on scenarios where cost is not a primary concern, and fast devices nearly replace slow ones. KVell [32] relaxes data ordering and employs a share-nothing design to fully utilize the bandwidth of fast devices. ListDB [30] deploys the entire LSM-tree on a system with only DRAM and PM and replaces SSTables with a NUMA-aware skip list. MioDB [17] leverages the byte-addressable feature of PM, managing data on PM using skip lists and performing zero-copy compaction for merging between skip lists.

Tuning and scheduling of LSM-tree-based KV stores. Many studies observe that configuration settings significantly impact the performance of LSM-trees. Monkey [13] proposed a configuration tuning framework that optimizes memory allocation strategies and read/write performance under worst-case scenarios. Dostoevsky [14] employs a hybrid strategy to determine the levels at which compaction occurs. Rafiki [37] and TiKV [41] utilize offline training methods to compute optimal configuration settings. These works achieve substantial performance improvements in their respective scenarios; however, their tuning and scheduling approaches are often highly dependent on workload characteristics.

6 Conclusion

In this work, we analyze the data sorting operations of sorting-based KV stores and identify operation coupling as the root cause of resource usage dependency and fragmentation on hybrid storage devices. While two existing approaches—fixed differentiated data management and superficial scheduling of data sorting operations—fail to fundamentally address the problem of inefficient resource utilization.

We propose DecouKV to mitigate resource usage dependency in sorting-based KV stores on hybrid storage devices via operation decoupling. Experimental results demonstrate that, compared to RocksDB and the aforementioned solutions, DecouKV improves CPU utilization by 25.4%–32.3% and increases throughput by 2.3–4.9× under write-intensive workloads, while reducing tail latency by 74.3%–91.4%. Furthermore, under read-intensive workloads, DecouKV also achieves a modest throughput improvement of 1.2-2.3×.

7 Acknowledgement

We thank our shepherd, Jinkyu Jeong, and the anonymous reviewers for their valuable comments. This work was supported in part by NSFC (62472392, 62172382) and the Youth Innovation Promotion Association CAS.

A Artifact Appendix

Abstract

Our artifact includes the prototype implementation of DecouKV and three other state-of-the-art comparison systems, along with the YCSB benchmark evaluated in the Evaluation section. Additionally, we provide the experimental scripts necessary to reproduce our results on hybrid devices. Please note that running the full evaluation requires several tens of hours to complete.

Scope

This artifact allows reviewers to reproduce the main experimental results of the paper, including:

- *DecouKV improves CPU utilization compared to other systems and reduces utilization fluctuations under write-intensive workloads (Load, YCSB-A).*
- *DecouKV achieves higher throughput than other systems under write-intensive workloads (Load, YCSB-A, -F); it also provides moderate improvements under read- or scan-intensive workloads (YCSB-B, -C, -D, -E).*
- *DecouKV reduces average latency compared to other systems under write-intensive workloads; it also brings slight improvements under read- or scan-intensive workloads.*

Contents

The artifact is hosted in a git repository. This repository includes DecouKV's source code as well as documentation and experimental scripts. It is structured as follows:

db_impl/: *This folder contains the source code implementation of DecouKV, as well as three other comparison systems (RocksDB, MatrixKV and ADOC). Users can switch between the test systems by modifying the value of 'systems' in the script. The source code is structured as follows:*

- ***adoc/****: This folder contains the source code implementation of ADOC.*
- ***decoukv/****: This folder contains the source code implementation of DecouKV.*
- ***matrixkv/****: This folder contains the source code implementation of MatrixKV.*
- ***rocksdb/****: This folder contains the source code implementation of RocksDB.*

test_results/: *This folder is automatically created after running the test script, and the test results will be saved here. The* ***ycsb_latency.csv*** *file stores the latency results after running the test, while the* ***ycsb_throughput.csv*** *file stores the throughput results.*

workloads/: *This folder contains the main parameters for the YCSB workload. Each YCSB workload file can be modified to change parameters such as database size (recordcount), number of operations (operationcount), read/write ratio, and key distribution (Uniform or Zipfian).*

db_config.yaml: *This file includes some system-level configurations that can be customized, including SSTable size, MemTable size, and other internal parameters.*

test.sh:*This file is the unified test script. It will compile all four systems, run Load, YCSB-A to F workloads, and output the CPU utilization fluctuation plot into* ***./test_results/****. This step may take over 50 hours to complete, and you can monitor the script's progress in the* ***test.log*** *file. Reducing the data volume (default: 100GB) can shorten the runtime, but it may affect the final results.*

Hosting

The artifact is hosted on GitHub at: `https://github.com/QingyangZ/DecouKV.git`*. Please refer to the* `main` *branch and the latest commit version for the exact code used in this paper.*

Requirements

The artifacts have been developed and tested on Ubuntu 20.04.6 LTS with Linux kernel 5.15. The hardware includes 128 GB Intel Optane DCPMM for fast devices and 960 GB Intel S4520 SSDs with SATA 3.0 interfaces for slow devices. Approximate resource requirements are a dedicated server with at least 10 CPU cores, one SSD with over 200GB of free space, and one PMEM device with over 50GB of free space.

A.1 Installation

1. *First, to clone the repository, run the following commands:* `git clone https://github.com/QingyangZ/DecouKV.git`*.*
2. *Then, to initialize and update the submodules, run:*

```
git submodule init
git submodule update
```

3. *Next, to install the required packages, use:*

```
sudo apt install libsnappy-dev
libgflags-dev zlib1g-dev libbz2-dev
liblz4-dev libzstd-dev
libpthread-stubs0-dev libnuma-dev
libstdc++-dev libpmem-dev liburing-dev
```

A.2 Experiment workflow

To run the full experiment, execute the `test.sh` *script in the background. This script compiles all four systems, runs* ***Load*** *and* ***YCSB-A to F*** *workloads, and outputs the CPU utilization fluctuation plot into the* `test_results/` *directory. After the test completes, generate throughput and latency figures by running* `draw_throughput.sh` *and* `draw_latency.sh`*.*

References

[1] Abutalib Aghayev, Sage Weil, Michael Kuchnik, Mark Nelson, Gregory R Ganger, and George Amvrosiadis. File systems unfit as distributed storage backends: lessons from 10 years of ceph evolution. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 353–369, 2019.

[2] Anandtech. Intel Launches Optane DIMMs Up To 512GB: Apache Pass Is Here! `https://github.com/facebook/rocksdb/wiki/RocksDB-Tuning-Guide`, 2018.

[3] Apache. cassandra-3.11.4. `https://github.com/apache/cassandra/tree/cassandra-3.11.4`.

[4] Oana Balmau, Florin Dinu, Willy Zwaenepoel, Karan Gupta, Ravishankar Chandhiramoorthi, and Diego Didona. SILK: Preventing latency spikes in Log-Structured merge Key-Value stores. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 753–766, 2019.

[5] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, et al. TAO:Facebook's distributed data store for the social graph. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 49–60, 2013.

[6] Miao Cai, Xuzhen Jiang, Junru Shen, and Baoliu Ye. Splitdb: Closing the performance gap for lsm-tree-based key-value stores. *IEEE Transactions on Computers*, 73(1):206–220, 2024.

[7] Wei Cao, Yang Liu, Zhushi Cheng, Ning Zheng, Wei Li, Wenjie Wu, Linqiang Ouyang, Peng Wang, Yijing Wang, Ray Kuan, et al. POLARDB meets computational storage: Efficiently support analytical workloads in Cloud-Native relational database. In *18th USENIX conference on file and storage technologies (FAST 20)*, pages 29–41, 2020.

[8] Helen HW Chan, Chieh-Jan Mike Liang, and et al. HashKV: Enabling efficient updates in KV storage via hashing. In *Proc. of USENIX ATC*, 2018.

[9] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A Distributed Storage System for Structured Data. In *Proc. of USENIX OSDI*, 2006.

[10] Hao Chen, Chaoyi Ruan, Cheng Li, Xiaosong Ma, and Yinlong Xu. SpanDB: A fast,Cost-EffectiveLSM-tree based KV store on hybrid storage. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 17–32, 2021.

[11] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154, 2010.

[12] Yifan Dai, Yien Xu, Aishwarya Ganesan, Ramnatthan Alagappan, Brian Kroth, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. From WiscKey to bourbon: A learned index for Log-Structured merge trees. In *Proc. of USENIX OSDI*, 2020.

[13] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. Monkey: Optimal navigable key-value store. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 79–94, 2017.

[14] Niv Dayan and Stratos Idreos. Dostoevsky: Better space-time trade-offs for lsm-tree based key-value stores via adaptive removal of superfluous merging. In *Proceedings of the 2018 International Conference on Management of Data*, pages 505–520, 2018.

[15] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, and et al. Dynamo: Amazon's Highly Available Key-value Store. In *Proc. of ACM SOSP*, 2007.

[16] Dgragh. BadgerDB. `https://github.com/dgraph-io/badger`.

[17] Zhuohui Duan, Jiabo Yao, Haikun Liu, Xiaofei Liao, Hai Jin, and Yu Zhang. Revisiting log-structured merging for kv stores in hybrid memory systems. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 674–687, 2023.

[18] Nima Elyasi, Changho Choi, and Anand Sivasubramaniam. Large-Scale graph processing on emerging storage devices. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 309–316, 2019.

[19] Robert Escriva. HyperLevelDB. `https://github.com/rescrv/HyperLevelDB/`.

[20] Facebook. RocksDB. `https://rocksdb.org`.

[21] Facebook. RocksDB Tuning Guide. `https://github.com/facebook/rocksdb/wiki/RocksDB-Tuning-Guide`.

[22] S. Ghemawat and J. Dean. LevelDB. https://leveldb.org.

[23] Tyler Harter, Dhruba Borthakur, Siying Dong, Amitanand Aiyer, Liyin Tang, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Analysis of HDFS under HBase: A facebook messages case study. In *12th USENIX Conference on File and Storage Technologies (FAST 14)*, pages 199–212, 2014.

[24] Dongxu Huang, Qi Liu, Qiu Cui, Zhuhe Fang, Xiaoyu Ma, Fei Xu, Li Shen, Liu Tang, Yuxing Zhou, Menglong Huang, et al. Tidb: a raft-based htap database. *Proceedings of the VLDB Endowment*, 13(12):3072–3084, 2020.

[25] Gui Huang, Xuntao Cheng, Jianying Wang, Yujie Wang, Dengcheng He, Tieying Zhang, Feifei Li, Sheng Wang, Wei Cao, and Qiang Li. X-engine: An optimized storage engine for large-scale e-commerce transaction processing. In *Proceedings of the 2019 International Conference on Management of Data*, pages 651–665, 2019.

[26] Xiaowei Jiang, Yuejun Hu, Yu Xiang, Guangran Jiang, Xiaojun Jin, Chen Xia, Weihua Jiang, Jun Yu, Haitao Wang, Yuan Jiang, et al. Alibaba hologres: A cloud-native service for hybrid serving/analytical processing. *Proceedings of the VLDB Endowment*, 13(12):3272–3284, 2020.

[27] Myoungsoo Jung. Hello bytes, bye blocks: Pcie storage meets compute express link for memory expansion (cxl-ssd). In *Proceedings of the 14th ACM Workshop on Hot Topics in Storage and File Systems*, pages 45–51, 2022.

[28] Olzhas Kaiyrakhmet, Songyi Lee, Beomseok Nam, Sam H Noh, and Young-ri Choi. SLM-DB:Single-LevelKey-Value store with persistent memory. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 191–205, 2019.

[29] Sudarsun Kannan, Nitish Bhat, Ada Gavrilovska, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. Redesigning LSMs for Nonvolatile Memory with NoveLSM. In *Proc. of USENIX ATC*, 2018.

[30] Wonbae Kim, Chanyeol Park, Dongui Kim, Hyeongjun Park, Young-ri Choi, Alan Sussman, and Beomseok Nam. ListDB: Union of Write-Ahead logs and persistent SkipLists for incremental checkpointing on persistent memory. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 161–177, 2022.

[31] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. GraphChi:Large-Scale graph computation on just a PC. In *10th USENIX symposium on operating systems design and implementation (OSDI 12)*, pages 31–46, 2012.

[32] Baptiste Lepers, Oana Balmau, Karan Gupta, and Willy Zwaenepoel. Kvell: the design and implementation of a fast persistent key-value store. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 447–461, 2019.

[33] Yongkun Li, Zhen Liu, Patrick PC Lee, Jiayu Wu, Yinlong Xu, Yi Wu, Liu Tang, Qi Liu, and Qiu Cui. Differentiated Key-Value storage management for balanced I/O performance. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 673–687, 2021.

[34] Yongkun Li, Chengjin Tian, Fan Guo, Cheng Li, and Yinlong Xu. ElasticBF: Elastic bloom filter with hotness awareness for boosting read performance in large Key-Value stores. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 739–752, 2019.

[35] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Hariharan Gopalakrishnan, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Wisckey: Separating keys from values in ssd-conscious storage. *ACM Transactions On Storage (TOS)*, 13(1):1–28, 2017.

[36] Siqiang Luo, Subarna Chatterjee, Rafael Ketsetsidis, Niv Dayan, Wilson Qin, and Stratos Idreos. Rosetta: A robust space-time optimized range filter for key-value stores. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 2071–2086, 2020.

[37] Ashraf Mahgoub, Paul Wood, Sachandhan Ganesh, Subrata Mitra, Wolfgang Gerlach, Travis Harrison, Folker Meyer, Ananth Grama, Saurabh Bagchi, and Somali Chaterji. Rafiki: A middleware for parameter tuning of nosql datastores for dynamic metagenomics workloads. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference*, pages 28–40, 2017.

[38] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. The log-structured merge-tree (lsm-tree). *Acta Informatica*, 33:351–385, 1996.

[39] Anastasios Papagiannis, Giorgos Saloustros, Pilar González-Férez, and Angelos Bilas. Tucana: Design and implementation of a fast and efficient scale-up key-value store. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 537–550, 2016.

[40] Markus Pilman, Kevin Bocksrocker, Lucas Braun, Renato Marroquin, and Donald Kossmann. Fast scans on key-value stores. *Proceedings of the VLDB Endowment*, 10(11):1526–1537, 2017.

[41] PingCAP. TiKV. `https://tikv.org`.

[42] Ashwini Raina, Jianan Lu, Asaf Cidon, and Michael J Freedman. Efficient compactions between storage tiers with prismdb. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pages 179–193, 2023.

[43] J. REN. YCSB-C. `https://github.com/basicthinker/YCSB-C`.

[44] Giampaolo Rodola. psutil documentation. `https://psutil.readthedocs.io`.

[45] Yongju Song, Wook-Hee Kim, Sumit Kumar Monga, Changwoo Min, and Young Ik Eom. Prism: Optimizing key-value store for modern heterogeneous storage devices. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 588–602, 2023.

[46] Xingbo Wu, Yuehai Xu, Zili Shao, and Song Jiang. LSM-trie: An LSM-tree-based Ultra-Large Key-Value Store for Small Data. In *Proc. of USENIX ATC*, 2015.

[47] Ting Yao, Yiwen Zhang, Jiguang Wan, Qiu Cui, Liu Tang, Hong Jiang, Changsheng Xie, and Xubin He. MatrixKV: Reducing write stalls and write amplification in LSM-tree based KV stores with matrix container in NVM. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 17–31, 2020.

[48] Jinghuan Yu, Sam H Noh, Young-ri Choi, and Chun Jason Xue. ADOC: Automatically harmonizing dataflow between components in Log-StructuredKey-Value stores for improved performance. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*, pages 65–80, 2023.

[49] Qiang Zhang, Yongkun Li, Patrick PC Lee, Yinlong Xu, Qiu Cui, and Liu Tang. Unikv: Toward high-performance and scalable kv storage in mixed workloads via unified indexing. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, pages 313–324. IEEE, 2020.

[50] Qingyang Zhang. DecouKV. `https://github.com/QingyangZ/DecouKV.git`.

[51] Da Zheng, Disa Mhembere, Randal Burns, Joshua Vogelstein, Carey E Priebe, and Alexander S Szalay. FlashGraph: Processing Billion-Node graphs on an array of commodity SSDs. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 45–58, 2015.

[52] Wenshao Zhong, Chen Chen, Xingbo Wu, and Song Jiang. REMIX: Efficient range query for LSM-trees. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 51–64, 2021.

SolFS: An Operation-Log Versioning File System for Hash-free Efficient Mobile Cloud Backup

Riwei Pan[†] Yu Liang[§] Lei Li[†] Hongchao Du[†] Tei-Wei Kuo[¶] Chun Jason Xue[‡]
[†]Department of Computer Science, City University of Hong Kong [§]ETH Zürich
[¶]Delta Electronics and National Taiwan University
[‡]Mohamed bin Zayed University of Artificial Intelligence

Abstract

Mobile cloud backup applications are widely used to safeguard user data. This paper found that current cloud backup is inefficient on resource-limited mobile devices because it consumes excessive CPU resources for delta synchronization that requires intensive hash computation to identify the modified ranges of file data. To address this issue, this paper presents SolFS, an operation log versioning file system to optimize mobile cloud backup efficiency. The core idea is that if the cloud backup application knows the modified offset and length of each write since the last backup, it will be able to identify the new modified data and upload them only, avoiding data hashing throughout the entire file. SolFS proposes a series of designs to achieve this design goal. First, SolFS introduces per-file mergeable operation logging that allows each file to manage its write operation logs (i.e., offset and length) in an extent tree and merge operation logs with contiguous or overlapping modified ranges of file data. Then, SolFS proposes the operation log persistence and versioning mechanism that allows different cloud backup applications to manage their own file versions without interfering with each other. In addition, SolFS incorporates techniques such as compact log and dynamic granularity, to optimize the memory and storage overhead to the system. Finally, SolFS achieves hash-free file difference identification with minimum additional overhead and extends the ability of cloud backup applications. The experimental results show that SolFS can significantly reduce the computational overhead of both APP-side or server-side by over 90% on average and the total cloud synchronization time by over 88.8% when files are updated.

1 Introduction

Mobile devices have become an essential part of our daily lives, with around 1.34 billion smartphones sold in 2023 [32]. These devices are now an important venue for personal data storage, making data loss a serious concern. To mitigate this risk, mobile cloud backup applications (APPs) are widely adopted to protect user data, such as Dropbox, OneDrive, and smartphone vendors' built-in cloud backup APPs. These mobile APPs usually choose to upload the entire file data, but this approach could result in high network traffic and low backup efficiency, especially when uploading files with minor data changes. Therefore, some APPs adopt delta synchronization to speed up the cloud synchronization time by leveraging hashing to identify and upload modified file data only [3,9,10,23,34]. However, the intensive hash computation caused by delta synchronization increases CPU utilization, resulting in excessive power consumption. In addition, it also leads to conflicting demands of limited mobile CPU resources between cloud backup APPs and other user applications. On the resource-limited mobile devices, the high cost could directly affect the user experience in daily usage, such as shortening the mobile device's battery life [9, 16]. Hence, these cloud backup APPs face a dilemma between cloud backup efficiency and user experience. This paper aims to improve cloud backup efficiency on mobile devices, with a particular focus on overcoming the dilemma between high network traffic and intensive CPU resource consumption.

There are many previous works to improve the efficiency of delta synchronization [5, 6, 10, 11, 24, 34, 36, 38]. A typical design is to divide a file into multiple chunks and then calculate their checksum (i.e., strong hash algorithms like MD5 and SHA256). By comparing the local file's checksum list with that from the cloud server, this approach can identify the modified chunks and upload them only to the cloud server to reduce network traffic, but the minimum network traffic transmitted is limited by the chunk size. The *rsync* [34] algorithm further improves this issue, which employs rolling checksum to identify byte-level modified data before performing chunk-level hashing. One issue of rsync is the high cost at the client side, because rolling checksum requires byte-by-byte comparison. *WebR2sync+* [38] optimizes the client-side overhead of rsync by moving the heavy byte-by-byte comparison to the server side and then the client only needs to perform chunk-level weak hashing and strong hashing. *QuickSync* [9] adopts

*Corresponding author: Yu Liang: yulianglenny@gmail.com.

the Content-defined Chunking (CDC) to detect redundant bytes and *NetSync* [36] reduces the computation cost of the chunk matching process with a fast CDC-based weak hashing. In addition, some researchers introduce lightweight hash algorithms to improve their computational cost [22, 37, 44]. However, on resource-limited mobile devices, these works still impose a high computational cost on identifying old and new modified data.

To address this issue in delta synchronization, we present SolFS, a lightweight operation log versioning file system built with existing mobile file systems. The key idea behind SolFS is to identify old data and new modified data through write operation logs (i.e., each write's offset and length) rather than hashing, thus significantly mitigating the computational cost. However, incorporating this idea into the file system introduces many challenges, including extra access overhead, more resource usage, and file version management across different cloud synchronization and cloud backup applications. SolFS presents a series of designs to overcome these challenges.

First, to reduce the memory and storage usage for operation logs, SolFS proposes per-file mergeable operation logging (or MLogging). MLogging allows each file to manage its own write operation logs in an extent tree that merges operation logs with contiguous or overlapping modification ranges. In addition, MLogging introduces the mlog cache design to improve the logging cost in sequential write operations and the on-demand loading design to minimize the memory usage. SolFS needs to persist operation logs to the disk after file data writebacks for subsequent modification range tracking. To minimize the storage cost, SolFS further introduces dynamic log granularity and compact operation logs. Unlike traditional versioning techniques, SolFS performs versioning for operation logs rather than file data, thus avoiding the issues of file fragmentation and double storage usage. A file's different versions of operation logs are stored and managed in a versioned inode chain, which enables cloud backup APPs to accurately track their individual cumulative modifications based on version number without interfering with each other. Therefore, cloud backup APPs are able to identify old data and new modified data without hashing the entire file to improve the efficiency of delta synchronization.

We evaluate SolFS's performance on a commercial smartphone and compare it with existing delta synchronization solutions. The experimental results show that SolFS can significantly reduce the computational overhead of both APP-side or server-side by over 90% on average and the total cloud synchronization time by over 88% when files are updated.

2 Background and Motivation

2.1 Cloud Backup on Mobile Devices

Mobile cloud backup APPs are commonly utilized to ensure user data security and save storage space by backing up various types of files, such as executable files, database files, and text documents. These APPs can be divided into third-party APPs and built-in APPs. Third-party APPs, such as Dropbox, aim to back up users' personal files and allow users to access their files elsewhere. Built-in APPs are provided by smartphone vendors and can further back up applications' internal files. This enables users to restore their mobile devices to a specific state if their devices are reset or lost.

Cloud backup APPs typically follow a design [11] with two major steps: reading file data and uploading file data to the cloud. For the second step, some APPs adopt delta synchronization to save the network bandwidth and to speed up the upload process [3, 9, 10, 23, 34], while other APPs choose to upload entire file data to avoid intensive hash computation and excessive power consumption caused by delta synchronization. We present and categorize common cloud backup APPs in Table 1 based on our observations.

Table 1: Delta synchronization support in seven common cloud backup APPs.

Mobile APP	Type	Delta Synchronization
Dropbox	Third-party	Not support
OneDrive	Third-party	Not Support
GoogleDrive	Third-party	Not Support
Seafile	Third-party	Supported
Huawei	Built-in	Supported
Samsung	Built-in	Supported
Oppo	Built-in	Supported

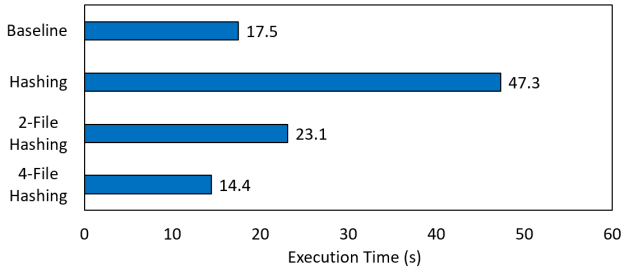
Importance of Frequent Cloud Backup. A fast, timely, and low-overhead file data cloud synchronization scheme has significant practical importance. Delta synchronization is very useful to reduce network traffic and improve upload efficiency in unstable mobile networks [36]. However, it relies on computationally intensive hash calculations and comparisons to identify file differences [3, 9, 10, 23, 34]. Such a high cost drives smartphone vendors and users to restrict their cloud backup frequency (e.g., once a week) to minimize the impact on user experience. However, the most recent user data, such as newly edited documents, is typically more valuable and should be backed up instantly, as cloud backup applications (e.g., Dropbox) do on desktops/servers [9, 24]. APP-specific data in health-care-related or business-related applications, could be important for users to have up-to-date backups because of important body status data and high-value business documents. In addition, as the volume of user data continues to grow, the limited storage space on a single device could become insufficient to store all user data, further making cloud data synchronization an increasingly frequent operation. These scenarios suggest that there is a dire need for mobile users to frequently conduct mobile cloud backups. Mobile users need a low-computation-cost cloud backup mechanism that allows them to avoid compromising between performance and data security.

2.2 Expensive File Hashing

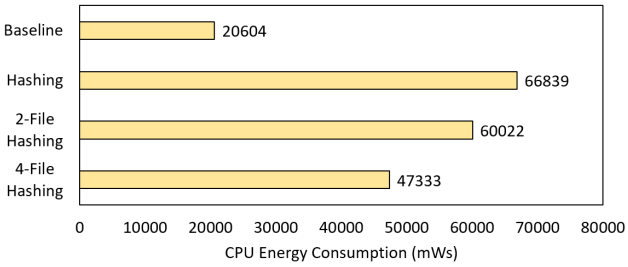
The current delta synchronization efficiency is not satisfactory on mobile devices, as it needs to perform hashing throughout the entire file to locate the modified ranges of data. Specifically, a typical design of delta synchronization involves: 1) the server divides a file into multiple chunks and computes each chunk's checksum based on hashing (e.g., SHA256 and MD5); 2) the file's checksum list is sent to the APP and the APP also performs chunking and hashing for the file to identify and upload the chunks with different checksum [3]. Despite how much file data is modified, this approach requires the hash calculation and comparison processing of all file data to identify data differences, leading to a long identification time and high CPU usage [31, 38]. It is expensive for resource-limited mobile devices because the excessive use of CPU resources not only degrades the user experience (competing with other applications for CPU resources), but also shortens the battery life of the mobile device in daily usage [9, 16]. To quantitatively evaluate this impact on a recent mobile device (Google Pixel 8), we develop an APP following this typical delta synchronization design.

This APP backs up 15GB of data in total[1] and the backup procedure includes reading the files, dividing them into chunks, and leveraging the built-in SHA256 algorithm for hashing, similar to the content hash used by the desktop version of Dropbox [12]. As depicted in Figure 1, *Hashing* introduces an additional 170% execution latency and 224% CPU energy consumption on average due to the intensive computation, compared to reading files only (*Baseline*). In addition, we also notice that the read operation contributes around 30.8% of total cost during hashing. To achieve better performance, the APP can perform parallel hashing for multiple files rather than single-threaded hashing, such as *2-File Hashing* and *4-File Hashing*. The parallel hashing could achieve a lower execution time compared with the single-threaded reading (4-File Hashing v.s. Baseline), but consumes more CPU resources at the same time, increasing the competition of CPU resources with other applications. Furthermore, its energy consumption with four threads (i.e., 47333 mWs) is still more than twice that of the baseline (i.e., 20604 mWs).

There are previous works available on the desktop/server platform to improve the upload efficiency of delta synchronization. Rsync [34] is proposed to employ a weak hash algorithm called rolling checksum, which uses a sliding window and performs byte-by-byte comparison to obtain byte-level file differences instead of that of chunk-level. This algorithm is easily prone to hash collisions and thus rsync has to use strong hashing for all unmodified data chunks to ensure that these data chunks are truly not modified. Hence, rsync still consumes many CPU resources on the APP side. WebR2sync+ [38] improves this issue by moving the heavy byte-by-byte comparison to the cloud server side, but the weak and strong hashing are still required on the APP side. *QuickSync* [9] adopts the Content-defined Chunking (CDC) to detect redundant bytes during network transmission and *NetSync* [36] employs the CDC technique to further avoid the computation cost of overlapping chunks. These previous efforts have focused on reducing the computation cost during delta synchronization; however, all these approaches still require hashing the entire file data to identify data differences, making intensive computation unavoidable, increasing CPU loads and energy consumption on mobile devices.



(a)



(b)

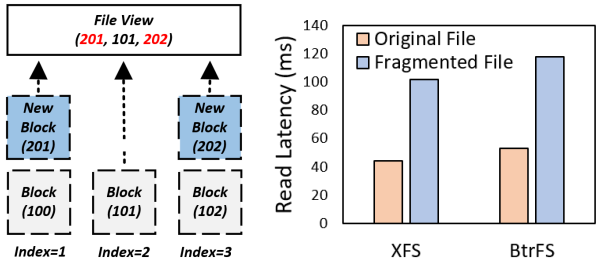
Figure 1: (a) execution time in seconds; (b) CPU energy consumption (mWs). The energy data is collected by the On Device Power Rails Monitor [2]. For reference, the device energy consumption from top to bottom is 37520, 120136, 86911, 65222 mWs, respectively.

Reducing computation is challenging. We rethink the nature of delta synchronization—uploading only modified file data. In other words, reading and hashing entire file data may be unnecessary if we have a proper system mechanism to identify the range of file changes. Then, we could either directly upload the modified range of data, or perform hashing on this selected range to further reduce network traffic, because sometimes the data after modification is the same as before modification. As small writes dominate [8, 19] on mobile devices, it is expected that this hash-free identification approach could reduce the computation cost on mobile devices by avoiding hashing the entire file data. Therefore, file system is a very suitable scenario to implement this idea, because it can instantly record file changes without incurring a high overhead. For example, the file snapshotting and versioning techniques in file system design [20, 28–30], such as the reflink feature of XFS and BtrFS, have the potential ability to achieve this goal. The copy-on-write (CoW) approach

[1] As a reference, we collected used storage space from 35 real users' mobile devices and found that each device stored 118GB of data on average.

adopted in these works allows cloud backup APPs to identify old data and new modified file data without hashing.

As shown in Figure 2(a), the CoW approach always allocates new blocks for new modified data (e.g., PBA 201 and 202), while old blocks (PBA 100-102) serve as old data, allowing the file system to classify them. However, this design will introduce file fragmentation issues after multiple file updates [15, 18, 39]. For example, the file view in Figure 2(a) includes PBA 201, 101, 202, becoming non-consecutive after updating data in a CoW manner. Figure 2(b) presents the impact of file fragmentation that could remarkably degrade the sequential read performance. BetrFS [42] tries to mitigate this drawback based on a copy-on-abundant-write design and WaybackFS [7] avoids file fragmentation by replacing old data with an undo log. However, supporting these features on F2FS [21], the most widely used file system on mobile devices, is complex and often requires changes to the disk layout [43]. In addition, all these solutions, whether based on CoW or undo logs, have to continuously allocate additional memory and storage space to manage multiple versions of file data, which is still expensive for resource-limited mobile devices.



(a) (b)

Figure 2: (a) CoW-based hash-free solution, where the number in each block presents a physical block address (PBA); (b) sequential read latency in a 10MB file. The fragmented file is created based on the reflink feature and we randomly update 1MB data on it.

3 SolFS Design

In this section, we introduce *SolFS* to cope with the mentioned challenges. SolFS[2] is a lightweight *operation log versioning file system* to mitigate hash computation during delta synchronization while overcoming the limitations of snapshotting and versioning file systems. The main design goals of SolFS involve:

- **Hash-free Identification**: SolFS is designed to identify modified ranges of file data without hashing by the collaboration with file system.
- **Overhead Minimization**: SolFS minimizes the I/O, memory and storage overhead discussed in Section 2.2.
- **Generality**: SolFS supports multiple backup APPs (e.g., built-in and third-party) on the same device to manage their respective file differences without interfering with each other.
- **Backward Compatibility**: There are over 6.5 billion of mobile devices in the world [13] and we expect that all existing mobile devices could benefit from SolFS.

[2]The source code of SolFS is available at https://github.com/MIoTLab/SolFS.

3.1 Architecture Overview

To achieve these design goals, SolFS introduces several effective designs to existing mobile file systems, as shown in Figure 3. Our designs are based on the observation that if the cloud backup APP knows the modified offset and length of each write since the last backup, it can accurately identify newly modified data. This allows cloud backup APPs to selectively upload the modified data directly, or apply hashing (e.g., chunk-level hashing and rolling hashing) to only the modified data range and then upload it without requiring hashing throughout the entire file.

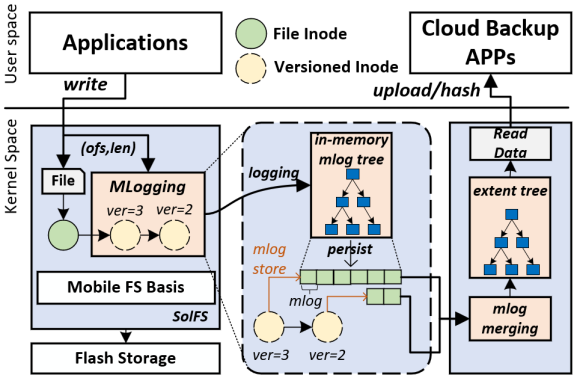


Figure 3: SolFS architecture (shown in blue).

Therefore, SolFS first introduces per-file mergeable operation logging (or *MLogging*, see Section 3.2), which keeps track of each write's offset and length by recording them as operation logs in a new auxiliary inode. This design avoids the file fragmentation issue when the file is continuously updated and minimizes the storage cost of operation logs by not storing actual file data. Specifically, MLogging enables mergeable operation logs (*mlogs* for short) by managing operation logs in an in-memory mlog tree, which is based on a modified extent tree design. The mlog tree will merge contiguous or overlapping operation logs if possible and minimize its additional overhead on write operations. To control uploaded data in different cloud backups, SolFS proposes the mlog versioning mechanism to manage different versions of mlogs. This technique leverages extended file attributes [26] to integrate the inode versioning functionality for mlogs while ensuring backward compatibility. Additionally, SolFS persists different versions of mlogs to different versioned inodes, supporting

multiple cloud backup APPs to accurately manage their own mlogs for unuploaded file data. During delta synchronization, a cloud backup APP can traverse the file's versioned inodes to merge mlogs into an extent tree that indicates the modified data ranges (see Section 3.3). Based on these ranges, the APP can either directly upload the modified data or apply hashing (e.g., rolling checksum) to further reduce network traffic.

3.2 Per-file Mergeable Operation Logging

MLogging identifies old and new modified file data by logging each write's offset and length (i.e, *mlog*). The different write's mlogs can be merged and the offset and length of each mlog can be aligned to two different granularities: byte-aligned and page-aligned. This design enables SolFS to balance memory/storage costs with the identification granularity of modified data. In addition, because MLogging writes new data to the original file inode and stores mlogs in additional inodes, no file fragmentation is introduced, ensuring good sequential read performance.

3.2.1 Mergeable Operation Log Tree.

To prevent the rapid growth in memory and storage usage for mlogs, MLogging introduces an in-memory mlog tree to manage mlogs of the head versioned inode (see Section 3.2.2). Based on the the extent tree design, the mlog tree can merge mlogs with contiguous or overlapping modification ranges. Specifically, a new mlog is inserted into the tree using its offset as the key. During insertion of a new mlog, MLogging traverses the mlog tree to identify the contiguous or overlapping mlogs. If found, they are merged, and memory/storage space is released; otherwise, the new mlog is inserted into the tree.

Figure 4 demonstrates this mechanism. For example, request 3 `(25,10)` overlaps with an existing mlog `(20,7)` so that MLogging only updates the existing mlog to `(20,15)`. In this way, MLogging reduces the high storage overhead of storing new operation logs. For sequential writes or appends, they only update the last-accessed mlog so that the direct mlog tree insertion is not optimal (i.e., $O(logN)$ operations). Therefore, SolFS introduces an mlog pointer cache at the root of the mlog tree, which records the last-accessed mlog. This design reduces mlog updates for sequential writes and appends to $O(1)$ operations, further improving additional cost on write operations. For instance, in Figure 4, request 1 and 2 are two subsequent write requests. Request 1 does not hit the mlog cache, requiring a traversal of the mlog tree, while request 2 hits the cache, allowing direct updates to the target mlog.

Persistence with Compact Mlogs. In the MLogging design, an in-memory mlog tree is allocated to track file differences when a file is opened and is persisted to the disk when the modified file data is written back. As shown in Figure 4, the on-disk layout of the mlog tree is an array of mlogs, where

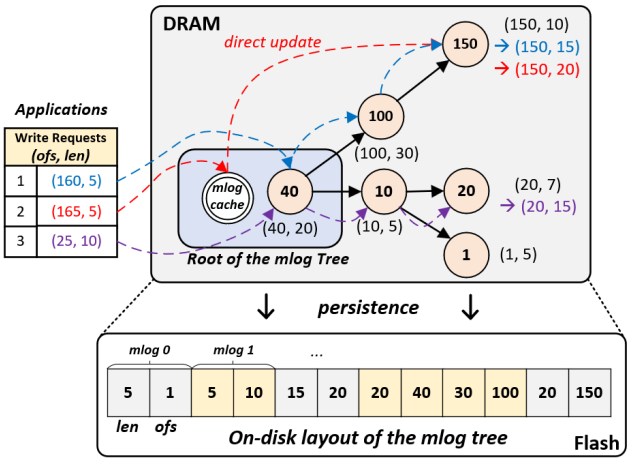


Figure 4: MLogging mechanism.

each mlog entry stores two integers—offset and length—both are aligned to a specific granularity (e.g., byte-level). Given the prevalence of small files and small writes (e.g., < 1MB) on mobile devices [8, 17, 19], the offset and length integers are designed to take 4 bytes each. In order to minimize the storage cost, MLogging persists the length first, followed by the offset, leveraging the highest bit of the length to support compact mlog design. As illustrated in Figure 5, a compact mlog is a compact form of an mlog entry, which embeds the offset into the unused bits of the length (i.e., storing both offset and length in four bytes). Specifically, the 0~15 bits are used to store the inline offset and the 16~30 bits are used to store the length. The highest bit of the length serves as an indicator of compact mlogs, because it is not used in the normal form of mlog entry (the maximum write buffer length is 2GB, requiring only 19 bits [27]). SolFS will reconstruct the mlog tree by scanning the on-disk mlog array, where the scanning process will dynamically skip to the next mlog based on the compact mlog indicator.

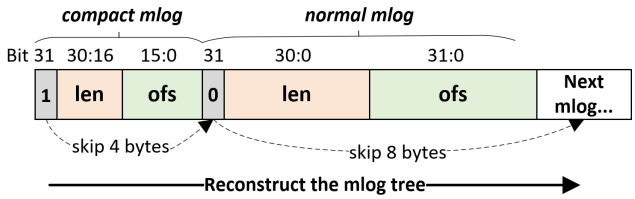


Figure 5: On-disk structure of normal and compact mlog.

Minimizing Memory Usage. A file may accumulate numerous mlogs after continuous random updates, and loading all of them into the in-memory mlog tree can significantly increase memory usage. To mitigate this issue, SolFS employs an on-demand loading design. During the open process, SolFS allocates an empty mlog tree rather than loading all mlogs from disk. All new mlogs generated during file updates are inserted into this tree. When persisting the mlog tree, SolFS

reads the existing mlogs from disk, merges them with the new mlogs in the in-memory mlog tree, and persists the updated mlog tree to disk. To minimize the additional cost on the I/O critical path of applications, SolFS uses a dedicated worker thread to handle this process asynchronously. When the file data is written back, the inode with a dirty mlog tree is sent to the worker thread, ensuring minimal impact on application performance.

Dynamic Granularity Alignment. SolFS allows the offset and length stored in mlogs to be dynamically aligned to different granularities. The main purpose of this design is to reduce the total number of mlogs by increasing the granularity from the byte-level to page-level, thereby lowering the memory and storage usage. In addition, this design helps SolFS avoid the integer overflow issue of the offset when updating very large files. Specifically, SolFS limits the number of in-memory mlogs per file (i.e., 5000 in SolFS). When more than this threshold, SolFS increases the granularity of the mlog tree. A new mlog tree is created and set to a higher granularity. Then, SolFS traverses mlogs in the low-granularity mlog tree, converts their offsets and lengths to the larger granularity and finally inserts them into the new mlog tree. While a larger granularity reduces memory and storage usage, it also decreases the resolution of file changes and increases network traffic during delta synchronization.

3.2.2 Mlog Versioning

Cloud backup APPs will continuously upload files to protect user data and thus a file may be uploaded multiple times. This necessitates SolFS to manage and classify file differences between any two cloud backups, and to support multiple cloud backup APPs on the same device. Inspired by classic versioning techniques, SolFS proposes the mlog versioning design to achieve this goal.

Versioned Inodes for Mlogs: Similar to prior works [7, 28, 29], this design introduces additional versioned inodes to the original file inode. However, unlike previous approaches, the versioned inodes store mlogs instead of file data, and the inode version is driven by each backup rather than write operations. Each versioned inode maintains its own set of mlogs, enabling SolFS to identify file modifications between any two versions by merging mlogs across different versioned inodes. The versioned inode includes four additional attributes: *ino_ver*, *ver_link*, *ino_flag* and *next_ino*. The *ino_ver* attribute represents the backup-driven inode version, while the *ver_link* attribute indicates the number of cloud backup APPs referencing this inode version. The *ino_flag* attribute tracks the state of this versioned inode, including its dirty state and the granularity of its persisted mlogs. The *next_ino* attribute is utilized to establish a connection between two inodes of consecutive versions, forming a chain of versioned inodes. To reach the design goal of backward compatibility, these four additional attributes are stored in the extended file attributes [26] of the inode, which ensures compatibility with the on-disk structures of the existing mobile file system. The versioned inodes shown in Figure 3 present the relationship between the original file inode and versioned inodes. The file inode and two versioned inodes (ver=3 and ver=2) are connected and indexed based on each inode's *next_ino*.

Backup-driven Versioning: Since a mobile device may install multiple cloud backup APPs (e.g., built-in and third-party), each with its individual backup times and progress, SolFS introduces backup-driven versioning to manage file versions across different backups and across different APPs. Specifically, when an APP backs up the latest data of a file, a new versioned inode (with a higher version number) is inserted at the head of the file's versioned inode chain by updating *next_ino*. A new mlog tree is also created to accommodate incoming mlogs since this backup, as shown in Figure 6. During the insertion of new versioned inode, the old mlog tree is persisted to the old head versioned inode and then releases its memory. In cases where the head versioned inode does not contain any mlogs, such as no modifications since the last backup, SolFS avoids inserting a new versioned inode to maintain a compact and concise versioned inode chain to reduce storage usage, such as the second backup in Figure 6. With backup-driven versioning, all file modified locations are recorded in a versioned inode chain. Given a start version and a target version, SolFS can traverse the corresponding versioned inodes in the chain to retrieve mlogs, which are used to calculate the final modified range of that file since the start version. In this way, SolFS ensures that each cloud backup APP can track its own file differences between any two backups without interference from other apps.

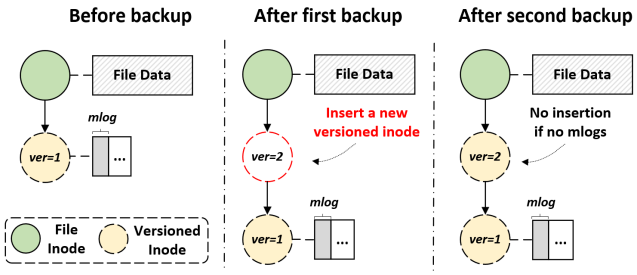


Figure 6: Backup-driven versioning.

3.3 Hash-free Delta Synchronization

In this section, we will introduce how SolFS achieves delta synchronization by enabling hash-free file difference identification through the collaborative version management and the extent tree that stores cumulative file modifications, as shown in Figure 7.

Collaborative Version Management: As discussed in Section 3.2.2, each backup is assigned a specific version number, which allows the cloud backup APP to manage file versions between the local and the cloud server. For example, after

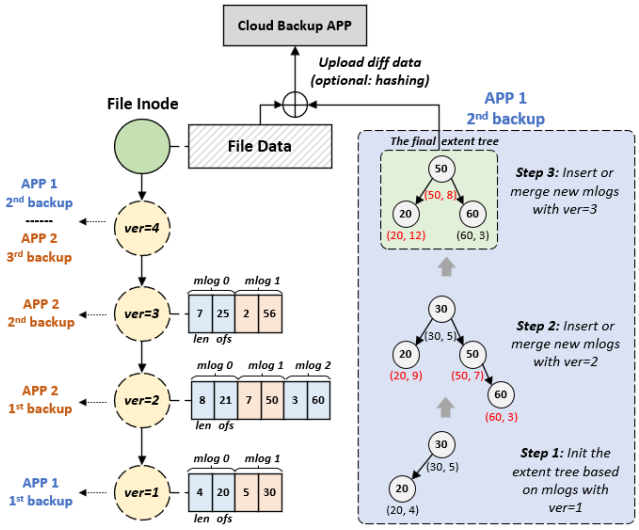


Figure 7: Delta Synchronization with Hash-free File Difference Identification.

backing up a file, the APP will obtain a version number from SolFS and synchronize this version number to the cloud server. The cloud server will maintain this version number for each file, and the version number is also stored in the extended file attributes. For the next backup, the APP receives the version number from the cloud server and tracks new file modifications from this version to the latest version. Cloud backup APPs can start their backup in any version of a file, but in the first backup, they will upload the entire file data.

We illustrate this collaborative version management in Figure 7. There are two APPs (APP 1 and APP 2) backing up the same file. Before any backups, the version of the original file data is zero (ver=0). After APP 1 performs the first backup, the first versioned inode (ver=1) is inserted and APP 1 obtains this inode's version number, indicating that the next backup of APP 1 starts from ver=1. Following this, APP 2 performs three consecutive backups, each with new modified data, thus increasing the inode version from ver=1 to ver=4. Note that the first backup of APP 2 will upload the entire file data as APP 2 does not maintain the version of this file before. Then, APP 1 performs its second backup based on its own version (ver=1), in which it traverses mlogs stored in the versioned inodes from ver=1 to ver=4 to identify cumulative file modifications to facilitate delta synchronization. Finally, both APP 1 and APP 2 will reach the versioned inode with ver=4, as the latest versioned inode (ver=4) does not contain mlogs. This collaborative version management also reduces the hash computation overhead on the cloud server, as it eliminates the need to generate and send a checksum list to APPs for hash comparison. By exchanging version information instead of checksum lists, network traffic is further minimized, particularly for large files where checksum lists can be substantial.

Upload with the Extent Tree: Given an original version,

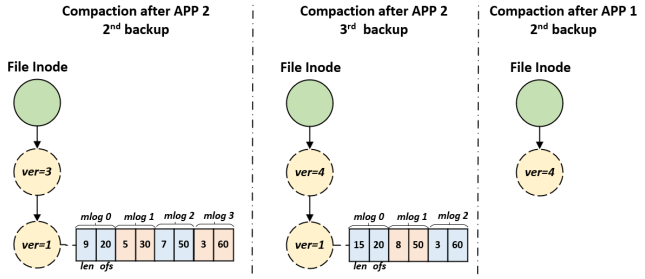


Figure 8: Versioned Inode Compaction.

SolFS enables the APP to track mlogs from that version to the latest version. SolFS employs an extent tree to merge mlogs stored across different versioned inodes to calculate the cumulative file modified range of the data. Based on this range, the APP can directly upload the corresponding file data to achieve hash-free delta synchronization. Optionally, the APP could choose to perform hashing on the specified data range to mitigate hash overhead of identifying file differences and further reduce network traffic. Note that this approach also avoids the overhead of reading the entire file data.

We demonstrate this design in the second backup of APP 1 in Figure 7. SolFS first initializes the extent tree based on mlogs from the version held by APP1 (ver=1). It then scans the versioned inode chain from ver=1 to ver=4, merging their mlogs to produce a final extent tree representing the cumulative modified range. If the mlog granularity is byte-level/page-level, the modified range includes bytes/pages 20–31, 50–57, and 60–62. APP 1 could use this location information to upload the corresponding file data to the cloud. The mlogs stored in different versioned inodes may have varying granularities, but during merging, all mlogs are converted into byte-level granularity for consistency.

Versioned Inode Compaction: Multiple backups can lead to redundant versioned inodes in the chain, as they may no longer be indexed or used by cloud backup apps, wasting storage space. To address this issue, SolFS employs a reference counter (i.e., *ver_link*) for each versioned inode to track the number of cloud backup APPs referencing it. Based on the example shown in Figure 7, after APP 2 performs three backups and before the second backup of APP 1, the reference counts for the four versioned inodes are one (ver=1), zero (ver=2), zero (ver=3) and one (ver=4). The versioned inodes with ver=2 and ver=3 are redundant as no APPs use them, allowing SolFS to compact them to save storage space. Specifically, SolFS compacts redundant versioned inodes by merging their mlogs into the adjacent versioned inode with a lower version number and positive reference counter, and then deletes redundant versioned inodes. After the second backup of APP 1, the reference count of versioned inode with ver=4 increases to two since both APP 1 and APP 2 point to it. At this moment, SolFS cannot find an adjacent versioned inode for compaction as the reference count of ver=1 inode becomes zero, which indicates versioned inodes lower than

the latest version are useless for subsequent backups.

We illustrate the proposed compaction process in Figure 8. After the second backup of APP 2, all mlogs stored in ver=2 inode are merged with mlogs stored the ver=1 inode, because the ver=2 inode is no longer used. After the second backup of APP 1, versioned inodes except for the head versioned inode are useless so that SolFS removes them to release resources. Finally, when all APPs reach the latest version, all versioned inodes with zero link are deleted. SolFS collects the reference count of each versioned inode during the delta synchronization process and compacts them by an asynchronous worker thread mentioned in *Minimizing Memory Usage* in Section 3.2.1.

4 Implementation Discussion

In this section, we will introduce the implementation details of the SolFS prototype on F2FS [21], which is the most popular file system on mobile devices. In addition, how to ensure version consistency and how to create an optimized cloud backup APP based on SolFS are also discussed in the following subsections.

4.1 File System Implementation

When opening a file in SolFS, an empty in-memory mlog tree is created using *kmem_cache_alloc* to reduce allocation cost and it is stored in the F2FS's inode info. During page writebacks, the inode with a dirty mlog tree is passed to a worker thread for asynchronous persistence via a shared list. As a result, SolFS only introduces minimal overhead to file opening and closing. SolFS does not change the original read and write processes of the existing mobile file system, but instead intercepts file operations that modify file data, such as write operations. Additionally, SolFS also supports *fallocate*, *truncate* and *mmap* with PROT_WRITE by inserting a new mlog for the data range specified by these functions' parameters. For *rename*, SolFS marks the newly renamed file as entirely changed. The asynchronous worker thread is primarily responsible for persisting mlog trees. Specifically, it wakes up periodically to process dirty mlog trees from inodes. The dirty mlog tree is converted into an mlog array as discussed in Section 3.2.1. SolFS then copies this mlog array to the pages of the corresponding head versioned inode, and marks all pages as dirty for asynchronous writebacks to complete the persistence process.

4.2 Version Consistency

When the system crash occurs, the consistency of file data is protected by the existing mobile file system. However, SolFS has to ensure version consistency in case of identifying inconsistent modified data during delta synchronization. This is critical because the persistence of newly modified file data and its corresponding mlogs is not atomic. An unexpected crash could disrupt their consistency, resulting in mismatches between local and cloud file data. To address this issue, SolFS introduces an *all-or-nothing* mechanism that leverages F2FS's write-ordering guarantees to maintain version consistency between the version number (i.e., *ino_ver*) and file data. For inode chain operations, the F2FS checkpointing is employed to ensure data consistency.

All or Nothing. Since system crashes are rare events and abandoning previous version controls does not affect the consistency of the file data itself, SolFS introduces an *all-or-nothing* mechanism to ensure version consistency, as shown in Figure 9. Specifically, SolFS employs a special dirty flag (i.e., *ino_flag*) in the file inode to indicate whether the mlog tree has been modified. When dirty data pages of the file inode are written back to the disk, SolFS marks the *ino_flag* as dirty (=1) and sends the corresponding inode to the worker thread. This flag is clear after successfully persisting the mlog tree. The write order of F2FS ensures that data writebacks occur before metadata (e.g., inode) writebacks and the *ino_flag* is written back to the disk along with the inode. If data pages are written back successfully but the system crashes before the file inode is persisted, the written blocks are released, and the inode reverts to its pre-write state. If the crash occurs after metadata writeback but before mlog writeback, the *ino_flag* remains dirty, allowing SolFS to detect inconsistencies between file data and mlogs. In this scenario, SolFS removes the versioned inode chain and reinitializes the file version starting from the version higher than the one at the time of the crashing. If the APP provides a stale version of the old versioned inode chain, SolFS cannot locate this version in the new versioned inode chain, as it is lower than the new version. Consequently, the APP is required to upload the entire file data and maintain a new version.

Checkpointing Inode Chain Operations. SolFS leverages F2FS's checkpointing mechanism to ensure the consistency of inode chain operations. Specifically, SolFS temporarily disables checkpointing while updating *next_ino* during inode chain operations and re-enables it once the operations are complete. This prevents inconsistent inode connections in the event of a system crash. If a crash occurs, SolFS relies on F2FS's roll-back/roll-forward recovery process to restore the file system to the last checkpointed state.

4.3 Cloud Backup APP Optimization

SolFS introduces three new ioctl interfaces for cloud backup APPs: *delta_open*, *delta_close* and *delta_getdiff*. Given a file to be backed up, the cloud backup APP first opens the file to obtain its file descriptor (FD), and retrieves the file's version number from the cloud server. With this FD, the APP can invoke *delta_open* to find the corresponding file inode and register a context to get the file difference. This function is also used to block write operations and flush all dirty

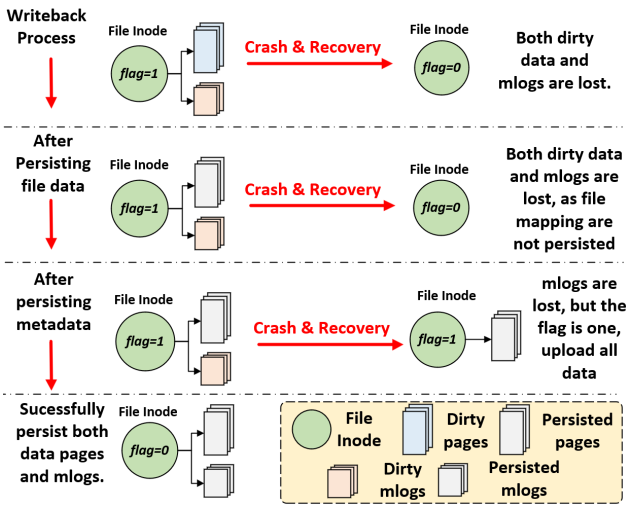


Figure 9: Version Consistency.

pages and mlogs of the file. Then, the APP leverages the file descriptor and the version number as input for *delta_getdiff*. As the number of returned modified ranges (mlog) is not certain, the APP can call *delta_getdiff* multiple times to recursively obtain all mlogs. To support recursive access, the context created by *delta_open* will record the progress of getting the file difference. This interface returns the cumulative file modified range and the latest version number, while also inserting a new versioned inode into the chain. The APP uses the cumulative modified range to access the corresponding file data. After identifying all modified ranges, the APP can invoke *delta_close* to release the resources of the context and unblock write operations.

To further reduce network traffic, we experimentally introduce a simple selective hashing approach for the modified range of data. Specifically, the cloud backup APP prioritizes the computational cost on mobile devices so that it chooses to directly upload modified file data, except for mlogs encompassing the entire file data. For this type of mlog, the entire file data is conducted with a rolling checksum, similar to WebR2sync+. In the scenario where the version inputted by the APP does not match any version of versioned inodes (i.e., version inconsistency), *delta_getdiff* will return a special modified range that covers the entire file data. SolFS is similar to existing file systems and does not include optimizations for file metadata backup. The backup APP can use the common fstat()/getxattr() interfaces to retrieve available metadata information and synchronize it to the cloud server.

5 Evaluation

In this section, we present a comprehensive evaluation of the proposed SolFS in comparison to the state-of-the-art delta synchronization designs, including *HashSync*, *Rsync* and *WebR2sync+*. *HashSync* refers to the typical design of delta synchronization [11] that separates a file into multiple chunks and uses MD5 as the hashing algorithm to identify modified data chunks. Rsync and WebR2sync+ employ rolling checksum in the APP side and server side, respectively, allowing them to detect byte-level data differences. We do not compare with *NetSync* because it is not open-sourced. The existing solutions do not support the Android system and thus we implement and cross-compile them to the mobile platform based on C++ and NDK [14]. The server side is implemented based on cpp-httplib [41] which receives requests from APPs and supports different cloud synchronization based on the parameters of HTTP POST requests.

Our evaluations are conducted on a Pixel 8 device based on Android 14 with Linux kernel version 5.15, and the cloud server is running on an Ubuntu 20.04 platform with a 24-core Intel i9 14900k and 32GB of memory. We assume that users upload files using only one cloud backup APP, as this is the most widely used scenario. In the following subsections, we will first introduce some micro-benchmarks to illustrate how SolFS achieves better performance. Then, the performance and overhead of SolFS on real applications will be discussed. Lastly, we also conduct some experiments to discuss the overhead of inode chain operations.

5.1 Cloud Synchronization Efficiency

In this section, we quantitatively measure the cloud synchronization efficiency of SolFS with a series of micro-benchmarks, including cloud sync time, network traffic and latency breakdown. The network bandwidth is set to ~100 Mbps, the chunk size is set to 2048 bytes and the granularity of mlog is byte-level. We use 10MB files and up to 1KB I/O size, considering small files and I/Os on mobile devices. For these experiments, we first create a 10MB file and upload it to the server. Then, we modify this file in different ways: random update, sequential update and insertion, where insertion means moving back all the file data behind the insertion location by re-writing the whole file. In our experiments, we insert data into the position of 5MB of the file. In addition, we further evaluate the effect of modification sizes. The modification size indicates the total modified bytes to that file and when the update/insertion size is larger than 1KB, this modification will be separated into multiple sub-modifications (each with 1KB). Finally, this file is uploaded again to the server and we record the relevant experimental results.

5.1.1 Cloud Sync Time and Network Traffic

We present the total sync time and network traffic of SolFS and existing solutions in Figure 10 and Figure 11. SolFS significantly outperforms hash-based solutions in both random and sequential updates. Specifically, SolFS reduces the sync time by 88.8% on average in random updates, since it identifies modified data based on offsets and lengths, mitigating the

overhead of hash computation and data reading. In addition, SolFS generates significantly less network traffic, averaging only 12.3% of the traffic when updating 1MB of data.

This efficiency stems from two key factors. First, the three hash-based approaches require transmitting a checksum list, whose size grows with the file size (e.g., 120KB for a 10MB file). Second, since random updates are not aligned with the chunk size, these updates often affect neighboring chunks, resulting in more modified data (e.g., over 2MB of data is modified when randomly updating 1MB of data). SolFS avoids these issues by precisely uploading only the modified data at the byte level. The performance of a sequential update also has the similar observation to that of random updates due to the same benefits of SolFS. For data insertion, both Rsync and WebR2sync+ support rolling checksum, enabling them to upload only the inserted data (around 1MB when inserting 1MB data). HashSync cannot identify the inserted data and has to upload all data following the insertion point, resulting in approximately 5x more network traffic (i.e., 5MB) and longer sync times compared to the other three approaches. As discussed in Section 4.3, SolFS can collaborate with the rolling checksum approach, achieving performance comparable to WebR2sync+.

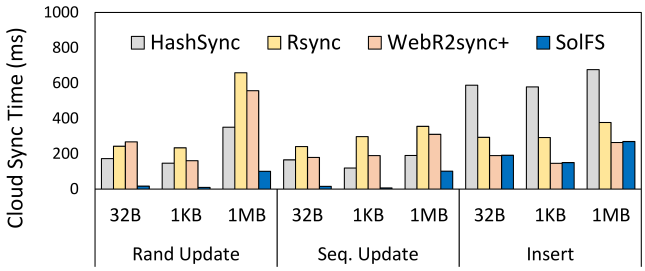


Figure 10: Cloud sync time on different types of file modifications.

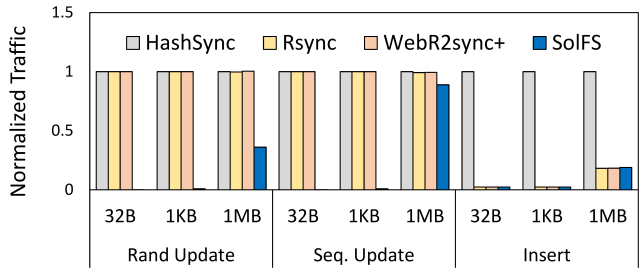


Figure 11: Normalized network traffic on different types of file modifications. HashSync serves as the baseline.

5.1.2 Sync Time Breakdown

To analyze the contribution of each part to sync time, we illustrate the sync time breakdown for a 1MB random update in Figure 12. SolFS outperforms other approaches by significantly reducing hash computation overhead on both the client and server sides (92% on average). For instance, Rsync exhibits lower server-side latency, while WebR2sync+ achieves lower client-side latency due to their differing design goals. SolFS, however, achieves lower latencies on both sides. Due to its low software overhead, network bandwidth becomes the primary factor limiting SolFS's performance. On a 50Mbps network, the client and server sides contribute approximately 21.8% (HashSync), 49.3% (Rsync), 42.2% (WebR2sync+), and 10.8% (SolFS) of the total sync time. On a 200Mbps network, these contributions increase to 52.7% (HashSync), 79.5% (Rsync), 74.5% (WebR2sync+), and 32.7% (SolFS). As network bandwidth grows, computation on both the client and server sides leads to significant software overhead. SolFS mitigates this issue by minimizing hash computation.

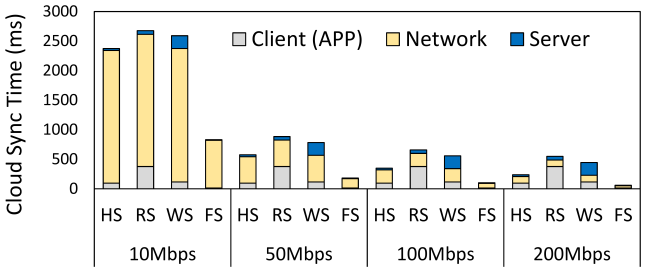


Figure 12: The sync time breakdown over different network bandwidths. HS, RS, WS, FS indicate HashSync, Rsync, WebR2sync+, SolFS, respectively.

5.2 Real Mobile Application Workloads.

5.2.1 Cloud Backup Performance on Real Workloads.

To evaluate the overall cloud backup performance, we dump file data from four popular applications, including Facebook with 220MB data, Twitter with 112MB data, Capcut with 220MB data and Dropbox with 843MB data. After a period of time (i.e., 20 minutes), we dump their file data again and store them in the server side. In this period, we play each application for 5 minutes: for Facebook and Twitter, we browse and pull new content; for Capcut we edit existing videos by adding effects, filters and text; for Dropbox, we edit a Word documents by adding new text, a PPT slide by adding a new page with image and text, and write several numbers to an Excel file. During playing applications, we collect file update traces from these four applications, including file names, file sizes, file update offsets and lengths. Then, we can compare SolFS with existing solutions based on the consistent file data and trace. We perform file synchronization using SolFS and other solutions and record their sync time.

The experimental results are shown in Figure 13. SolFS significantly outperforms existing solutions on mobile application workloads, by improving 71% of sync time on average. For example, SolFS reduces the HashSync's average sync time over these four APPs from 89s to 29s. The main reason

is that random writes dominate the mobile platform, such as many SQLite database updates on Facebook and Twitter. We observe that data insertions are rare on mobile workloads and applications usually operate on binary data so that the performance of HashSync is better than Rsync and WebR2sync+ on mobile workloads. By avoiding hashing, SolFS has a greater performance than HashSync. In addition, the SolFS design helps reduces by around 70% of CPU usage on average of the four APPs during delta synchronization.

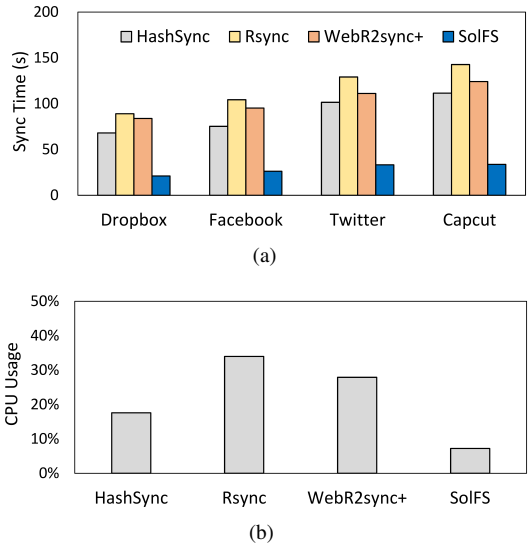


Figure 13: (a) cloud sync time over different applications; (b) CPU usage on average of the four commonly-used APPs.

5.2.2 Overhead Analysis on Real Workloads

In this section, we focus on the potential overhead of SolFS, including the I/O performance, and additional memory and storage usage compared with the original mobile file system (i.e., F2FS). To comprehensively evaluate the performance of SolFS on different file operations, we choose to replay long-duration file operations traces collected in real mobile devices. Hence, nine open sourced file operation traces [25] are used in our evaluation, where each trace represents a specific application running for two hours. The traces include nine applications, including Candycrush (CC), Chrome (CH), Earth (EA), Facebook (FB), Map (MA), TikTok (TT), Twitter (TW), Youtube (YT) and Zombie (ZB).

Read and Write Overhead. We evaluate the read and write performance based on the trace replay time. Before running each trace, we clear the page and inode caches, and record the running time of each trace. As shown in Figure 14(a), SolFS shows a comparable I/O performance (difference < 1.5%) with the original F2FS. On average, the I/O performance of SolFS is around 99.2%. During trace replay, the average CPU usage of F2FS and SolFS is 8.3% and 8.5%, respectively, representing the overhead when MLogging is disabled or enabled during normal operations. The slight increase in SolFS's CPU usage occurs because SolFS inserts a small mlog entry (4 or 8 bytes) into the mlog tree during write operations. For write operations that hit the mlog cache (e.g., sequential writes and appends), SolFS uses a fast path to update the mlog at $O(1)$ cost; for cache misses, SolFS needs to traverse the in-memory mlog tree. As SolFS does not load mlogs from the existing versioned inode to the mlog tree , the number of a file's mlogs is relatively small, which reduces the searching cost. SolFS does not change the read process of the F2FS so that the read performance is the same. The additional overhead of file open is also low, because the on-demand mlog loading design avoids reading mlogs from disk synchronously in the open operation.

Memory and Storage Overhead. SolFS consumes additional memory and storage space to maintain the mlog tree and mlogs while a file is open. First, we measure the additional memory usage by running traces from different APPs. To evaluate the worst-case scenario, we use a byte-level mlog tree and retain mlogs even after the file is closed. As shown in Figure 14(b), the additional memory usage after running nine traces (simulating 18 hours of app runtime) is 470KB. This is because many operations, such as sequential writes, appends, or updates to the same location, only require fewer mlogs. Additionally, the on-demand loading design reduces memory usage, as SolFS does not load all mlogs from disk into memory and only maintains an in-memory mlog tree for new mlogs. For storage overhead, SolFS further reduces memory usage through its compact mlog design. Given the prevalence of small writes on mobile devices, mlogs are frequently compacted, reducing storage space by 33% compared to memory usage. Note that increasing the granularity of mlogs can further decrease memory usage. Memory allocated for the mlog tree and mlogs are released once the file is closed, further minimizing SolFS's memory footprint. In summary, the memory and storage overhead of SolFS are acceptable for mobile devices.

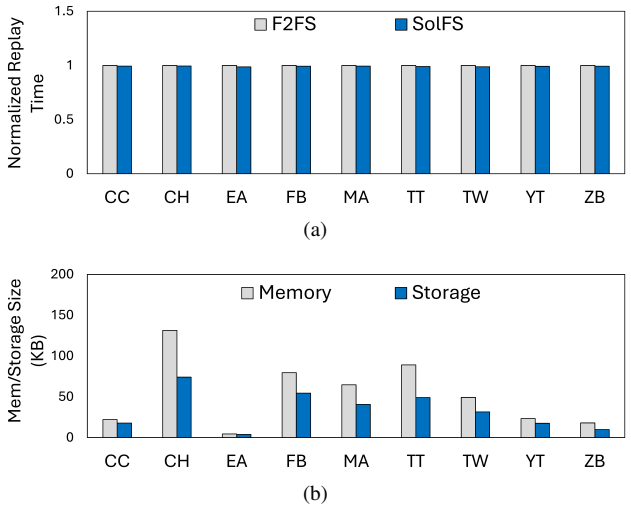


Figure 14: (a) normalized I/O performance; (b) additional memory and storage overhead.

5.3 Discussion about Inode Chain Operations

Depth of Version Inode Chain. As SolFS supports multiple cloud backup APPs based on a versioned inode chain, the running time of searching file differences could be affected by the depth of the versioned inode chain. To evaluate the effect of the depth, we create a 10MB file and randomly update total 2MB of data. Given a depth of X, each level of versioned inode will be updated by 2MB/X of data. After clearing page cache and inode cache, we use the *delta_getdiff* interface to search for file differences and record the searching time. As presented in Figure 15(a), with the growth of the versioned inode chain, the search time increases from 1.67ms to 7.6ms due to the additional I/O overhead. Even with the depth = 10, this cost is still significantly lower than hashing. In addition, users generally do not use many cloud backup APPs at the same time, and thus the depth of the versioned inode chain can be kept small.

Impact of Mlog Quantity. In SolFS, the number of mlogs in an mlog tree primarily affects the overhead of two operations: 1) converting the mlog tree from small granularity (byte-level) to large granularity (page-level), and 2) merging the in-memory mlog tree with the on-disk mlog tree. First, for mlog tree conversion, Figure 15(b) shows that the conversion time increases with the number of mlogs, ranging from 0.95ms for 1K mlogs to 10ms for 10K mlogs. Second, when persisting an in-memory mlog tree, SolFS loads and merges on-disk mlogs into the in-memory mlog tree before writing it back. This overhead also scales with the number of mlogs. In our evaluation, loading and merging a large mlog tree (10K mlogs) introduces around 30ms of latency, including read I/O and in-memory merging time. Since scenarios with intensive random updates to the same file are relatively rare, the number of mlogs typically remains small (e.g., < 100). Therefore, both mlog tree conversion and merging operations introduce minimal overhead on applications.

Granularity and Memory Overhead. To evaluate the impact of different mlog granularities, we conduct sequential and random write experiments by writing 2MB of data to a 10MB file using a 1KB buffer size. Initially, with byte-level granularity, SolFS consumes 0.133KB and 102.6KB of additional memory for sequential and random writes, respectively. When increasing the granularity to page-level, memory usage for random writes decreases by approximately 80%, to 20.35KB. The larger granularity enhances opportunities for mlog merging, but as expense of more network traffic during delta synchronization.

Inode Chain Compaction Overhead. The overhead of inode chain compaction includes two parts: mlog merging and inode re-connection. The first part depends on the number of mlogs, which is discussed in *Granularity Conversion*. Inserting 1K mlogs takes around 0.95ms, which is relatively low-cost. For the second part, the inode re-connection is similar to the rename operation, i.e., updating two inodes' metadata, which consumes around 2 ms. Inode chain compaction as a periodic asynchronous task, its overhead is acceptable for mobile devices.

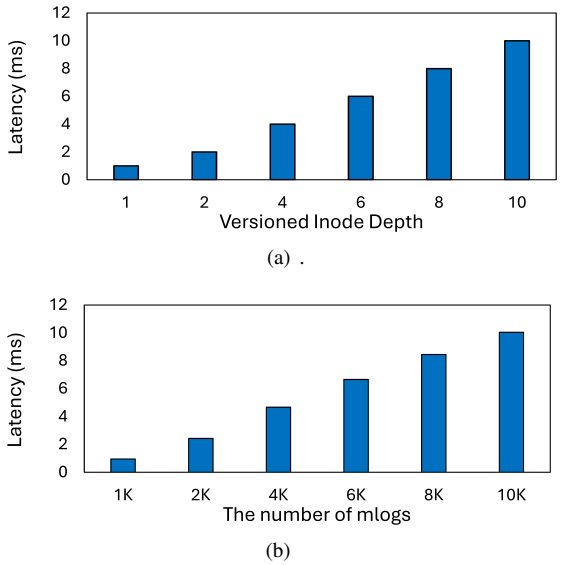


Figure 15: (a) searching time over versioned inode depths and (b) conversion time over the different numbers of mlogs.

6 Related Works

Delta Synchronization in Cloud Backup. Many previous works discuss how to use delta synchronization to improve the cloud backup efficiency [5, 6, 10, 11, 24, 34, 36, 38]. Drago [10, 11] introduced five critical capabilities of cloud backup: chunking, bundling, deduplication, delta-encoding, and compression. Tridgell [34] introduces rsync algorithm that uses rolling checksum to identify modified data. Cui [9] considered the backup efficiency of mobile cloud backup applications and proposed QuickSync to reduce unnecessary sync traffic. Xiao [38] moves the heavy hash computation and comparison from the client side to the server side. Xia [36] introduces content-defined chunking to improve the hash comparison during delta synchronization. Wu [35] designs a faster delta synchronization mechanism in encrypted cloud storage. In addition to the protocol improvement of delta synchronization, some researchers propose lightweight algorithms to improve their computational efficiency [22, 37, 44] However, these efforts still require intensive computation on hash calculation and comparison, significantly increasing the CPU usage and power consumption on resource-limited mobile devices. SolFS mitigates hash computation in delta synchronization based on the collaborative design between cloud backup APPs and file system.

File Snapshotting and Versioning. Many file systems implement file snapshotting or versioning mechanisms. Most of file systems support snapshotting or versioning based on a copy-on-write manner [1, 4, 28–30, 33], in which newly modi-

fied data is written to a new block while the old data is kept accessible to the snapshot file. file fragmentation because file data is scattered in multiple inodes, which reduces the sequential read performance. BetrFS [42] mitigates the fragmentation issue in snapshot file systems by caching random writes. In addition to copy-on-write, data logging is another common approach for snapshotting or versioning mechanisms. NOVA-Fortis [40] implements snapshotting by inserting private data logs into each inode. WaybackFS [7] implements versioning by storing old modified data as undo logs in an auxiliary inode. However, these efforts require additional storage space to store modified data because they need to keep track of the file data for a specific snapshot/version of the file. Therefore, SolFS does not adopt these designs to identify old and new modified data. It stores operation logs only rather than file data, and thus significantly reduces the storage usage.

7 Conclusions

Cloud backup is high-cost on mobile devices. SolFS aims to improve its computation overhead of delta synchronization on mobile devices. It introduces MLogging, which allows each file to manage its write operation logs (i.e., offset and length) in an extent tree, and the operation log persistence and versioning mechanism. Combining these techniques together, SolFS achieves hash-free difference identification while minimizing the additional cost to the original system. We evaluated SolFS on commercial smartphones and the results show that it can significantly reduce the computational overhead of both APP-side or server-side by over 90% on average and the total cloud synchronization time by over 88% when files are updated.

References

[1] OpenZFS on Linux. https://zfsonlinux.org/, 2019. [Online; accessed 13-September-2022].

[2] Android. Power Profiler. https://developer.android.com/studio/profile/power-profiler, 2024. [Online; accessed 13-May-2024].

[3] Enrico Bocchi, Idilio Drago, and Marco Mellia. Personal cloud storage: Usage, performance and impact of terminals. In *2015 IEEE 4th International Conference on Cloud Networking (CloudNet)*, pages 106–111. IEEE, 2015.

[4] Bolosky, William J and Corbin, Scott and Goebel, David and Douceur, John R. Single instance storage in Windows 2000. In *Proceedings of the 4th USENIX Windows Systems Symposium*, pages 13–24. Seattle, WA, 2000.

[5] Fei Chen, Zhipeng Li, Changkun Jiang, Tao Xiang, and Yuanyuan Yang. Cloud object storage synchronization: Design, analysis, and implementation. *IEEE Transactions on Parallel and Distributed Systems*, 33(12):4295–4310, 2022.

[6] Cheng-Kang Chu, Wen-Tao Zhu, Jin Han, Joseph K Liu, Jia Xu, and Jianying Zhou. Security concerns in popular cloud storage services. *IEEE Pervasive Computing*, 12(4):50–57, 2013.

[7] Brian Cornell, Peter A Dinda, and Fabián E Bustamante. Wayback: A user-level versioning file system for linux. In *Proceedings of Usenix Annual Technical Conference, FREENIX Track*, pages 19–28, 2004.

[8] Jace Courville and Feng Chen. Understanding storage i/o behaviors of mobile applications. In *2016 32nd Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–11. IEEE, 2016.

[9] Cui, Yong and Lai, Zeqi and Wang, Xin and Dai, Ningwei. Quicksync: Improving synchronization efficiency for mobile cloud storage services. *IEEE Transactions on Mobile Computing*, 16(12):3513–3526, 2017.

[10] Idilio Drago, Enrico Bocchi, Marco Mellia, Herman Slatman, and Aiko Pras. Benchmarking personal cloud storage. In *Proceedings of the 2013 Conference on Internet Measurement Conference*, pages 205–212, 2013.

[11] Idilio Drago, Marco Mellia, Maurizio M. Munafo, Anna Sperotto, Ramin Sadre, and Aiko Pras. Inside dropbox: understanding personal cloud storage services. In *Proceedings of the 2012 internet measurement conference*, pages 481–494, 2012.

[12] Dropbox. Content Hash. https://www.dropbox.com/developers/reference/content-hash, 2023. [Online; accessed 4-April-2023].

[13] Exploding Topics. How Many People Own Smartphones? https://explodingtopics.com/blog/smartphone-stats, 2023. [Online; accessed 13-May-2024].

[14] Google LLC. Android NDK. https://developer.android.com/ndk, 2024. [Online; accessed 13-Jun-2024].

[15] Sangwook Shane Hahn, Sungjin Lee, Cheng Ji, Li-Pin Chang, Inhyuk Yee, Liang Shi, Chun Jason Xue, and Jihong Kim. Improving file system performance of mobile storage systems using a decoupled defragmenter. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 759–771, 2017.

[16] Matthew Halpern, Yuhao Zhu, and Vijay Janapa Reddi. Mobile cpu's rise to power: Quantifying the impact of generational mobile cpu design trends on performance,

energy, and user satisfaction. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 64–76. IEEE, 2016.

[17] Sooman Jeong, Kisung Lee, Seongjin Lee, Seoungbum Son, and Youjip Won. I/O stack optimization for smartphones. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 309–320, San Jose, CA, June 2013. USENIX Association.

[18] Cheng Ji, Li-Pin Chang, Sangwook Shane Hahn, Sungjin Lee, Riwei Pan, Liang Shi, Jihong Kim, and Chun Jason Xue. File fragmentation in mobile devices: Measurement, evaluation, and treatment. *IEEE Transactions on Mobile Computing*, 18(9):2062–2076, 2018.

[19] Cheng Ji, Riwei Pan, Li-Pin Chang, Liang Shi, Zongwei Zhu, Yu Liang, Tei-Wei Kuo, and Chun Jason Xue. Inspection and characterization of app file usage in mobile devices. *ACM Transactions on Storage (TOS)*, 16(4):1–25, 2020.

[20] Dohyun Kim, Kwangwon Min, Joontaek Oh, and Youjip Won. ScaleXFS: Getting scalability of XFS back on the ring. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*, pages 329–344, Santa Clara, CA, February 2022. USENIX Association.

[21] Changman Lee, Dongho Sim, Jooyoung Hwang, and Sangyeun Cho. {F2FS}: A new file system for flash storage. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 273–286, 2015.

[22] Giwon Lee, Haneul Ko, and Sangheon Pack. An efficient delta synchronization algorithm for mobile cloud storage applications. *IEEE Transactions on Services Computing*, 10(3):341–351, 2015.

[23] Zhenhua Li, Cheng Jin, Tianyin Xu, Christo Wilson, Yao Liu, Linsong Cheng, Yunhao Liu, Yafei Dai, and Zhi-Li Zhang. Towards network-level efficiency for cloud storage services. In *Proceedings of the 2014 Conference on Internet Measurement Conference*, pages 115–128, 2014.

[24] Zhenhua Li, Christo Wilson, Zhefu Jiang, Yao Liu, Ben Y Zhao, Cheng Jin, Zhi-Li Zhang, and Yafei Dai. Efficient batched synchronization in dropbox-like cloud storage services. In *ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing*, pages 307–327. Springer, 2013.

[25] Yu Liang, Riwei Pan, Tianyu Ren, Yufei Cui, Rachata Ausavarungnirun, Xianzhang Chen, Changlong Li, Tei-Wei Kuo, and Chun Jason Xue. CacheSifter: Sifting cache files for boosted mobile performance and lifetime. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*, pages 445–459, Santa Clara, CA, February 2022. USENIX Association.

[26] Linux Manual Page. Xattr - Extended attributes. https://man7.org/linux/man-pages/man7/xattr.7.html, 2023. [Online; accessed 13-September-2023].

[27] Linux Manual Page. Write(2). https://man7.org/linux/man-pages/man2/write.2.html, 2024. [Online; accessed 13-May-2024].

[28] Kiran-Kumar Muniswamy-Reddy, Charles P Wright, Andrew Himmer, and Erez Zadok. A versatile and user-oriented versioning file system. In *FAST*, volume 4, pages 115–128, 2004.

[29] Zachary Peterson and Randal Burns. Ext3cow: A time-shifting file system for regulatory compliance. *ACM Transactions on Storage (TOS)*, 1(2):190–212, 2005.

[30] Ohad Rodeh, Josef Bacik, and Chris Mason. Btrfs: The linux b-tree filesystem. *ACM Transactions on Storage (TOS)*, 9(3):1–32, 2013.

[31] Nikolai Samteladze and Ken Christensen. Delta: Delta encoding for less traffic for apps. In *37th Annual IEEE Conference on Local Computer Networks*, pages 212–215. IEEE, 2012.

[32] Statista. Number of smartphones sold to end users worldwide from 2007 to 2023. https://www.statista.com/statistics/263437/global-smartphone-sales-to-end-users-\since-2007/, 2023. [Online; accessed 13-December-2024].

[33] Adam Sweeney, Doug Doucette, Wei Hu, Curtis Anderson, Mike Nishimoto, and Geoff Peck. Scalability in the xfs file system. In *USENIX Annual Technical Conference*, volume 15, 1996.

[34] Tridgell, Andrew and Mackerras, Paul and others. The rsync algorithm. *Joint Computer Science Technical Report Series, The Australian National University*, TR-CS-96-05, 1996.

[35] Suzhen Wu, Zhanhong Tu, Yuxuan Zhou, Zuocheng Wang, Zhirong Shen, Wei Chen, Wei Wang, Weichun Wang, and Bo Mao. Fastsync: a fast delta sync scheme for encrypted cloud storage in high-bandwidth network environments. *ACM Transactions on Storage*, 19(4):1–22, 2023.

[36] Wen Xia, Can Wei, Zhenhua Li, Xuan Wang, and Xiangyu Zou. Netsync: A network adaptive and deduplication-inspired delta synchronization approach

for cloud storage services. *IEEE Transactions on Parallel and Distributed Systems*, 33(10):2554–2570, 2022.

[37] Wen Xia, Yukun Zhou, Hong Jiang, Dan Feng, Yu Hua, Yuchong Hu, Qing Liu, and Yucheng Zhang. {FastCDC}: A fast and efficient {Content-Defined} chunking approach for data deduplication. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 101–114, 2016.

[38] He Xiao, Zhenhua Li, Ennan Zhai, Tianyin Xu, Yang Li, Yunhao Liu, Quanlu Zhang, and Yao Liu. Towards web-based delta synchronization for cloud storage services. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*, pages 155–168, 2018.

[39] Weijun Xiao, Yinan Liu, Qing Yang, Jin Ren, and Changsheng Xie. Implementation and performance evaluation of two snapshot methods on iscsi target storages. In *Proc. of NASA/IEEE Conference on Mass Storage Systems and Technologies*. Citeseer, 2006.

[40] Jian Xu, Lu Zhang, Amirsaman Memaripour, Akshatha Gangadharaiah, Amit Borase, Tamires Brito Da Silva, Steven Swanson, and Andy Rudoff. Nova-fortis: A fault-tolerant non-volatile main memory file system. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 478–496, 2017.

[41] Yhirose. cpp-httplib. https://github.com/yhirose/cpp-httplib, 2024. [Online; accessed 13-Jun-2024].

[42] Yang Zhan, Alexander Conway, Yizheng Jiao, Nirjhar Mukherjee, Ian Groombridge, Michael A Bender, Martin Farach-Colton, William Jannen, Rob Johnson, Donald E Porter, et al. How to copy files. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 75–89, 2020.

[43] Tiangmeng Zhang, Renhui Chen, Zijing Li, Congming Gao, Chengke Wang, and Jiwu Shu. Design and implementation of deduplication on f2fs. *ACM Transactions on Storage*, 2024.

[44] Yucheng Zhang, Hong Jiang, Dan Feng, Wen Xia, Min Fu, Fangting Huang, and Yukun Zhou. Ae: An asymmetric extremum content defined chunking algorithm for fast and bandwidth-efficient data deduplication. In *2015 IEEE Conference on Computer Communications (INFOCOM)*, pages 1337–1345. IEEE, 2015.

Z-LFS: A Zoned Namespace-tailored Log-structured File System for Commodity Small-zone ZNS SSDs

Inhwi Hwang
Seoul National University

Sangjin Lee
Chung-Ang University

Sunggon Kim
Seoul National University of Science and Technology

Hyeonsang Eom
Seoul National University

Yongseok Son*
Chung-Ang University

Abstract

This paper presents a novel zoned namespace (ZNS) tailored log-structured file system (LFS) called `Z-LFS` for commodity small-zone ZNS SSDs. Specifically, `Z-LFS` first enables append-only updates on metadata while leveraging the unique metadata characteristic of LFS on ZNS SSDs. Second, `Z-LFS` devises speculative log stream management according to the workload temperature to maximize active zone utilization. Finally, `Z-LFS` adopts conflict-aware zone allocation to minimize resource contention within ZNS SSDs while considering LFS features. We implement `Z-LFS` based on F2FS in the Linux kernel and evaluate it with commodity ZNS SSD. Our evaluations show that `Z-LFS` achieves higher performance by up to 33.44× and 3.5× compared with F2FS and a state-of-the-art interface for commodity ZNS SSDs, respectively.

1 Introduction

The zoned namespace (ZNS) SSD is a new storage device and has received significant attention from researchers and enterprise storage vendors [7, 19, 42, 52]. In contrast to legacy SSDs (i.e., conventional namespace (CNS) SSD), the ZNS SSD divides the physical address space into fixed-size zones. Each zone must be written sequentially (i.e., append-only update) and reset explicitly for reuse while shifting the responsibility of data management to the host (e.g., garbage collection) [8, 44, 55]. This minimizes the DRAM usage and over-provisioning space inside SSDs [9, 45, 53, 57], resolves the log-on-log issue [17, 56], and mitigates I/O interference [10, 18, 33, 51].

Unfortunately, adapting an existing mature file system on the ZNS SSD is not trivial due to the ZNS constraints. For example, in-place updating file systems (e.g., EXT4 [40] and XFS [54]) cannot be utilized on ZNS SSD. Meanwhile, out-of-place updating file systems including log-structured file systems (LFSs), such as SpriteLFS [49], F2FS [29], and NILFS2 [27], can naturally be utilized on ZNS SSD by performing out-of-place updates. Specifically, F2FS [29] is a representative LFS supporting the utilization of ZNS SSD. Accordingly, F2FS has been widely adopted to utilize ZNS SSD in previous studies [1, 17, 30, 50]. Nevertheless, current Linux LFSs with multiple log streams (e.g., F2FS) may meet two issues due to their CNS-based designs on ZNS SSD. Specifically, 1) LFS has a potential inability to use ZNS SSD as a standalone SSD [1] resulting from CNS-based metadata design and its in-place updates and 2) there can be a performance issue due to CNS-based data placement and I/O operations.

CNS-based metadata design and update: The existing LFSs manage metadata in block-level granularity and perform in-place updates for the metadata in a fixed location to handle well-known LFS issues such as update propagation (i.e., wandering tree problem [13,23,25,29,41,49,59,60]). For example, F2FS performs in-place updates for its metadata stored at a block-aligned and fixed location to solve the wandering tree problem [29]. Accordingly, when utilizing a commodity ZNS SSD, the LFSs require an additional CNS SSD to support the in-place updates. However, this leads to an increase in the cost of a storage system.

CNS-based data placement and I/O operations: The existing LFSs place data on sequential logical block addresses (LBAs) to fully utilize the high sequential write performance of CNS SSD while minimizing GC overhead [16, 34]. Meanwhile, the LFSs may encounter potential under-utilization issues in ZNS SSDs, especially small-zone ZNS SSDs. Specifically, small-zone ZNS SSDs have two main features [15,21,42]: 1) fine-grained mappings between zones and internal resources [43, 46] such as channels and dies and 2) a large number of zones that support parallel write operations (i.e., active zones [44]) at the cost of reduced internal parallelism within a zone (i.e., intra-zone parallelism). Accordingly, to maximize the ZNS SSD performance, the LFSs with multiple log streams should efficiently utilize active zones based on their log streams and consider the mappings to avoid the conflicts among zones that share identical SSD internal

*Corresponding Author: Yongseok Son (`sysganda@cau.ac.kr`).

resources (e.g., dies and channels).

Previous studies have introduced different techniques to efficiently utilize ZNS SSDs on various layers [9, 17, 42, 43]. ZenFS [9] is a storage backend for running RocksDB on ZNS SSDs. ZNS+ [17] introduces a new customized ZNS SSD design and interface which enables offloading data copy operations to the SSD. eZNS [42, 43] is an elastic ZNS interface that allocates active zones based on the application workloads and enhances performance isolation through an I/O scheduler. Our study is inspired by these studies and in line with them in terms of efficiently exploiting ZNS SSDs or maximizing the utilization of active zones for ZNS SSDs. In contrast, our study focuses on designing ZNS-tailored metadata and logging strategies of LFS on a standalone commodity ZNS SSD.

Our goal in this study is to design a ZNS-tailored LFS to maximize 1) the cost-efficiency of the storage system and 2) the resource utilization of small-zone ZNS SSDs. To this end, we propose a novel ZNS-tailored log-structured file system for small-zone ZNS SSDs called `Z-LFS`. Our key idea of `Z-LFS` is to enable 1) ZNS-tailored metadata management to enable standalone usage of ZNS SSD and 2) ZNS resource control by leveraging log stream utilization and addressing resource conflicts.

Specifically, `Z-LFS` first enables append-only updates on metadata to efficiently store metadata on ZNS SSDs without relying on CNS SSD. To do this, `Z-LFS` leverages the unique metadata characteristics of LFS on ZNS SSDs, which differ from those on CNS SSDs due to ZNS constraints. In our observation, LFS metadata on ZNS SSD can be classified into two categories based on their life cycle: immutable and mutable metadata. Immutable metadata is not updated after being written, and its life cycle is identical to that of the associated LFS segments. Meanwhile, mutable metadata is updated frequently and has a distinct life cycle from the segments. With these characteristics, `Z-LFS` appends immutable metadata to its corresponding segment and stores mutable metadata separately using a delta logging approach.

Second, `Z-LFS` devises speculative log stream management that dynamically coordinates active zones to maximize their utilization. We observe that allocating zones without considering the write intensity of log streams can limit the effective use of zone-level parallelism. To address this, `Z-LFS` speculatively determines the active zone quota for each log stream based on its write intensity, which varies according to workloads. Finally, `Z-LFS` adopts conflict-aware zone allocation designed to minimize contention among zones that share SSD internal resources including dies and channels. In our analysis, when a host uses zones mapped to the same die or channel, zone-level parallelism is reduced due to the die or channel-level conflicts. Based on this analysis, `Z-LFS` allocates zones and distributes data across zones mapped to distinct internal resources, thereby reducing the resource conflicts.

Note that our approach is a fully software-based and targets commodity devices to ensure broader applicability and compatibility without requiring modification to ZNS SSDs. By doing so, our file system can be easily deployed on mature storage systems with commodity ZNS SSDs.

We implement `Z-LFS` with the three techniques based on F2FS in Linux kernel 5.17.4. We evaluate `Z-LFS` with various micro/macro benchmarks and RocksDB in a single-tenant environment. The result shows that `Z-LFS` improves the performance by up to 33.4× and 3.5× compared with F2FS and a state-of-the-art interface for commodity ZNS SSD (eZNS [42]) with F2FS, respectively. Finally, we open the source code of `Z-LFS` at `https://github.com/Z-LFS/Z-LFS` to aid future studies in file systems and storage stacks for ZNS SSDs.

2 Background and Motivation

2.1 Small-zone and Large-zone ZNS SSDs

ZNS SSD has two main constraints [32, 35, 44]. Specifically, 1) data must be written sequentially within a single zone, and once data is written, it cannot be overwritten until the zone is reset. In addition, 2) ZNS SSD also restricts the maximum number of active zones where write operations can be performed in parallel [42]. A zone has one of six states (i.e., *opened, closed, full, empty, read-only, and offline*), which can be changed according to operations such as write requests or zone management commands [44]. Within the states, *opened* and *closed* states represent active zones. The active zone shifts to the *full* state when it is filled or through a FINISH zone command, and shifts to the *empty* state through a RESET zone command, allowing the active zone to be reclaimed.

A zone in the ZNS SSD is allocated across channels, and the zone size is determined by the manufacturer [6, 7, 39]. Thus, the degree of internal parallelism can vary depending on the zone size [11, 12, 20, 21]. In a ZNS SSD with a large-zone (large-zone ZNS SSD), each zone spans more dies across channels [42], supporting higher intra-zone parallelism. Meanwhile, since it provides only a few active zones (e.g., up to 14 zones [8, 9]), the large-zone ZNS SSD can limit flexibility and show low performance isolation [15, 42, 46]. On the other hand, a small-zone ZNS SSD exposes finer-grained physical zones; each zone is contained within a die [15, 46]. Thus, the small-zone ZNS SSD can support more active zones (e.g., up to 384 zones), providing higher flexibility and performance isolation. Meanwhile, the small-zone ZNS SSD can show lower parallelism when only a small number of active zones is utilized. To exploit the internal parallelism of the small-zone ZNS SSD (zone-level parallelism), hosts should aim to process I/O requests across a large number of active zones.

2.2 Challenges of LFS on ZNS SSDs

Considering ZNS SSD, the design of LFS based on CNS SSD faces challenges, particularly in addressing three main issues

as follows. We will explain the details of the LFS challenges using F2FS as an example since it supports the ZNS SSD and is one of the representative and widely used LFSs.

Enabling append-only update metadata on ZNS SSD: F2FS places its metadata in a block-aligned and fixed location and updates them in place on CNS SSD. On the other hand, in the case of a ZNS SSD, since it disallows random writes and in-place updates within a zone, the metadata management scheme in F2FS requires an additional CNS SSD, increasing the overall cost of a storage system. To utilize the ZNS SSD as a standalone SSD, we can adapt the existing CNS-based metadata approach (i.e., F2FS) with minor modifications to fit within the ZNS constraints. In this approach, each metadata type has a pair of zones, and the metadata is written alternatively to one of the pair with zone granularity. However, due to the ZNS constraint, when the metadata is randomly updated, this approach may need to read the metadata from one zone in one of the pair, modify the metadata, and write the modified metadata to the other zone. This approach can significantly decrease the performance (up to 9.32×) and increase the amount of write (up to 7.8×) in our preliminary evaluation. Therefore, it is a challenge to efficiently design the metadata under the ZNS constraints and perform append-only updates.

Maximizing utilization of active zone under ZNS constraints: In the case of node/data in the main area of F2FS [29], the multi-head logging design separates the node and data into six types based on their hotness (i.e., hot/warm/cold). F2FS allocates a zone for a log stream and writes corresponding data or nodes sequentially in the zone. When the zone becomes full, F2FS allocates a new zone. Consequently, F2FS uses only six *active* zones at most, limiting zone-level parallelism and under-utilizing the device bandwidth, especially for small-zone ZNS SSDs. To make the best use of the available active zones at maximum, there can be a static and straightforward approach. For example, we can statically map six log streams to available active zones as much as possible. However, the number of available active zones is limited, and the I/O traffic on each log stream varies according to the workloads. Specifically, when the I/O traffic becomes skewed toward a particular log stream, the log streams with low I/O traffic can waste active zones while the log streams with high I/O traffic cannot harness the resources. Thus, this approach may still have the issue of under-utilization of active zones. Consequently, it is a challenge to optimally scale the active zones in an opportunistic manner to maximize the parallelism under the active zone constraints.

Mitigating SSD internal resource conflict among zones: Small-zone ZNS SSD uses fine-grained mapping between host-exposed zone and SSD internal resources such as channel and die. For example, the ZNS SSD used in our evaluation maps each zone to a single die and a single channel, rather than spanning a zone across multiple dies or channels. Accordingly, although an LFS may use multiple active zones to improve zone-level parallelism, if these zones are mapped to the same channels or dies, the LFS may fail to achieve optimal performance due to limited parallelism. Therefore, an LFS should consider and manage the mapping between zones and the internal resources to avoid the resource conflicts.

3 Analysis of ZNS SSD performance

3.1 Basic performance of ZNS SSD

To demonstrate the impact of zone-level parallelism on read and write performance with varying number of zones to issue I/O, we evaluate the base read and write performance of small-zone ZNS SSDs on the testbed described in §6.1. We use FIO [22] to perform direct I/O operations on the raw device, bypassing the page cache, with 16 threads and a queue depth of 32. The number of zones issuing I/O (i.e., active zones for writes) and the request sizes are varied during the evaluation.

As shown in Figure 1, write performance improves as the number of active zones increases across all request sizes. However, the degree of performance improvement and the maximum throughput are significantly influenced by the request size. Specifically, for 128KB requests, write performance saturates at 128/256 active zones, reaching approximately 2.4 GB/s. In contrast, small (e.g., 4KB) and large (e.g., 2MB) request sizes lead to performance degradation. For example, with 128 active zones, the write performance for 4KB and 2MB requests is only 20% and 47%, respectively, of the peak performance achieved. For small requests, the performance degradation is primarily due to the excessive number of I/O operations. For large requests, the I/O processing time within a zone increases, delaying subsequent write requests on other zones, thereby reducing zone-level parallelism and overall performance. Similarly, read performance also increases as the number of active zones increases but saturates with fewer zones (i.e., 32) than write operations. Unlike write operations, read performance is not sensitive to request size. These results highlight the importance of leveraging zone-level parallelism for both read and write operations, particularly given the limited intra-zone parallelism of small-zone ZNS SSDs, as discussed in §2.1.

3.2 Interference among internal resources

To evaluate the impact of interference among internal resources and understand the internal structure of the ZNS SSD, we measure the read performance using two threads to read from two different zones. In this evaluation, one thread reads data sequentially from zone ID 0, while the other reads data sequentially from a different zone, varying from zone ID 1 to 256. Figure 2 illustrates the read throughput of the two threads as the second thread's zone ID changes.

As shown in the figure, we observe two conflict points (i.e., channel-level and die-level) that degrade the performance compared with no conflict points as described in previous studies [43]. The performance degradation occurs when the

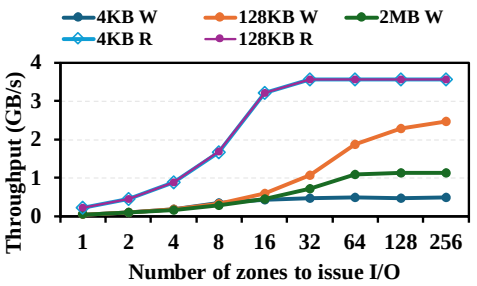


Figure 1: Raw write (W) and read (R) performance of a small-zone ZNS SSD.

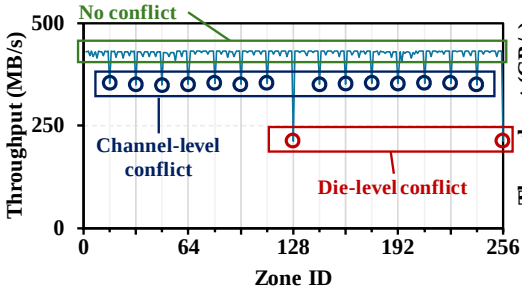


Figure 2: Performance of scenarios with/without resource conflict.

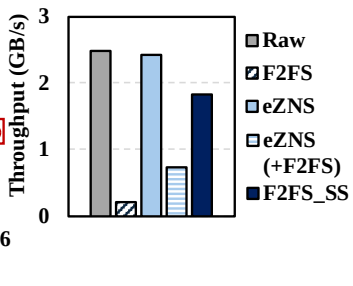


Figure 3: Write performance of various techniques.

zone ID is a multiple of 16, where throughput drops by 19% compared with the no conflict scenario. This degradation results from channel-level conflicts, where I/O requests from both threads contend for access to the same channel in the ZNS SSD even if two different zones are utilized. Second, the performance drops significantly when the zone ID is a multiple of 128. At these points, die-level conflicts arise, reducing performance by more than 50% compared with the no-conflict scenario.

These results indicate that an LFS should allocate zones while minimizing conflicts with the internal resources of the ZNS SSD to achieve optimal performance. Based on these observation, we infer that the ZNS SSD used in the evaluation consists of 16 channels and 128 dies, with zones statically mapped to internal resources in a round-robin manner. Leveraging this model, we design a conflict-aware zone allocation technique described in §4.6.

3.3 File system performance on ZNS SSD

We evaluate the write performance of F2FS, eZNS with F2FS (eZNS (+F2FS)), and F2FS with static stripping (F2FS_SS) as shown in Figure 3. eZNS [42, 43] is a state-of-the-art technique to exploit zone-level parallelism by distributing the write operations to the active zones via logical zones (i.e., v-zone [42, 43]). F2FS_SS represents an extension of F2FS with a static striping approach that evenly distributes active zones across log streams (e.g., 64 active zones per log stream) to further utilize zone-level parallelism in F2FS. For the evaluation, we use FIO with 16 threads performing sequential writes. These threads issue direct I/Os for the raw device and eZNS, while buffered I/Os are used for F2FS, eZNS with F2FS, and F2FS_SS.

F2FS exhibits significantly lower performance compared with the potential performance of the raw device, as described in §2.2 due to a lack of awareness of limited intra-zone parallelism. Specifically, F2FS allocates a single zone for each log stream, utilizing a total of six zones. eZNS delivers full performance in scenarios without F2FS, as FIO threads write uniformly to v-zones, ensuring balanced utilization. However, when combined with F2FS, eZNS experiences write performance degradation due to two main reasons from lack of cross-layer awareness as follows. (1) Similar to standalone F2FS, F2FS even with eZNS recognize v-zones as normal zones. Therefore, F2FS allocates a single v-zone for each log stream and cannot utilize more v-zones [43]. (2) eZNS adjusts the number of active zones mapped to v-zones based on whether write operations occurs in a v-zone, rather than intensity of those operations in the v-zone. As a result, even when a v-zone is heavily utilized, eZNS allocates the same number of active zones to all v-zones regardless of the utilization of each v-zone. Consequently, while eZNS with F2FS improves active zone utilization compared with standalone F2FS, it delivers sub-optimal performance when I/O traffic is skewed toward a specific log stream.

The lack of cross-layer awareness may be addressed through coordination between F2FS and eZNS layers. However, such cross-layer coordination necessitates additional interaction mechanisms and tight coupling between the layers. Specifically, this requires communication burden between F2FS and eZNS whenever the write intensity of log streams changes. For example, when the write intensity of a log stream changes, F2FS transfers the intensity information to eZNS. Using the information, eZNS can increase or decrease the number of active zones mapped with the v-zone associated with the log stream. Therefore, this cross-layer coordination approach can increase system complexity and maintenance overhead.

F2FS_SS demonstrates better performance than F2FS and eZNS with F2FS. However, static striping exhibits sub-optimal performance. It is because, similar to eZNS with F2FS, static striping allocates the same number of active zones to all log streams, regardless of their write intensity, thereby limiting the utilization of zone-level parallelism. The performance gap between eZNS with static stripping primarily arises from differences in request size. eZNS issues small requests (e.g., 4KB) for each v-zone to minimize latency while static stripping issues larger requests (e.g., 2MB).

These results demonstrate that evenly distributing active zones across log streams can waste active zones and limit the zone-level parallelism of write-intensive log streams. To maximize zone-level parallelism across log streams, LFS should dynamically allocate active zones based on the write intensity of each log stream.

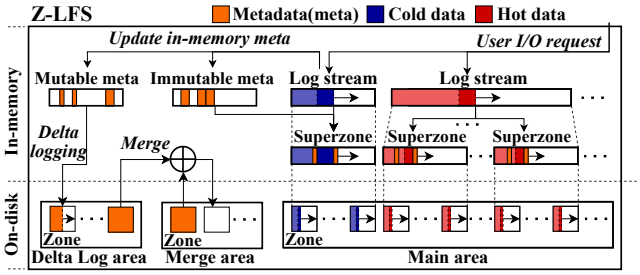


Figure 4: Overall architecture and procedure of Z-LFS.

4 Design and Implementation

4.1 Design Goals of Z-LFS

We design Z-LFS to meet the following three design goals:

- **ZNS-tailored metadata management:** To maximize the cost-efficiency, Z-LFS should update metadata in an efficient append-only manner while considering unique characteristics and constraints of LFS on ZNS SSD.
- **Speculative log stream management:** To maximize active zone utilization, Z-LFS should identify write intensity of each log stream and optimally scale active zones for that stream.
- **Conflict-aware zone allocation:** To avoid conflicts among zones mapped to overlapping internal resources (dies or channels), Z-LFS should track the mapping between them and allocate conflict-free zones as much as possible.

4.2 Strategies of Z-LFS

We present the three key strategies and explain how they meet the design goals of Z-LFS.

Strategy #1: *To enable efficient append-only metadata update while considering LFS features on ZNS SSD, Z-LFS classifies metadata into immutable and mutable types based on their alignment with segment life cycles. Z-LFS manages the two types of metadata separately: it appends immutable metadata to the corresponding segments and adopts delta logging mutable metadata in a dedicated area.*

Strategy #2: *To maximize zone-level parallelism while considering workloads, Z-LFS speculates on and determines the active zone quota for each log stream and dynamically scales active zones to log streams as workload changes.*

Strategy #3: *To avoid channel-level conflict, Z-LFS organizes consecutive zones without channel overlap into a superzone. To prevent die-level conflict, Z-LFS groups superzones mapped to the identical dies as a interference group and selectively allocates superzones from different groups to a log stream.*

4.3 Overall Architecture of Z-LFS

Figure 4 shows the overall architecture of Z-LFS.

Architecture: Z-LFS manages six log streams categorized by their types and temperature similar to F2FS. There are two primary types of log streams: data and node. Data log streams stores user data, while node log streams stores inode or index information for data blocks. Each type is further divided into hot, warm, and cold streams based on hotness. For LFS metadata management, Z-LFS classifies metadata into immutable and mutable metadata based on their life cycle. Immutable metadata shares the same life cycles as their corresponding segments, while mutable metadata has distinct life cycle independent of segments.

In Figure 4, the on-disk layout of Z-LFS is organized into three areas: the delta log area, the merge area, and the main area. Delta log area is dedicated to logging mutable metadata, while merge area stores tables of mutable metadata that is arranged and consolidated from the delta log area. Main area stores data and node log streams, along with their associated immutable metadata. In the main area, considering resource conflicts, Z-LFS organizes consecutive zones into a superzone, which is further divided into multiple fixed-sized segments.

Procedure: As illustrated on the right side of Figure 4, when handling user I/O requests, Z-LFS writes blocks to the appropriate data or node log streams. During this process, it speculates on the write intensity of each log stream as workloads change and dynamically allocates superzones. Specifically, Z-LFS allocates the number of superzones to each log stream in proportion to the ratio of its write requests to the total write requests within a given time window (**Strategy #2**). This ensures that log streams with higher write intensity can utilize more zone-level parallelism. To avoid resource contention, Z-LFS also considers channel-level and die-level conflicts when organizing and allocating superzones (**Strategy #3**). Z-LFS then selects a segment from allocated superzones to a log stream in a round-robin manner and writes blocks sequentially to the chosen segments.

As data and node blocks are written, Z-LFS updates their associated metadata as illustrated on the left side of Figure 4. For immutable metadata, which shares the same life cycle as its corresponding segment, Z-LFS appends the metadata to the end of the segment within the same superzone (**Strategy #1**). This ensures that immutable metadata is naturally cleaned alongside its segment during GC, eliminating the need for additional metadata operations such as metadata GC.

On the other hand, mutable metadata is managed via a different mechanism from immutable metadata. As mutable metadata entries are small (typically tens of bytes) and frequently updated, Z-LFS buffers the changes (deltas) and logs them to a dedicated delta log area. The delta log area operates as a circular log and is cleaned when half of the area is filled. For cleaning, Z-LFS performs a merge operation which consolidate logged deltas from the delta log area into a table and then writes them to the merge area. To reduce the impact on user I/O performance, this merge operation is performed asynchronously.

4.4 Append-only metadata management

4.4.1 Life cycle based metadata management

In LFSs on CNS SSD, life cycles of metadata is distinct from those of segments since segments can be updated after being written. For example, slack space recycling [29, 47], used in LFSs to reduce GC costs, enables writing new data into invalidated space within already written segments. This necessitates frequent updates to metadata associated with the segment, resulting in metadata life cycles that diverge from those of the segments.

LFSs on ZNS SSDs, unlike those on CNS SSDs, have metadata that share the same life cycle as a segment due to the sequential write constraints of ZNS SSDs. Specifically, in LFSs on ZNS SSDs, once a segment is written, its blocks cannot be updated until GC. Instead, the blocks can only be invalidated, and the segment must be erased before reuse. Accordingly, the life cycle of certain metadata is identical to that of its associated segment. For example, a metadata (e.g., segment summary (SS) in F2FS) stores reverse mapping information of blocks within a segment, including 1) the IDs of all nodes pointing to the blocks and 2) their offsets within the nodes. This information is never modified once written, as the blocks within the segment cannot be updated. Such metadata is utilized during GC, and its life cycle ends when the corresponding segment is cleaned by GC. After GC, the metadata is always newly written. We classify such write-once and never-updated metadata as immutable metadata. In contrast, metadata like bitmaps indicating valid blocks in a segment (e.g., segment information table (SIT) in F2FS) and block addresses of nodes (e.g., node address table (NAT) in F2FS) are frequently updated, which we categorize as mutable metadata.

Leveraging the observation that the life cycle of immutable metadata aligns with its corresponding segment, `Z-LFS` appends immutable metadata to the end of the segment and stores them together in the same zone. this approach ensures that the immutable metadata remains valid as long as the segment lives and is naturally cleaned together during GC. As a result, `Z-LFS` eliminates the need for additional operations such as metadata GC and over-provisioned area to perform them for the immutable metadata. In addition, since the location of the immutable metadata is fixed to the end of the segment, `Z-LFS` avoids tracking metadata locations, typically required when metadata is stored in an append-only manner.

4.4.2 Metadata delta logging

`Z-LFS` manages mutable metadata using an append-only approach with delta logging. Specifically, mutable metadata is typically small, with most entries consisting of only a few dozen bytes. In addition, since each entry holds independent information, only some entries may be updated while others remain unchanged. Thus, a block which contains mutable metadata can be partially updated. To efficiently handle the partial metadata updates, `Z-LFS` collects only the modified entries (i.e., deltas), aggregates them into a new metadata log block, and logs the block in delta log area. We demonstrate that the write amplification of `Z-LFS` is comparable to that of F2FS on CNS SSD, even under metadata-intensive workloads with frequent fsync operations (see Figure 11).

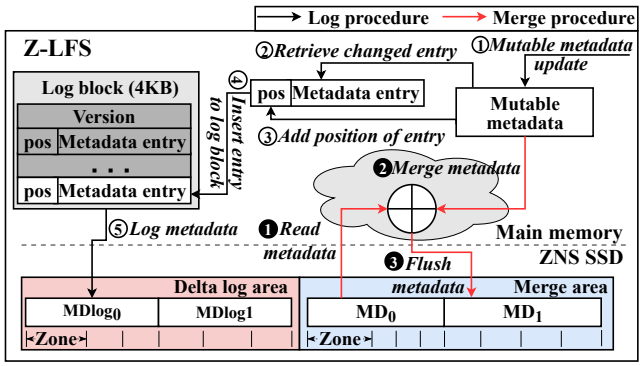


Figure 5: Architecture and procedure of delta logging (MD_x: one side of metadata table pair, $MDlog_x$: one side of metadata log pair, pos: position of the metadata entry).

Meanwhile, a pure log-structured approach that logs metadata blocks with a GC procedure could be a candidate for managing mutable metadata. However, with this approach, updating the metadata changes its location, causing the metadata to be dispersed across the ZNS SSD. As a result, the file system should track and persist the location of each metadata block. This can lead to the need for metadata of metadata and add an additional complexity to metadata management. In contrast, `Z-LFS` consolidates metadata locations to eliminate the need to track the locations of individual metadata blocks.

Architecture: Figure 5 depicts the architecture and overall procedure of delta logging in `Z-LFS`. For metadata delta logging, a pair of logs consisting of zones (e.g., $MDlog_0$, $MDlog_1$) is maintained in the delta log area. The metadata deltas are written to the pair in a circular manner. To keep the consistency of mutable metadata, `Z-LFS` maintains a pair of metadata tables (MD_0, MD_1) in the merge area. `Z-LFS` alternatively updates one side of the metadata table pair, ensuring that one side always contains valid metadata while the other may contain outdated metadata. By combining circular logging deltas in the delta log area with alternative metadata updates in the merge area, `Z-LFS` prevents partial metadata updates and ensures crash consistency, as described in §4.7.

Delta logging procedure: During user I/O requests processing, mutable metadata are updated in memory (①). Only when `Z-LFS` needs to perform a checkpoint[1], the delta logging is triggered. For delta logging, `Z-LFS` retrieves changed entries (①) and appends their positions (`pos`) to each entry (②). `pos` indicates the location of the metadata entry in the metadata table, stored in the merge area, ensuring it can be identified after system recovery. Then, `Z-LFS` inserts the en-

[1]The trigger condition of the checkpoint is the same as that in F2FS.

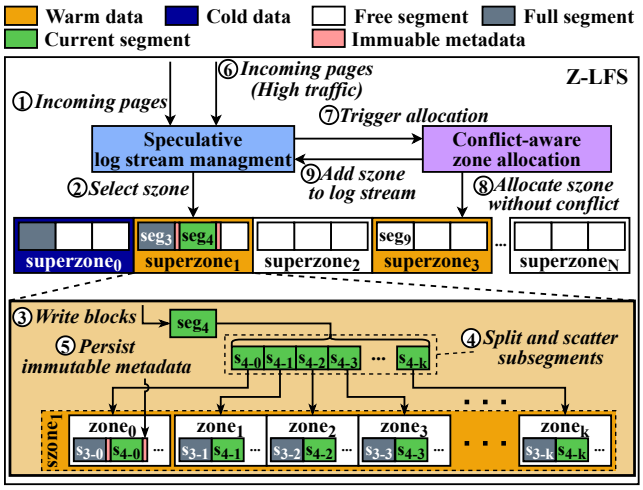


Figure 6: Architecture and procedure to process user data in Z-LFS (szone: superzone, seg_n: n^{th} segment, $s_{n\text{-}k}$: k^{th} subsegment of n^{th} segment).

tries in a 4KB log block (④). When the log block becomes full, it is written to the delta log area ($MDlog_0$) (⑤). This procedure of ①-⑤ is repeated until Z-LFS logs all of changed entries in the delta log area. If the current delta log areas ($MDlog_0$) overflows during logging, the remaining entries are written to the next delta log area ($MDlog_1$). During delta logging, Z-LFS records a log version number in log blocks. The log version increments whenever delta logging is invoked and is used during system recovery to determine which log block should be recovered, as described in §4.7.

Asynchronous merge procedure: When a delta log area (e.g., $MDlog_0$) becomes full during a log operation, Z-LFS triggers an asynchronous merge operation to clean the area and reclaim space for future logs. In this process, Z-LFS first reads the valid side of the metadata table pair (MD_0) into memory (❶). It then merges this metadata with the entries logged in the delta log area (❷). To reduce overhead, Z-LFS buffers the entries in memory when they are logged in the delta log area. This allows Z-LFS to avoid scanning the delta log area on the ZNS SSD when retrieving logged metadata entries, at the cost of additional memory consumption (§6.3). However, the total size of memory buffer is limited by the total size of the delta log area. After merging, Z-LFS flushes the updated metadata table to the storage device (❸). Once the flush operation completes, Z-LFS safely resets the delta log area ($MDlog_0$) applied to the merge area (MD_1), reclaiming the space for future use.

4.5 Speculative log stream management

Figure 6 shows the architecture and overall procedure to process user I/O requests.

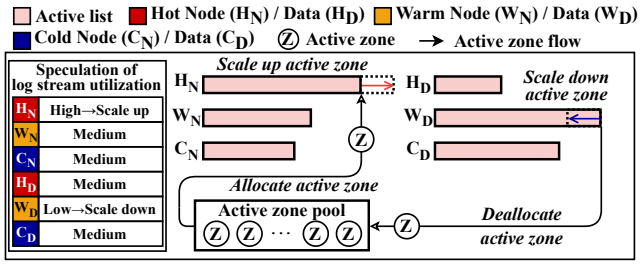
Architecture: Z-LFS organizes all zones in the main area into superzones. Superzones are logically divided into segments for finer space management, and the segments contain subsegments. The subsegments in a segment are logically continuous but physically mapped to non-contiguous blocks (e.g., a distinct 128KB). The organization of superzones, segments, and subsegments is determined during file system initialization. Z-LFS has six log streams (e.g., hot/warm/cold data and node) and tracks the current segment, which indicates the last written position for each log stream. For crash consistency, the information of the current segments persists in checkpoint blocks and can be recovered as in F2FS.

Procedure: In Figure 6, when Z-LFS flushes data and node, it identifies the type of log stream (i.e., warm data) associated with incoming pages (①). Then, to store the pages, Z-LFS selects a superzone of the identified log stream ($superzone_1$). Subsequently, to initiate the I/O operation, it selects the first free segment (seg_4) within the superzone and then allocates blocks from the segment (②). For a segment, Z-LFS splits the segment (seg_4) into smaller subsegments ($s_{4\text{-}0}$-$s_{4\text{-}k}$) and scatters each subsegments across the zones ($zone_0$-$zone_k$) in the superzone (③,④). Z-LFS keeps the order of segments (seg_3, seg_4, ...) within each superzone as the order of subsegments ($s_{3\text{-}0}$, $s_{4\text{-}0}$, ...) within each zone. By doing so, it can keep the sequential write constraint within each zone and easily recover the mapping of subsegment and LBA. When a segment becomes full, its corresponding immutable metadata is appended to the end of the segment (⑤).

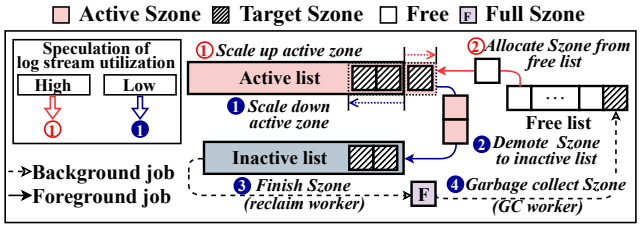
When the warm data log stream is speculated to be high-traffic (⑥), Z-LFS scales up the active zones for the log stream. Then, to allocate an additional free superzone to increase active zones, Z-LFS triggers conflict-aware zone allocation (⑦). Z-LFS allocates superzone without conflict with previously allocated superzones (⑧) and adds the superzone to the log stream (⑨). With the superzones (i.e., $Superzone_1$ and $Superzone_3$) allocated for the log stream, Z-LFS selects a superzone in a round-robin manner (e.g., $superzone_3 \rightarrow superzone_1 \rightarrow superzone_3$) and selects the first free segment (e.g., seg_9) within the selected superzone ($superzone_3$) to provide higher zone-level parallelism.

Speculation of I/O utilization of log streams: To determine the optimal number of active zones for each log stream, Z-LFS employs a speculative, quota-based approach for managing active zones. This approach speculates on write intensity and allocates an appropriate quota of active zones to each log stream. Additionally, Z-LFS performs the speculation separately for data and node log streams. This separation is based on our observation, as described in §4.6, that data and node log streams are typically not used concurrently.

To do this, we define the total available active zones for data or node log streams (A). A is determined as the smaller of the following values: (1) the number of active zones required to achieve peak throughput (A_{peak}) and (2) half of the maximum number of active zones available after reserving zones for file system metadata management (A_{avail}). Z-LFS uses the smaller value to keep the maximum number of active zone constraint of ZNS SSD. Thus, A is defined as $A = min(A_{peak}, \frac{A_{avail}}{2})$.



(a) Overview of scaling active zones of multiple log streams



(b) Procedure of scaling active zones within a log stream.

Figure 7: Dynamic scaling of active zones of log streams.

The number of active zones allocated to a specific log stream i (A_i) is determined proportionally to its write request volume (W_i) relative to the total request volume among log streams of the same type ($W_T = \sum_{i \in T} W_i$, where T represent either data or node) within a time window. This is expressed as: ($A_i = A \times \frac{W_i}{W_T}$). After each time window elapses, Z-LFS tracks the write traffic for each log stream and adjusts the allocation of active zones accordingly.

Scaling active zones of log streams: Figure 7a depicts an overview of scaling active zones across log streams. Z-LFS maintains a global active zone pool to keep the active zone constraint of ZNS SSD while scaling active zones of log streams based on the speculation. For each log stream, Z-LFS compares the allocated number of active zones by the speculation with the current number of active zones utilized by the log stream. If the allocated number is greater than the current utilization, Z-LFS determines that the log stream's traffic to be low. Conversely, if the allocated number is smaller, the traffic of the log stream is considered high. As shown in the figure, for a log stream speculated as high I/O traffic (e.g., hot node log stream: H_N), Z-LFS scales up active zones by allocating them from the active zone pool to the log stream. Meanwhile, when a log stream exhibits low utilization (e.g., warm data log stream: W_D), Z-LFS scales down and deallocates the active zones occupied by the log stream.

Scaling up active zone: Figure 7b shows an example of a procedure for scaling active zones within a log stream. As shown in the figure, Z-LFS manages active, inactive, and free lists of superzones for efficient active zone scaling. The active list contains superzones where write requests are actively performed. Z-LFS selects a superzone in the active list in a round-robin manner and allocates a segment in the superzone. In this example, as utilization of the log stream increases,

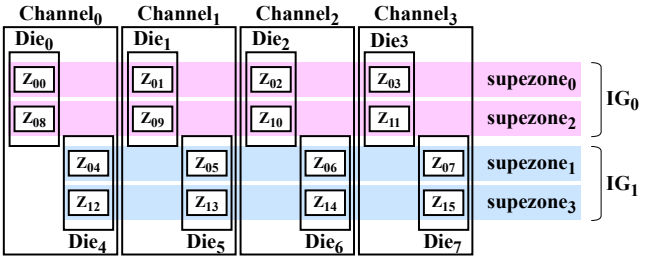


Figure 8: Superzone and IG organization for conflict-aware zone allocation. (Z_n: n^{th} zone, IG_n: interference group n)

Z-LFS tries to scale up the active zones allocated to the log stream (①). To do this, it allocates a free superzone from the free list and inserts the superzone into the active list (②). Then, the allocated superzone is utilized by Z-LFS in a round-robin manner, naturally increasing zone-level parallelism.

Scaling down active zone: As shown in Figure 7b, when the utilization of the log stream decreases due to workload change, Z-LFS scales down active zones (❶). To do this, Z-LFS demotes two superzones in the active list to the inactive list (❷) and reclaims the active zones of the superzones in the inactive list by issuing FINISH zone commands in a background manner (❸). Finally, after a GC worker frees the superzones according to the GC policy, the superzones are shifted from the full state to the free state and are inserted into the free list (❹). Note that the reclaim and GC processes are decoupled from the normal I/O path by the background workers to minimize the overhead caused by the FINISH zone commands and GC, respectively.

4.6 Conflict-aware zone allocation

Superzone and IG organization: Figure 8 illustrates the organization of superzones and interference groups (IGs) based on the modeling of ZNS SSD internal resources, as described in §3.2. Since zones are mapped to dies and channels in a round-robin manner, allocating consecutive zones can help avoid resource conflict. For example, in a ZNS SSD with 8 dies and 4 channels as shown in the figure, Z_{00} ($zone_{00}$), Z_{01}, Z_{02}, and Z_{03} are mapped to different channels, while Z_{00} and Z_{04} are mapped to the same channel. Similarly, Z_{00}-Z_{07} are mapped to different dies, while Z_{00} and Z_{08} are mapped to the same die.

Based on this model, to avoid conflict while reducing the complexity of zone management, Z-LFS organizes consecutive zones (Z_{00}-Z_{03}) into a superzone ($superzone_0$) to avoid the channel-level conflict. As a result, each superzone consists of zones equal to the number of channels. Furthermore, Z-LFS groups superzones mapped to the same die ($superzone_0$ and $superzone_2$) into an interference group (IG_0). This static organization, in which the number of interference groups corresponds to the number of dies divided by the number of channels, is determined by the zone IDs as well as the number of channels and dies. Based on the analysis in §3.2, Z-LFS organizes a superzone with 16 zones and groups superzones

into 8 IGs for our testbed ZNS SSD.

Superzone allocation for log streams: To minimize the die conflicts, Z-LFS allocates active zones in units of superzones to the log streams and prioritizes the allocation of superzones from non-overlapping IGs, avoiding overlap with superzones from the previous IG. Furthermore, we observe that data and node log streams are typically not used and written concurrently. Specifically, node log streams are primarily written during checkpointing, while data log streams are blocked during this process. Thus, enforcing non-overlapping IG allocations without differentiating between data and node log streams may unnecessarily limit active zone utilization. For example, if there are two IGs and a superzone is allocated from one IG to a node log stream, data log streams can only use a superzone from the other IG. This limits the number of superzones that can be utilized even if no die conflicts occur. As a result, this observation is incorporated into the allocation strategy to further exploit active zone utilization.

To do this, Z-LFS maintains separate IG allocations for data and node log streams by maintaining distinct IG lists. When a superzone is allocated to a data or node log stream, Z-LFS marks its corresponding IG in the respective allocation list to prevent other concurrent log streams from being assigned superzones within the same IG. By doing so, since blocks of the same log stream are written to zones that do not overlap at the die level, this approach avoids conflicts during write operations. Similarly, it ensures conflict-free sequential reads, since blocks were already written to non-overlapping zones. We note that if all superzones within an IG become completely exhausted, Z-LFS can allocate superzones from overlapping IGs as a fallback. To minimize such cases, Z-LFS prioritizes selecting a superzone from the IG with the fewest remaining free superzones as a victim during a GC procedure.

4.7 Crash Consistency

Z-LFS recovers the file system state against a system crash with roll-back and roll-forward recovery mechanisms, similar to F2FS. Roll-back restores the state to that after the last committed checkpoint (CP), meanwhile, roll-forward restores the state to that after the last `fsync()` call from the last committed CP. We focus on explaining the roll-back recovery of metadata since other recovery procedures are performed in the same way as F2FS.

Immutable metadata is appended directly to the end of its corresponding segment. However, if a segment is partially written, its corresponding immutable metadata is written to the checkpoint block during checkpointing. As a result, Z-LFS ensures the consistency between segment and its metadata through the checkpoint process as F2FS does.

To recover mutable metadata, Z-LFS scans deltas in all the log blocks in the delta log area and merges the deltas with the valid metadata tables to the outdated merge area. Specifically, Z-LFS first references checkpoint blocks. In Z-LFS, these blocks store information about which of the mutable metadata tables in pairs in the merge area is valid. This allows Z-LFS to read the valid sides of metadata table pairs and identify the valid mutable metadata table. During the recovery procedure, Z-LFS scans log blocks in the delta log area and retrieve valid log blocks. Specifically, the checkpoint blocks in Z-LFS store the latest log version when the checkpoint is performed. Thus, Z-LFS can compare the versions in log blocks (i.e., log version) with the version in the last committed checkpoint (i.e., checkpoint version). If the log version is equal to or less than the checkpoint version, Z-LFS considers it valid. Otherwise, Z-LFS ignores the log block since the corresponding checkpoint for the log block is not committed. Then, Z-LFS identifies the position (pos) of each metadata entry (delta) from the valid log blocks. Using these positions, Z-LFS merges the deltas with the valid metadata table. Finally, Z-LFS flushes the merged metadata table to the outdated merge area and proceeds to other recovery procedures, including roll-forward.

After Z-LFS recovers from the system crash with roll-back and roll-forward, Z-LFS does not recover the active zone pool. Instead, it initializes the active zone pool while reclaiming the active zones within ZNS SSD, starting with a clean and consistent active zone state. To reclaim the active zones, Z-LFS changes the zones from the *closed* state[2] to the *full* state. Specifically, Z-LFS scans the states of zones in the ZNS SSD via a REPORT zone command and finishes the zones in the *closed* state via a FINISH zone command. We note that this recovery does not affect the consistency of Z-LFS since FINISH zone commands do not change the contents of blocks. Similarly, IG allocation lists are initialized since allocated superzones for log streams are reclaimed.

4.8 Implementation

We implement Z-LFS based on F2FS and categorize metadata in F2FS into immutable and mutable metadata. For example, Z-LFS classifies segment summary (SS) as immutable metadata, and segment information table (SIT) and node address table (NAT) as mutable metadata. Since SS, which stores reverse mapping information, is write-once and never-updated metadata, Z-LFS considers SS as immutable metadata. On the other hand, SIT includes bitmaps indicating valid blocks in a segment and NAT includes the logical address of node blocks, which are frequently updated regardless of segment life cycles. Z-LFS manages these metadata as mutable metadata through delta logging and merge.

Since SIT and NAT may have different lifetimes, managing them together can lead to unnecessary merge overhead. For instance, if SIT is updated more frequently than NAT, NAT may experience unnecessary merges as SIT exhausts the delta log area. To avoid this, Z-LFS allocates separate delta log areas, merge areas, and memory buffers for SIT and NAT, and

[2] After a system crash, the ZNS SSD changes zones from the *opened* state to the *closed* state. However, the zones with *closed* state still occupy active zones [30, 44].

performs delta logging and merging independently for each. This approach allows Z-LFS to efficiently handle metadata updates according to each metadata type, minimizing write amplification and performance degradation.

5 Discussion and limitation

Large-zone ZNS SSDs: Z-LFS can be adopted for large-zone ZNS SSDs. Since append-only metadata management of Z-LFS requires a small amount of active zones, it can be adopted on a large-zone ZNS SSD even if it has a more strict constraint on active zones than a small-zone ZNS SSD. Furthermore, Z-LFS can use the small random writable area in the large-zone ZNS SSD [9] by placing mutable metadata and updating it in-place. On the other hand, speculative log stream management and conflict-aware zone allocation may not be beneficial for the large-zone ZNS SSD even if adopted, because the large-zone ZNS SSD we used already leverages zone parallelism at the device level. However, if a large-zone ZNS SSD has sufficient active zones for higher flexibility at the cost of intra-zone parallelism, Z-LFS can leverage the active zones, increasing zone-level parallelism.

Single-stream LFS: All Z-LFS techniques are effective in an LFS with multiple log streams. However, in an LFS with a single log stream, speculative log stream management may not yield meaningful performance improvements since it scales active zones across multiple log streams. Meanwhile, the other techniques of Z-LFS, such as append-only metadata management and conflict-aware zone allocation, still are effective, since issues caused by CNS-based metadata design and SSD resource conflict persist even in the single-stream LFS.

Multi-tenant environments: While Z-LFS is designed and evaluated in a single-tenant environment, it can also be deployed in a multi-tenant environment. However, Z-LFS can be further optimized to better support multi-tenancy. For instance, we can incorporate I/O scheduling and rate-limiting mechanisms in Z-LFS as in prior studies [38, 42]. This can improve performance and fairness, especially under bursty write workloads. We leave such optimization and evaluation as future work.

Space overhead of Z-LFS: Z-LFS can reduce the storage system cost by append-only metadata management. However, Z-LFS introduces additional space overhead to manage LFS metadata on a standalone ZNS SSD due to the ZNS constraints. In the delta log area, Z-LFS requires at least two zones to circularly log mutable metadata. In the merge area, similar to an existing LFS (e.g., F2FS), Z-LFS maintains a pair of metadata table. However, since the pair should be managed by a zone granularity, the space overhead increases. By appending immutable metadata to the end of its associated segment, Z-LFS reduces the space overhead by 10.7× compared to performing delta logging for both immutable and mutable metadata. The total space overhead of Z-LFS, including both delta log and merge areas, accounts for approximately 0.02% of the total ZNS SSD capacity. This result demonstrates that the space overhead of Z-LFS is negligible.

6 Evaluation

6.1 Experimental setup

Testbed: We use a machine equipped with i7-13700K 3.4GHz CPU (16 physical cores) and 32GB main memory. We use a commodity ZNS SSD [5] and its equivalent CNS SSD [3]. The ZNS SSD consists of 40,704 zones, each with a size of 96MB, providing a total space of 3.92TB. The maximum number of active zones is 384. We use Linux kernel 5.17.4 and f2fs-tool 1.15.0 for the mkfs tool.

Comparison: We compare Z-LFS with F2FS [29], F2FS with static striping (F2FS_SS) as described in §3.3, ZenFS [9], eZNS [42, 43] which is the state-of-the-art ZNS interface for commodity ZNS SSDs, and eZNS with F2FS. F2FS and F2FS_SS are mounted on ZNS SSD with CNS SSD, denoting F2FS (+CNS SSD) and F2FS_SS (+CNS SSD), respectively. To evaluate eZNS without any file system (i.e., raw device), we denote it as eZNS. To evaluate the performance of eZNS with a file system, we mount F2FS on ZNS SSD with eZNS and denote it as eZNS with F2FS (+CNS SSD), since it also requires an additional CNS SSD. We configure a namespace in eZNS to allow all threads in the namespace to utilize all available active zones (384) regardless of the namespace.

6.2 Workloads

Micro-benchmark: We employ FIO [22] to evaluate Z-LFS using the sequential and random read/write workloads with 16 threads. With file systems (e.g., F2FS, F2FS_SS, eZNS with F2FS (+CNS SSD), and Z-LFS), each thread performs 10GB I/O to its dedicated file with a 4KB request size[3]. For comparisons without any file system (e.g., Raw, eZNS), each thread issues 10GB I/O with a 128KB request size. In this case, we use the 128KB request size to demonstrate ideal performance on a raw device and use it for comparison. Unless otherwise stated, we use this configuration in all the FIO evaluations. We also evaluate file system-level garbage collection (GC) to show the effectiveness of Z-LFS even in GC. In this experiment of GC, to reduce the aging time, we create a 150GB partition of a ZNS SSD, pre-fill 128GB in the partition, and then perform random writes for 10 minutes to trigger file system-level GC. Furthermore, we use MDtest [36] to evaluate Z-LFS, under the metadata-intensive workloads. For the evaluation, we create 1 million files with various file sizes of 4/32/128KB and fsync after each file creation.

Macro-benchmark: We use Filebench [2] with four workloads including fileserver, varmail, webserver, and videoserver. We use 16 threads, 16MB file size in fileserver, 1MB in varmail and webserver, and 1GB in videoserver.

[3]We mostly use 4KB as a request size since it is the default size and usually used in most OS and applications [24, 31, 48].

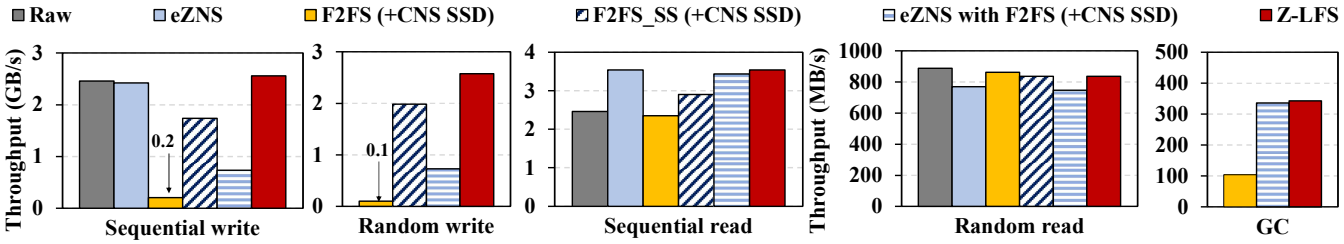


Figure 9: Throughput of sequential/random write/read and garbage collection (GC).

File system	F2FS (+CNS SSD)	F2FS_SS (+CNS SSD)	eZNS with F2FS (+CNS SSD)	Z-LFS
Average latency	626 us	31.9 us	89.4 us	**24.5 us**
Tail latency	47 ms	103.9 us	39 ms	**45.3 us**

Table 1: Write average and tail latency (99.9%)

RocksDB: To evaluate Z-LFS with the real-world application, we use RocksDB [14] and its benchmark tool, db_bench with fillseq, fillrandom, readrandom, and overwrite workloads as in Figure 14. We use 16 threads, 50 million operations, and 1KB value size.

6.3 Micro-benchmark

Write throughput: In Figure 9, Z-LFS achieves higher performance by up to 12.4×/25.2× and 47%/30% compared with F2FS and F2FS_SS in the case of sequential/random writes, respectively. The performance gap between F2FS and Z-LFS results from the better exploitation of zone-level parallelism by utilizing multiple active zones per log stream. Z-LFS further improves utilization of zone-level parallelism and performance over F2FS_SS by speculative log stream management technique. Specifically, Z-LFS utilize up to 128 active zones for the most intensive log stream (i.e., warm data log stream) while the static striping in F2FS_SS limits the maximum utilization as 64 zones.

Z-LFS improves performance by up to 3.5× on both sequential and random write workload compared with eZNS with F2FS. As described in §3.3, due the lack of cross-layer awareness of F2FS, eZNS achieve sub-optimal utilization of active zone, leading to performance degradation. On the other hand, since Z-LFS is the file system itself, it can measure the current utilization for each log stream at the file system level, identify the intensively utilized log streams, and allocate more active zones to them than less utilized log streams.

Read throughput: In Figure 9, interestingly, Z-LFS improves the sequential read performance by 50% compared with F2FS. It is because Z-LFS scatters data across multiple zones while leveraging internal parallelism, even in the read operation. Similarly, F2FS_SS and eZNS with F2FS also achieve higher performance compared with F2FS. Furthermore, Z-LFS improves performance by 22% over F2FS_SS, resulting from the conflict-aware zone allocation of Z-LFS as described in §4.6. The random read performance of the file systems is overall low due to low locality, and Z-LFS exhibits similar random read performance to F2FS without additional overhead.

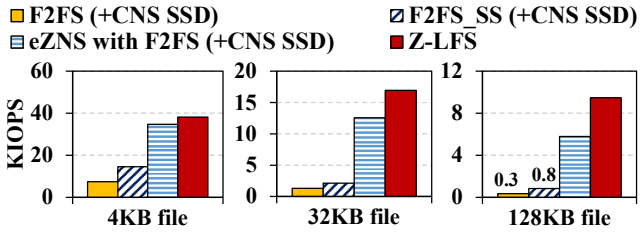
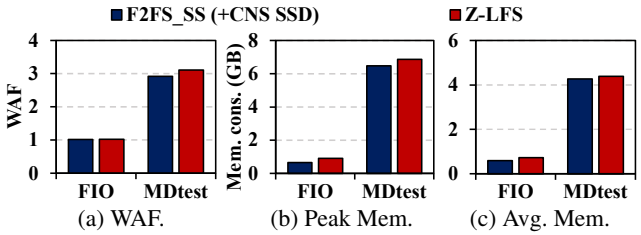


Figure 10: Throughput of metadata-intensive workloads

Write latency: Table 1 presents the average and 99.9th percentile tail latency in the random write workload. As shown in the table, Z-LFS reduces the average latency by up to 96.1%, 23.23%, and 72.63% and tail latency by up to 99.90%, 56.4%, and 99.88% of F2FS, F2FS_SS, and eZNS with F2FS, respectively. This result demonstrates that Z-LFS successfully improves the write latency as well as throughput. This is because Z-LFS can quickly flush the page cache by exploiting zone-level parallelism. Thus, upcoming application write requests can be cached in the page cache with reduced delay.

GC-intensive workload: In Figure 9, Z-LFS exhibits up to 3.3× higher throughput compared with F2FS on ZNS SSD under file system GC operations, while showing similar performance to eZNS with F2FS. The performance improvement over F2FS is attributed to speculative log stream management of Z-LFS, which distributes GC operations of log streams across multiple active zones similar to the normal write operations. Meanwhile, the overhead from the file system GC operations limits the GC performance of Z-LFS. These results indicate that Z-LFS can leverage zone-level parallelism even in the scenario involving frequent file system GC operations.

Metadata-intensive workload: Interestingly, a metadata-intensive workload (e.g., file creation) in F2FS frequently writes the metadata (i.e., inode or dentry blocks) in ZNS SSD instead of CNS SSD. Thus, even in metadata intensive workloads, the efficient utilization of active zones can considerably impact performance. Accordingly, as shown in Figure 10, Z-LFS achieves a significant improvement of up to 27.7× and 11.4× compared with F2FS and F2FS_SS in the file creation workload in MDtest, respectively. Compared with eZNS with F2FS, Z-LFS achieves up to 1.6× higher performance since it can scale up the node log stream via speculative log stream management. This highlights the importance of managing active zones for LFS log streams more flexibly. These results



(a) WAF. (b) Peak Mem. (c) Avg. Mem.

Figure 11: Write amplification factor and memory consumption.

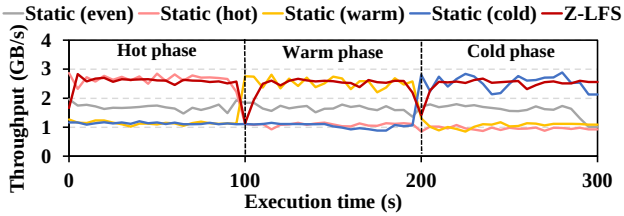
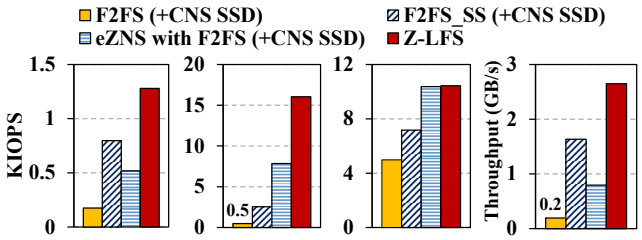


Figure 12: Write throughput of active zone scaling of log streams.



(a) Fileserver (b) Varmail (c) Webserver (d) Videoserver

Figure 13: Throughput of various filebench workloads.

demonstrate that Z-LFS can improve the performance even in the metadata-intensive workloads.

Write amplification: To evaluate write amplification factor (WAF) impact of Z-LFS in both data and metadata-intensive workloads, we measure WAF using random write of FIO and MDtest with a 4KB file size. Figure 11a represents that the WAF of Z-LFS is similar to that of F2FS_SS on both workloads. This result demonstrates that the append-only metadata management of Z-LFS minimizes the metadata WAF.

Memory consumption: We measure the average and peak memory consumption of Z-LFS. In Figure 11b, the peak memory consumption of Z-LFS is higher by up to 38% and 3% compared with F2FS in FIO and MDtest, respectively. Figure 11c shows that the average memory consumption of Z-LFS is higher by up to 22% and 6% compared with F2FS in FIO and MDtest, respectively. Although this result shows Z-LFS has a higher memory consumption than that of F2FS due to the memory consumption from memory buffers used in delta logging, the absolute amount of consumed memory is relatively low. Specifically, Z-LFS consumes an extra 250MB and 380MB memory compared with F2FS in FIO and MDtest, respectively. Furthermore, the amount of memory required in Z-LFS does not grow indefinitely since the maximum memory consumption of all the memory buffer is bounded by the size of all the delta log areas. Consequently, this result demonstrates that Z-LFS achieves append-only metadata management at the cost of memory consumption but the additional absolute amount of memory is relatively small and limited.

Effectiveness of active zone scaling: We evaluate the effectiveness of scaling active zones of log streams compared with the four static allocation schemes (static) in Z-LFS as in Figure 12. For static (even), we use F2FS_SS which allocates even number of active zones for all log streams. For each static (hot/warm/cold) scenario, we allocate 128 active zones for each hot/warm/cold data log stream and 32 active zones for the others, respectively. We use an FIO random write workload to configure three phases (hot, warm, cold) and generate data for each phase by varying file extensions. During each phase, each data is written to its corresponding data log stream. Each phase is executed for 100 seconds, resulting in a total of 300 seconds. Z-LFS with active zone scaling achieves comparable performance compared with each optimal static scheme for each phase. Consequently, the results indicate that Z-LFS optimally scales active zones in response to varying workloads at runtime.

6.4 Macro-benchmark

Fileserver: Figure 13a shows Z-LFS improves the performance by up to 7.21×, 1.6×, and 2.47× compared with F2FS, F2FS_SS, and eZNS with F2FS, respectively, in the fileserver workload. The result shows Z-LFS yields a significant improvement even in a realistic data-intensive workload.

Varmail: Figure 13b shows that Z-LFS achieves improvements of up to 33.44×, 6.30×, and 2.04× compared with F2FS, F2FS_SS, and eZNS with F2FS, respectively. Since varmail generates and flushes nodes frequently, Z-LFS can accelerate the flushing of the nodes to the main area, similar to the MDtest result.

Webserver: In Figure 13c, Z-LFS outperforms F2FS and F2FS_SS by up to 2.09× and 1.62×, respectively, even if webserver is a read-intensive workload. It is because the read operations can benefit from the high utilization of multiple zones, similar to the read cases depicted in Figure 9. Furthermore, Z-LFS improves performance over F2FS_SS, resulting from the conflict-aware zone allocation similar to the result of read throughput in §6.3. eZNS with F2FS also shows high read performance since it can leverage the internal parallelism from multiple zones where data is scattered.

Videoserver: In Figure 13d, Z-LFS improves the performance by up to 13.7×, 1.62×, and 3.33× compared with F2FS, F2FS_SS, and eZNS with F2FS, respectively, in the videoserver workload. Since videoserver includes more data-intensive operations compared with fileserver, Z-LFS can achieve higher performance.

6.5 Real-world Application

In this evaluation, we compare Z-LFS with a recent RocksDB optimized file system, ZenFS [9], F2FS, F2FS_SS, and eZNS with F2FS. We note that we use a CNS SSD for ZenFS to support the in-place updates for RocksDB's log/lock files

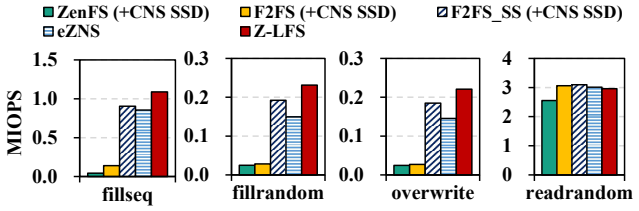


Figure 14: RocksDB performance under various workloads.

using F2FS. Figure 14 shows that Z-LFS improves the performance by up to 25.0×, 7.83×, 1.20×, and 1.27× in fillseq; 9.28×, 7.83×, 1.20×, and 1.55× in fillrandom; and 9.01×, 8.12×, 1.19×, and 1.52× in overwrite compared with ZenFS, F2FS, F2FS_SS, and eZNS with F2FS, respectively. As expected, in readrandom, Z-LFS achieves similar performance compared with F2FS, F2FS_SS and eZNS with F2FS. These results demonstrate that three techniques in Z-LFS can also be effective even in real-world applications.

7 Related Work

Interfaces on ZNS SSDs: eZNS [42] proposes an elastic ZNS interface which allocates active zones based on the application workload profile and enhances performance isolation with an I/O scheduler. CSAL [61] is a storage acceleration layer to take advantage of high-density QLC SSD by adopting ZNS interface on the QLC SSD in a cloud environment. ZMS [20] is a zone interface on mobile flash storage to address the lack of on-device memory and the frequent fsync. Our study is in line with these studies [20, 42, 61] in terms of improving ZNS-based storage systems while we focus on devising a ZNS-tailored file system instead of an interface.

Storage backend of RocksDB for ZNS SSDs: Bjørling et al. [9] claim advantages of ZNS SSDs and propose ZenFS for RocksDB [14] on ZNS SSDs. Im et al. [21] and ZenFS+ [46] are enhanced versions of ZenFS as backend IO engines to maximize the parallelism of ZNS SSDs. WALTZ [32] is also an enhanced version of ZenFS to provide tight tail latency by leveraging ZONE APPEND commands. WA-Zone [37] suggests wear-aware zone management to balance inter-zone and intra-zone wear considering the access pattern of LSM-tree. These studies [9, 21, 32, 37, 46] are in line with our study in terms of improving the performance on ZNS SSD. Meanwhile, Z-LFS is a general and ZNS-tailored log-structured file system to run any application instead of specific applications.

Linux kernel subsystems on ZNS SSDs: ZNSwap [8] is a swap subsystem optimized for ZNS SSDs. It introduces a host-side GC mechanism which is co-designed with the swap logic to reduce GC overheads and improve the performance. RAIZN [26] is a logical volume manager which exposes a ZNS SSD array as a single ZNS interface to applications. BIZA [58] is also an interface for ZNS arrays to further improve endurance and performance by exploiting the zone random write area and internal parallelisms within ZNS SSDs. Our study is in line with them [8, 26, 58] in terms of investigating a Linux kernel component while we target designing a suitable file system for ZNS SSDs.

File systems for ZNS SSDs: ZoneFS [4, 28] is a simple file system exposing each zone of a ZNS SSD as a file. OPRW [30] leverages the write pointer of ZNS SSDs to accelerate the `fsync()` performance. Our work is in line with these works [4, 28, 30] in terms of supporting or optimizing file systems for ZNS SSDs. On the other hand, we redesign the metadata of LFS to enhance its compatibility with ZNS SSDs by considering LFS features on ZNS SSD. Furthermore, we efficiently scale the active zones of log streams to maximize the internal parallelism of ZNS SSD.

8 Conclusion

This paper presents a ZNS-tailored log-structured file system called Z-LFS for commodity ZNS SSDs. Z-LFS enables standalone usage of ZNS SSDs by an append-only metadata approach, leveraging the unique characteristics of LFS metadata to minimize write amplification. Z-LFS efficiently utilizes active zones for log streams via speculative log stream management. Z-LFS resolves conflicts between zones sharing internal resources within ZNS SSD via conflict-aware zone allocation. In our evaluations, Z-LFS achieves higher performance by up to 33.44× and 3.5× compared with F2FS and a state-of-the-art interface for commodity ZNS SSDs, respectively.

Acknowledgments

We sincerely thank our shepherd, Ming-Chang Yang, and the anonymous reviewers for their invaluable feedback. This work was supported by the National Research Foundation of Korea (NRF) (No. RS-2025-00554650) and in part by the Brain Korea 21 (BK21) FOUR Intelligence Computing funded by NRF (No. NRF-4199990214639) (Corresponding Author: Yongseok Son).

References

[1] File systems. https://zonedstorage.io/docs/linux/fs.

[2] Filebench - A Model Based File System Workload Generator. https://github.com/filebench/filebench.

[3] Samsung PM1733 NVMe SSD. https://download.semiconductor.samsung.com/resources/brochure/PM1733%20NVMe%20SSD.pdf.

[4] Zonefs. https://www.kernel.org/doc/html/v5.17/filesystems/zonefs.html.

[5] Samsung pm1731a review from sth., 2022. https://news.samsung.com/global/samsung-introduces-its-first-zns-ssd-with-maximized-user-capacity-and-enhanced-lifespan.

[6] Abdul R Abdurrab, Tao Xie, and Wei Wang. Dloop: A flash translation layer exploiting plane-level parallelism. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, pages 908–918. IEEE, 2013.

[7] Hanyeoreum Bae, Jiseon Kim, Miryeong Kwon, and Myoungsoo Jung. What you can't forget: exploiting parallelism for zoned namespaces. In *Proceedings of the 14th ACM Workshop on Hot Topics in Storage and File Systems*, pages 79–85, 2022.

[8] Shai Bergman, Niklas Cassel, Matias Bjørling, and Mark Silberstein. Znswap: Un-block your swap. *ACM Transactions on Storage*, 19(2):1–25, 2023.

[9] Matias Bjørling, Abutalib Aghayev, Hans Holmberg, Aravind Ramesh, Damien Le Moal, Gregory R Ganger, and George Amvrosiadis. Zns: Avoiding the block interface tax for flash-based ssds. In *USENIX Annual Technical Conference*, pages 689–703, 2021.

[10] Chandranil Chakraborttii, Vikas Sinha, and Heiner Litz. Ssd qos improvements through machine learning. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 511–511, 2018.

[11] Feng Chen, Binbing Hou, and Rubao Lee. Internal parallelism of flash memory-based solid-state drives. *ACM Transactions on Storage (TOS)*, 12(3):1–39, 2016.

[12] Feng Chen, Rubao Lee, and Xiaodong Zhang. Essential roles of exploiting internal parallelism of flash memory based solid state drives in high-speed data processing. In *2011 IEEE 17th International Symposium on High Performance Computer Architecture*, pages 266–277. IEEE, 2011.

[13] Jungsik Choi, Jaewan Hong, Youngjin Kwon, and Hwansoo Han. Libnvmmio: Reconstructing software IO path with Failure-AtomicMemory-Mapped interface. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 1–16, 2020.

[14] Facebook RocksDB team. A persistent key-value store for fast storage environments, 2021. https://rocksdb.org.

[15] Jin Yong Ha and Heon Young Yeom. zceph: Achieving high performance on storage system using small zoned zns ssd. In *Proceedings of the 38th ACM/SIGAPP Symposium on Applied Computing*, pages 1342–1351, 2023.

[16] XYHR Haas and X Hu. The fundamental limit of flash random write performance: Understanding, analysis and performance modelling. *IBM Research Report, 2010/3/31, Tech. Rep*, 2010.

[17] Kyuhwa Han, Hyunho Gwak, Dongkun Shin, and Jooyoung Hwang. Zns+: Advanced zoned namespace interface for supporting in-storage zone compaction. In *OSDI*, pages 147–162, 2021.

[18] Jian Huang, Anirudh Badam, Laura Caulfield, Suman Nath, Sudipta Sengupta, Bikash Sharma, and Moinuddin K Qureshi. Flashblox: Achieving both performance isolation and uniform lifetime for virtualized ssds. In *FAST*, volume 17, pages 375–390, 2017.

[19] J Hwang. Towards even lower total cost of ownership of data center it infrastructure. In *Proceedings of the NVRAMOS Workshop, Jeju, Korea*, pages 24–26, 2019.

[20] Joo-Young Hwang, Seokhwan Kim, Daejun Park, Yong-Gil Song, Junyoung Han, Seunghyun Choi, Sangyeun Cho, and Youjip Won. ZMS: Zone abstraction for mobile flash storage. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*, pages 173–189, 2024.

[21] Minwoo Im, Kyungsu Kang, and Heonyoung Yeom. Accelerating rocksdb for small-zone zns ssds by parallel i/o mechanism. In *Proceedings of the 23rd International Middleware Conference Industrial Track*, pages 15–21, 2022.

[22] J.Axboe. Fio benchmark. https://github.com/axboe/fio.

[23] Dongwon Kang, Dawoon Jung, Jeong-Uk Kang, and Jin-Soo Kim. μ-tree: An ordered index structure for nand flash memory. In *Proceedings of the 7th ACM & IEEE international conference on Embedded software*, pages 144–153, 2007.

[24] Woon-Hak Kang, Sang-Won Lee, Bongki Moon, Yang-Suk Kee, and Moonwook Oh. Durable write cache in flash memory ssd for relational and nosql databases. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, 2014.

[25] Juwon Kim, Minsu Kim, Muhammad Danish Tehseen, Joontaek Oh, and YouJip Won. IPLFS: Log-Structured file system without garbage collection. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 739–754, 2022.

[26] Thomas Kim, Jekyeom Jeon, Nikhil Arora, Huaicheng Li, Michael Kaminsky, David G Andersen, Gregory R Ganger, George Amvrosiadis, and Matias Bjørling.

Raizn: Redundant array of independent zoned namespaces. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 660–673, 2023.

[27] Ryusuke Konishi, Yoshiji Amagai, Koji Sato, Hisashi Hifumi, Seiji Kihara, and Satoshi Moriai. The linux implementation of a log-structured file system. *ACM SIGOPS Operating Systems Review*, 40(3):102–107, 2006.

[28] Damien Le Moal and Ting Yao. zonefs: Mapping posix file system interface to raw zoned block device accesses. 2020.

[29] Changman Lee, Dongho Sim, Joo Young Hwang, and Sangyeun Cho. F2fs: A new file system for flash storage. In *FAST*, volume 15, pages 273–286, 2015.

[30] Euidong Lee, Ikjoon Son, and Jin-Soo Kim. An efficient order-preserving recovery for f2fs with zns ssd. In *Proceedings of the 15th ACM Workshop on Hot Topics in Storage and File Systems*, pages 116–122, 2023.

[31] Eunji Lee, Seunghoon Yoo, Jee-Eun Jang, and Hyokyung Bahn. Shortcut-jfs: A write efficient journaling file system for phase change memory. In *Mass Storage Systems and Technologies (MSST), 2012 IEEE 28th Symposium on*, pages 1–6. IEEE, 2012.

[32] Jongsung Lee, Donguk Kim, and Jae W Lee. Waltz: Leveraging zone append to tighten the tail latency of lsm tree on zns ssd. *Proceedings of the VLDB Endowment*, 16(11):2884–2896, 2023.

[33] Minkyeong Lee, Dong Hyun Kang, Minho Lee, and Young Ik Eom. Improving read performance by isolating multiple queues in nvme ssds. In *Proceedings of the 11th International Conference on Ubiquitous Information Management and Communication*, pages 1–6, 2017.

[34] Xiaojian Liao, Youyou Lu, Erci Xu, and Jiwu Shu. Max: A Multicore-Accelerated file system for flash storage. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 877–891, 2021.

[35] Renping Liu, Zhenhua Tan, Yan Shen, Linbo Long, and Duo Liu. Fair-zns: Enhancing fairness in zns ssds through self-balancing i/o scheduling. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2022.

[36] G Lockwood. Ior and mdtest (2019), 2019. https://github.com/hpc/ior.git.

[37] Linbo Long, Shuiyong He, Jingcheng Shen, Renping Liu, Zhenhua Tan, Congming Gao, Duo Liu, Kan Zhong, and Yi Jiang. Wa-zone: Wear-aware zone management optimization for lsm-tree on zns ssds. *ACM Transactions on Architecture and Code Optimization*, 21(1):1–23, 2024.

[38] Ricardo Macedo, Yusuke Tanimura, Jason Haga, Vijay Chidambaram, José Pereira, and João Paulo. PAIO: General, portable I/O optimizations with minor application modifications. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*, pages 413–428, 2022.

[39] Bo Mao, Suzhen Wu, and Lide Duan. Improving the ssd performance by exploiting request characteristics and internal parallelism. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(2):472–484, 2017.

[40] Avantika Mathur, Mingming Cao, Suparna Bhattacharya, Andreas Dilger, Alex Tomas, and Laurent Vivier. The new ext4 filesystem: current status and future plans. In *Proceedings of the Linux symposium*, volume 2, pages 21–33, 2007.

[41] Changwoo Min, Sanidhya Kashyap, Steffen Maass, and Taesoo Kim. Understanding manycore scalability of file systems. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 71–85, 2016.

[42] Jaehong Min, Chenxingyu Zhao, Ming Liu, and Arvind Krishnamurthy. eZNS: An elastic zoned namespace for commodity ZNS SSDs. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 461–477, 2023.

[43] Jaehong Min, Chenxingyu Zhao, Ming Liu, and Arvind Krishnamurthy. ezns: Elastic zoned namespace for enhanced performance isolation and device utilization. *ACM Transactions on Storage*, 20(3):1–41, 2024.

[44] NVM Express Workgroup. Nvm express zoned namespaces command set 1.1b, 2021. https://nvmexpress.org/developers/nvme-command-set-specifications/.

[45] Gijun Oh, Junseok Yang, and Sungyong Ahn. Efficient key-value data placement for zns ssd. *Applied Sciences*, 11(24):11842, 2021.

[46] Myounghoon Oh, Seehwan Yoo, Jongmoo Choi, Jeongsu Park, and Chang-Eun Choi. Zenfs+: Nurturing performance and isolation to zenfs. *IEEE Access*, 11:26344–26357, 2023.

[47] Yongseok Oh, Eunsam Kim, Jongmoo Choi, Donghee Lee, and Sam H Noh. Optimizations of lfs with slack space recycling and lazy indirect block update. In *Proceedings of the 3rd Annual Haifa Experimental Systems Conference*, pages 1–9, 2010.

[48] Alma Riska and Erik Riedel. Disk drive level workload characterization. In *USENIX Annual Technical Conference, General Track*, volume 2006, pages 97–102, 2006.

[49] Mendel Rosenblum and John K Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems (TOCS)*, 10(1):26–52, 1992.

[50] Dongjoo Seo, Ping-Xiang Chen, Huaicheng Li, Matias Bjørling, and Nikil Dutt. Is garbage collection overhead gone? case study of f2fs on zns ssds. 2023.

[51] Hojin Shin, Myounghoon Oh, Gunhee Choi, and Jongmoo Choi. Exploring performance characteristics of zns ssds: Observation and implication. In *2020 9th Non-Volatile Memory Systems and Applications Symposium (NVMSA)*, pages 1–5. IEEE, 2020.

[52] Theano Stavrinos, Daniel S Berger, Ethan Katz-Bassett, and Wyatt Lloyd. Don't be a blockhead: zoned namespaces make work on conventional ssds obsolete. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, pages 144–151, 2021.

[53] Qiuping Wang and Patrick P. C. Lee. Zapraid: Toward high-performance raid for zns ssds via zone append. APSys '23, page 24–29, New York, NY, USA, 2023. Association for Computing Machinery. doi:10.1145/3609510.3609810.

[54] Randolph Y Wang and Thomas E Anderson. xfs: A wide area mass storage file system. In *Proceedings of IEEE 4th Workshop on Workstation Operating Systems. WWOS-III*, pages 71–78. IEEE, 1993.

[55] Denghui Wu, Biyong Liu, Wei Zhao, and Wei Tong. Zn-skv: Reducing data migration in lsmt-based kv stores on zns ssds. In *2022 IEEE 40th International Conference on Computer Design (ICCD)*, pages 411–414. IEEE, 2022.

[56] Jingpei Yang, Ned Plasson, Greg Gillis, Nisha Talagala, and Swaminathan Sundararaman. Don't stack your log on my log. In *2nd Workshop on Interactions of NVM/Flash with Operating Systems and Workloads (INFLOW 14)*, 2014.

[57] Junseok Yang, Seokjun Lee, and Sungyong Ahn. Selective power-loss-protection method for write buffer in zns ssds. *Electronics*, 11(7):1086, 2022.

[58] Shushu Yi, Shaocong Sun, Li Peng, Yingbo Sun, Ming-Chang Yang, Zhichao Cao, Qiao Li, Myoungsoo Jung, Ke Zhou, and Jie Zhang. Biza: Design of self-governing block-interface zns afa for endurance and performance. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, SOSP '24, page 313–329, New York, NY, USA, 2024. Association for Computing Machinery. doi:10.1145/3694715.3695953.

[59] Jiacheng Zhang, Jiwu Shu, and Youyou Lu. ParaFS: A Log-Structured file system to exploit the internal parallelism of flash devices. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 87–100, 2016.

[60] Runyu Zhang, Duo Liu, Xianzhang Chen, Xiongxiong She, Chaoshu Yang, Yujuan Tan, Zhaoyan Shen, and Zili Shao. Loffs: A low-overhead file system for large flash memory on embedded devices. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2020.

[61] Yanbo Zhou, Erci Xu, Li Zhang, Kapil Karkra, Mariusz Barczak, Wayne Gao, Wojciech Malikowski, Mateusz Kozlowski, Łukasz Łasek, Ruiming Lu, et al. Csal: the next-gen local disks for the cloud. In *Proceedings of the Nineteenth European Conference on Computer Systems*, pages 608–623, 2024.

ARTIFACT EVALUATED
usenix ASSOCIATION
AVAILABLE

CLONE: Customizing LLMs for Efficient Latency-Aware Inference at the Edge

Chunlin Tian, Xinpeng Qin, Kahou Tam, Li Li†, Zijian Wang, Yuanzhe Zhao, Minglei Zhang, and Chengzhong Xu

University of Macau

Abstract

Deploying large language models (LLMs) on edge devices is crucial for delivering fast responses and ensuring data privacy. However, the limited storage, weight, and power of edge devices make it difficult to deploy LLM-powered applications. These devices must balance latency requirements with energy consumption and model accuracy. In this paper, we first quantify the challenges of deploying LLMs on off-the-shelf edge devices and then we present CLONE, an in-depth algorithm-hardware co-design at both the model- and system-level that intelligently integrates real-time, energy optimization while maintaining robust generality. In order to maximize the synergistic benefits of these algorithms in always-on and intermediate edge computing settings, we specialize in a 28nm scalable hardware accelerator system. We implement and extensively evaluate CLONE on two off-the-shelf edge platforms. Experiments show that CLONE effectively accelerates the inference process up to 11.92×, and saves energy up to 7.36×, while maintaining high-generation.

1 Introduction

Large language models (LLMs) [19, 80, 96, 109, 132] are reshaping artificial intelligence for their remarkable performance to comprehend human language and handle language-related tasks [120, 129, 136]. Though born from the cloud, deploying LLMs on edge devices such as personal PC, smartphones, robots, and even IoT devices is becoming an important trend [81, 119]. First, on-device LLM can be widely used to support different kinds of applications, including chatbots [6, 114], robotics [61, 162], and autonomous vehicles [144]. Moreover, directly accessing the LLM on the edge not only preserves data privacy but also can provide instant service without relying on a stable internet connection. Despite the potential benefits, deploying LLMs on commercial-off-the-shelf (COTS) edge devices faces stringent space, weight, and power (SWaP) constraints [10, 138] due to the billion-parameter size coupled with intensive computing costs. For instance, the widely used Llama-7B model [132] requires approximately 14GB of memory for inference, even when utilizing 16-bit precision (FP16) format. However, the available RAM on typical edge devices only ranges from 4GB to 12GB [125]. Furthermore, inferring a single token with Llama-7B requires approximately 14 TFLOPs for an 11-token prompt [25], which is roughly 360× the 39 GFLOPs needed by VGG-19 to process a single image with an input resolution of 224 × 224 [36]. Moreover, LLMs' inference process demands substantial energy consumption. For instance, GPT-3 [31] consumes 300 J/response [20] on an NVIDIA A100 GPU, which is 400× the 0.75 J/response required by ResNet-50 [72]. Thus, intensive memory footprint, high computational demands, and significant energy costs pose severe bottlenecks to deploying LLMs at the edge.

Limitation of Prior Arts. To break the SwaP constraints, several optimization techniques have been proposed. Techniques such as model architecture search [54, 60, 88], quantization [26, 55, 57, 91, 143], and pruning [27, 30, 32, 84] help to reduce the size of models while preserving their performance. However, these methods typically focus on optimizing the model itself and may not fully address the system-level trade-offs related to storage and weight. Additionally, recent improvements in LLM compilers and software stacks [56, 99, 113, 128] have facilitated the adoption of co-processors to sustain high performance by leveraging co-processors or near-sensor processing alongside GPUs [4, 44, 57, 74, 74, 86, 107, 111, 118, 122, 148, 159]. However, they incur additional computational/communication overhead that significantly reduces the edge device lifecycle and interferes with the smooth execution of other applications. To address power constraints, Dynamic Voltage and Frequency Scaling (DVFS) [10, 47, 48, 65, 112] has been widely adopted to adjust the power usage of processors by dynamically changing their voltage and frequency. However, most existing approaches are designed for discriminative CNN or RNN and treat the entire network as a black box during tuning. Generative LLMs have not undergone the necessary scrutiny due to the auto-regressive inference schemes and the heterogeneity

of stochastic requests and generated outputs. *Thus, an edge-tailored LLM system that intelligently coordinates model- and system-level optimizations, balancing energy efficiency, latency, and model performance is urgently required.*

Challenge. Designing such a customization system is not straightforward and faces the following critical challenges. First, the "billion-parameter" poses a significant barrier to deployment on resource-constrained edge devices due to the high memory footprint and computation cost. How to customize the LLM according to the hardware profile of a specific edge device to effectively meet the memory constraint and reduce the computing latency while ensuring the generating capability is the first critical challenge. In addition, when deploying customized LLMs on edge devices, LLM-powered applications feature stochastic and dynamic I/O, unlike traditional models with stable inputs. Moreover, tailored layers introduce unique approximation characteristics, while interference from co-running applications during mobile execution can cause runtime variance. Therefore, designing an efficient system-level controller to meet latency targets while optimizing energy consumption is another challenge. Furthermore, model-level (accuracy/latency) and system-level (energy/latency) optimizations in isolation can be naive, often resulting in unnecessary energy consumption or reduced accuracy. Effectively coordinating these optimizations to balance accuracy, latency, and energy efficiency is the third challenge.

In this work, we present CLONE, a comprehensive system that intelligently customizes LLM for efficient latency-aware inference on commercial-off-the-shelf edge devices. It adopts a hierarchical structure to jointly handle model- and system-level optimization in a unified manner to consider real-timeliness, model accuracy, and energy efficiency jointly. Specifically, CLONE consists of two main phases: offline device-specific model tailoring and online latency-aware system optimization. Within the offline tailoring process, for a specific device, CLONE reformulates the tailoring of target LLMs for a specific device as a generative task within a continuous representation space. Using an encoder-evaluator-decoder architecture, it generates optimal pruning configurations through gradient-based optimization, effectively bridging the gap between resource constraints and model performance. To support diverse edge applications [6, 138, 146], CLONE performs parameter-efficient fine-tuning on the customized LLM using multiple plug-and-play Low-Rank Approximation (LoRA) [52] adapters, ensuring adaptability and optimized performance. Once the customized model is ported to edge devices, during online inference, CLONE employs a Mixture-of-Experts (MoE) [59, 131] router to dynamically integrate optimal LoRA adapters, enabling more precise responses to stochastic, complex, or mixed-task end-user requests. At the system level, CLONE applies Dynamic Voltage and Frequency Scaling (DVFS) to the per-token, autoregressive inference process of LLMs, optimizing the supply voltage (V_{DD}) and operating frequency (F_{req}) to minimize energy consumption while meeting real-time constraints. Recognizing the LLM layer characteristics, especially post-pruned uneven parameter, CLONE enhances the granularity of DVFS adjustments at layer boundaries. This provides greater flexibility and adaptability for power-sensitive systems compared to traditional workload-level (whole model) black-box optimizations. While existing vanilla DVFS on edge devices are typically discrete, CLONE introduces a learning-based DVFS approach to reduce budget gaps that maximize system efficiency and model performance. To fully harness the benefits of these technologies, we have developed a specialized 28nm scalable hardware accelerator system. It contains a LoRA Processing Unit (LPU) for hot-swapping adapters, enabling dynamic model performance adjustments through dedicated data paths, and a Special Function Unit (SFU) designed for fine-grained and continuous DVFS adjustments equipped with a fast-switching low-dropout (LDO) voltage regulator and an all-digital phase-locked loop (ADPLL). The LPU supports the runtime MoE router algorithm to handle stochastic requests, eliminating bottlenecks of general-purpose processors and boosting computational efficiency. The SFU implements the learning-based DVFS algorithm to effectively manage runtime variance and swiftly achieve the desired V_{DD} and F_{req}. Specifically, we make the following key contributions:

- We propose CLONE, an efficient software-hardware co-design system for customized LLM inference on resource-constrained edge devices, where the model- and system-level coordinators cooperate to intelligently balance the inference speed and energy efficiency while maintaining robust generative performance.
- We design hierarchical offline/online optimization phases that effectively coordinate customization by addressing static model weights, hardware characteristics, stochastic I/O patterns, and runtime variance. Additionally, a 28nm accelerator with specialized datapaths supports these algorithms for optimized performance.
- To evaluate the effectiveness of CLONE, we conduct extensive experiments based on commercial-off-the-shelf edge devices, representative LLMs and benchmarks.

2 Background and Related Work

2.1 Large Language Models

LLM Architecture. In contrast to traditional deep neural networks (DNNs) and convolutional neural networks (CNNs) [38, 76], which integrate diverse types of layers (e.g., convolutional (CONV), fully connected (FC), recurrent (RC), pooling, etc.) designed for specific tasks, large language models (LLMs) predominantly consist of a uniform stack of transformer decoder layers. For instance, Llama-7B [132] adopts

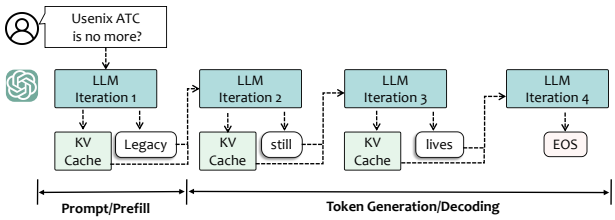


Figure 1: Overview of LLMs autoregressive inference.

a homogeneous architecture composed of 32 identical LlamaDecoderLayers. Each decoder layer encompasses two core components: LlamaAttention and LlamaMLP. Despite the structural uniformity across decoder layers, their contributions to model efficiency and effectiveness vary significantly [26, 137]. Consequently, optimizing the inference execution of LLMs necessitates a detailed analysis of the individual impact of each layer (§3.1).

LLMs Inference. LLMs process a structured sequence involving multiple forward passes through the model to sequentially generate each output token. Figure 1 shows the inference process with a simple example. Typically, this process mainly contains two stages [3, 75, 118]. **1) Pre-fill** takes a prompt sequence and generates the key-value (KV) cache for each Transformer layer of LLM. Upon receiving the prompt *"Usenix ATC is no more?"*, the tokenizer embeds input as tokens, denoted as $X_{in} \in \mathbb{R}^{n \times d}$, where d is the hidden size and n is the length of input token. Then, the LLM handles all input tokens in parallel during a single forward iteration to generate a KV cache. The output of attention is sent to MLP to generate the first output token *"Legacy"*. Large-scale matrix multiplications are required to generate the KV cache, which makes the pre-fill computing intensive. **2) Decoding** utilizes and updates the KV cache to generate tokens step-by-step. Following the generation of the first token, the LLM leverages the KV caches prepared earlier and adds new information to them. The creation of each new token is influenced by the tokens generated before it. During each token generation, for the input $X_{dec} \in \mathbb{R}^{1 \times d}$, attention layers load the previously stored KV cache, and new KV pairs are computed and concatenated to the existing cache. The output of the last decoder layer is sent to the final prediction layer to predict the next token sequentially. It executes iteratively until an End of Sequence (EOS) token is encountered or a predefined termination criterion is met. Unlike traditional models with fixed input formats and structured workflows [38, 76], LLM inputs and outputs are highly non-deterministic. This stems from the diverse, open-ended nature of user prompts, which vary widely in structure, intent, and context [15, 87, 89]. Additionally, autoregressive token generation is inherently probabilistic, driven by sampling methods [50, 111, 115, 132] and variability in training datasets [11, 13, 134], making outputs context-sensitive.

2.2 Bottlenecks of Deploying Edge LLMs

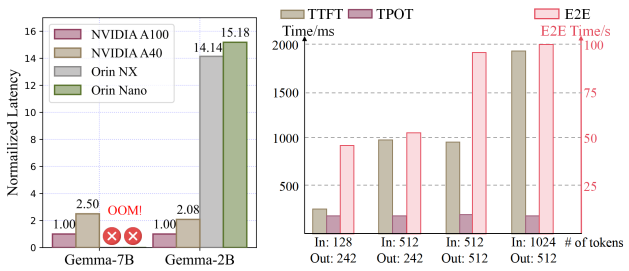
Table 1 lists specifications for server-level and edge-level processors commonly used for ML workloads, highlighting resource collapse. Despite the potential benefits, including privacy preservation and instant responses without depending on a stable internet connection [42, 105], deploying LLMs on the edge faces the following critical bottlenecks.

1) High Memory Footprint. The main contributors of "billion-parameter" LLMs are model weights (memory is occupied by the model parameters) and KV cache (memory is occupied by the caching of self-attention tensors to avoid redundant computation). For example, Llama-7B in 16-bit precision requires approximately 14GB memory ($7B \times$ sizeof(FP16)). Its architecture with 32 layers, 32 heads per layer, and a head dimension of 128 incurs a memory cost of 0.5MB per token, accounting for K and V matrices. Consequently, processing 4096 tokens demands 2GB, limiting the size of models that can be deployed on edge devices with 4–12GB memory [125] and often causing Out-of-Memory (OOM) errors.

Table 1: Popular ML hardware specifications.

GPU Types	Peak Perf.	Memory	Bandwidth	Peak Power
Server-level				
NVIDIA A100	312 TFLOPS	80GB	1935 GB/s	300W
NVIDIA A40	149.7 TFLOPS	48 GB	696 GB/s	300W
Edge-level				
Jetson Orin NX	100 TOPS	16GB	102.4GB/s	25W
Jetson Orin Nano	40 TOPS	8GB	68 GB/s	15W

2) High Inference Latency. Inference latency is a crucial metric for mobile optimization because if the latency of a service exceeds the human-acceptable thresholds (e.g., 33.3 ms for a 30 FPS video frame rate [28, 161] or 50 ms for interactive applications [29, 92]), end-users will abandon the service [161]. To evaluate inference QoE for LLMs, there are three main metrics: 1) *Time To First Token (TTFT)* latency from input to the output of the first token (prefill). Low TTFT means fast response, which is essential for user experience in real-time interactions. 2) *Time Per Output Token (TPOT)* latency to track the auto-regressive process of each token generated serially. TPOT refers to how each user perceives the "speed" of the model. 3) *End-to-end (E2E)* latency, the overall time it takes for the model to generate the full response for a user, denoted as: $E2E = (TTFT) + (TPOT) * N$, where N is the generated token number. Figure 2 (a) illustrates the latency across different processors running the same LLMs, representative Gemma-2B/7B [96], on the Wikitext2 [95] dataset. While Gemma-7B is OOM for edge device. For the same user request on Gemma-2B, the Orin Nano experiences 15.18× more latency than the NVIDIA A100, significantly impacting the Quality of end-user Experience (QoE) [53, 68, 69, 90]. Additionally, LLMs operate with an auto-regressive and probabilistic inference process [121]. Figure 2 (b) shows



(a) Server/edge latency. (b) Latency breakdown.

Figure 2: LLMs inference latency analysis. (a) Latency efficiency of two common LLMs (Gemma-2B/7B) inference use cases over server-level and edge-level processors. (b) Inference latency with different prompt and output tokens.

as prompt sizes increase (128 → 512 → 1024) inference Gemma-2B on Orin NX. We can observe that the number of pre-filled tokens increases significantly (255→1935ms) due to parallel processing capabilities. Conversely, the TPOT remains relatively stable (180∼200ms) due to serial decoding generation. However, the E2E is more than 100s, which fails to serve the end-user demands, indicating opportunities for optimizing practical inference performance.

3) High Energy Consumption. In server-level GPU deployments, it is typically assumed that these units are plugged into wall power due to the substantial energy demands of LLMs inference [4, 64, 74, 107, 111, 141]. However, this assumption does not hold for edge-level devices such as robots and autonomous vehicles, where energy availability is severely restricted [66]. In contrast to conventional on-device tasks, LLM inference consumes an order of magnitude more energy. For example, a Google search driven by a large AI model expends 8.9 watt-hours (Wh) of energy, approximately 30 times the 0.3 Wh required by a standard Google search [124]. Importantly, "Scaling Laws" [21, 63] suggest that as model parameters scale up, there is a corresponding increase in both performance and energy consumption. Transitioning from Gemma-2B to 7B, for instance, boosts accuracy from 42.3% to 64.3% on the MMLU benchmark [46]—a comprehensive evaluation platform—at the cost of tripling energy usage. Consequently, there is a pressing need to optimize the energy efficiency of LLM inference at edge to meet QoE.

3 Motivation

In this section, we present key observations from model-intrinsic dimensions (layer characteristics (§3.1)), stochastic I/O (§3.2)) and system factors (hardware and runtime variance (§3.3)) for practical LLM inference on edge devices. The design space is analyzed across three critical axes: generation quality, latency, and energy efficiency.

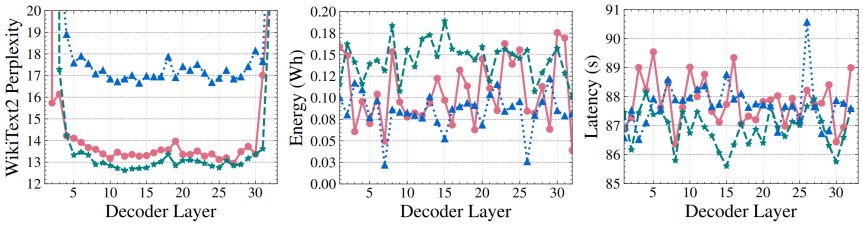
3.1 Heterogeneity of Model Characteristics

In order to investigate the contribution of different decoder layers, we conduct an in-depth analysis of the inherent sensitivity of the stacked transformer architecture across three critical dimensions: generative ability, energy efficiency, and latency. For general-purpose, we employ zero-shot perplexity (PPL) [14, 96, 132], a common metric, to evaluate generative abilities, lower PPL indicates higher model adeptness in predicting sequence tokens. Figure 3 (a) presents the PPL on the WikiText2 [95] dataset after removing specific decoder layers in the Llama-7B, Llama2-7B [133], and Vicuna-7B [160] models. The results show that the front and back layers have higher PPL values than the middle layers. Because the front layers play a critical role in feature extraction, while the back layers significantly influence output generation, both contribute markedly to model performance. while LLMs maintain a homogeneous layer structure, the generative impact varies by layer due to input and output data stream heterogeneity. To evaluate the system effectiveness, we infer LLMs on WikiText2, monitoring energy consumption and duration using CodeCarbon [24]. Figure 3 (b, c) shows that removing different decoder layers yields non-uniform changes in end-to-end energy and latency. This suggests that, beyond nominal parameter counts, each layer contributes heterogeneously to practical resource usage. We attribute this to (i) varying sequence-length–dependent workloads, (ii) hardware-level cache and parallel-sync effects, and (iii) non-linear inter-layer interactions typical of deep networks.

► *Motivation 1*: LLM layers contribute unevenly to effectiveness and efficiency, underscoring opportunities for model customization by pruning non-essential components and system optimization through fine-grained layer-wise tuning.

3.2 Heterogeneity of Stochastic Input/Output

As discussed at §2.1, inference serving systems exhibit significant non-deterministic due to the diversity of input prompts, and LLM auto-regressive output exhibit distinct execution behaviors. **Case study.** As shown in Figure 5 (a), we first illustrate the distribution of prompt input and generated output tokens on an Azure LLM inference services trace [111]. These long-tail patterns and the unpredictability of token generation lengths highlight the need for dynamic resource allocation to manage varying computational demands. **Performance impact.** To analyze the inherent patterns of mixed-task end-user requests, we visualized the task embedding similarities of the Flanv2 dataset [135] using a heatmap, as shown in Figure 4. Each axis enumerates the individual subtasks within the dataset, and each matrix cell quantifies the pairwise correlation between subtasks. The results reveal significant data heterogeneity across tasks. **System impact.** Otherwise, requests of different input and output lengths possess different compute and energy characteristics. As shown in Figure 5



(a) Generative-ability (b) Energy (c) Latency

Figure 3: Layer sensitivity analysis: removing specific layers one-by-one to measure each layer's impact on Llama-7B (●), Llama2-7B (⋆), and Vicuna-7B (▲).

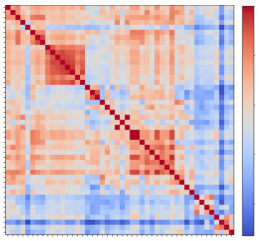
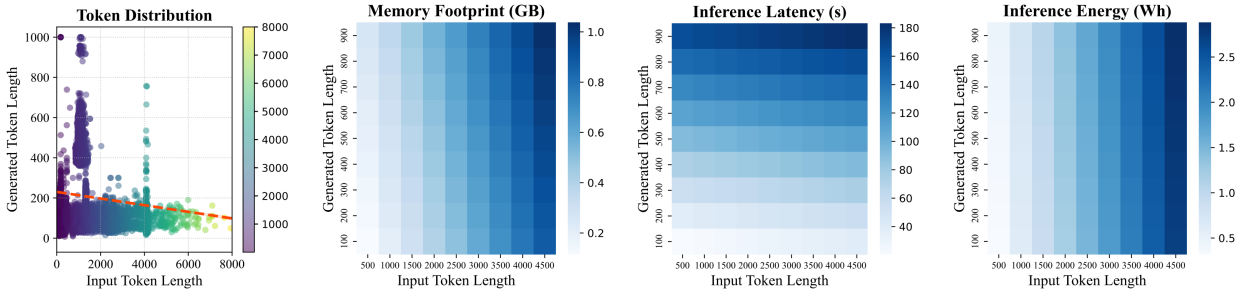


Figure 4: Task embedding similarity heatmap.



(a) Practical inference traces. (b) Memory consumption. (c) Latency analysis. (d) Energy consumption.

Figure 5: Comprehensive analysis of LLM inference system using Gemma-2B on NVIDIA Orin NX. (a) Analysis of practical LLM inference traces., while (b-d) highlight memory, latency, and energy consumption w.r.t. input and generated length.

(b-d), we measure Gemma-2B [96] on Nvidia Orin NX with different input and generated length to evaluate the memory footprint, latency and energy consumption. Memory consumption is influenced by both static model parameters and the dynamic KV cache, which grows as the number of tokens to be processed increases. E2E latency is primarily impacted during the decoding phase, as it is step-by-step, whereas the prefill phase computes all prompts in parallel. Consequently, an increase in generated tokens significantly impacts overall latency. Energy consumption is higher during the prefill phase due to large-scale matrix operations, while the decoding phase, processing one token at a time, has lower power demands. Inefficient system performance degrades QoE, making it critical to meet diverse LLM request demands without compromising output quality.

► *Motivation 2*: Stochastic I/O with mixed-task requests underscores the need for precise inference services to improve model performance. Similarly, significant resource slack suggests potential for fine-grained, LLM-specific system optimizations to improve utilization efficiency.

3.3 Heterogeneity of Runtime Platform

Practical system heterogeneity encompasses performance and energy variations arising from platform diversity (static heterogeneity, Table 1), diverse LLM application demands, and interference from co-running applications (dynamic heterogeneity). Table 2 lists LLM requests from various applications or tasks, each with distinct QoE demands [83, 130, 139, 140, 142, 147, 152, 161]. To investigate the impact of edge device hardware and concurrent running apps, we deployed the Gemma-2B model on two NVIDIA platforms—Jetson Orin NX and Orin Nano—and processed a workload comprising 128 prompt tokens and 242 generated tokens (the first scenario in Figure 2(b)). We profiled the resulting inference behavior, and Figure 6 reports the TTFT, TPOT, and E2E inference latency for both devices, alongside the corresponding performance degradation observed in a foreground web-search application [78, 161]. We find that 1) Due to the varying processing capabilities of different edge devices, the response time to the same end-user request can differ significantly. For instance, when both are in an idle state, the Orin NX, with its superior computational power, provides more timely feedback (TTFT: 255; TPOT: 198 ms) compared to the Orin Nano (TTFT: 268ms; TPOT: 231ms). 2) Resource contention caused by user interaction with the concurrent running app prominently slows down the model inference progress. For instance, the E2E inference latency on Orin NX is 42.6s when there is no user interaction in the foreground. However, it increases to 61.4s with a website searching application running in the foreground. This highlights the impact of latency on user experience, as service abandonment occurs when latency

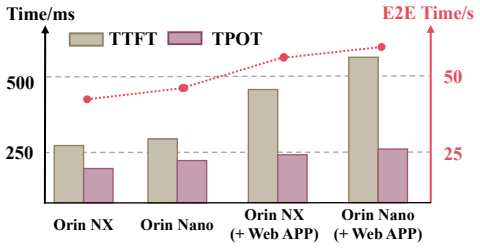


Figure 6: Impact of different hardware devices (Nano/NX) and concurrently running apps (web search) on edge LLMs inference.

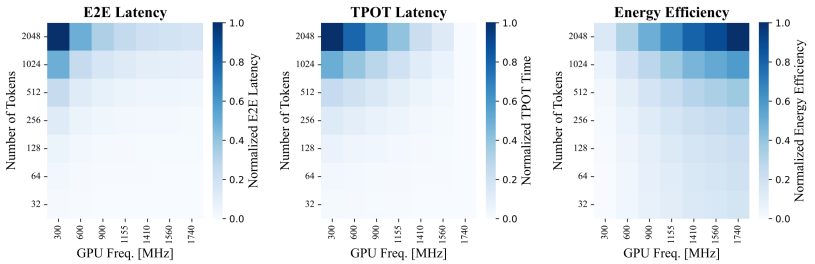


Figure 7: Impact of different token lengths and GPU frequencies on a) end-to-end (E2E) latency, b) time per output token (TPOT), and c) energy efficiency.

Table 2: SLOs of various LLM applications. Time To First Token (TTFT): refers to the latency from input to the first token output (prefill), while Time Per Output Token (TPOT) is the latency for each subsequent token (decode).

LLM Application	Service-Level Objective
Chatbot [5, 108]	Readable TTFT/TPOT
Search Engine [97]	Low TTFT, Medium TPOT
Event Logger [1, 58, 110]	Tolerable TTFT, Medium TPOT
Smart Reply [40, 98]	Low TTFT, Low TPOT
Code Generator [37]	Medium TTFT, High TPOT
Virtual Assistant [7, 39]	Low TTFT, Medium TPOT

exceeds the human-acceptable threshold [83, 161]. Figure 7 demonstrates that higher GPU frequencies reduce E2E latency and TPOT by accelerating token processing (a, b) while supporting energy-efficient control via frequency scaling (c). Dynamically adjusting processor frequencies based on workload size enhances efficiency with minimal performance impact, where the optimal frequency is determined by the LLM architecture and output length. Although DVFS [47, 48, 65] is commonly used to scale voltage and frequency for low-intensity workloads, existing schemes often rely on coarse-grained, black-box, workload-level adjustments with heuristic methods [10, 47, 48, 65, 112].

► ***Motivation 3:*** Hardware type and LLM application SLOs significantly impact end-user QoE, while resource preemption from foreground applications exacerbates runtime system heterogeneity. Accurate runtime estimation of application demands and device capabilities is crucial for guiding DVFS and optimizing LLM inference system efficiency.

4 CLONE: Design

4.1 System Overview

Figure 8 presents the architecture of CLONE which can be mainly divided into the following two phases: 1) offline device-specific tailor and 2) online latency-aware optimization. The overall workflow of CLONE can be represented as

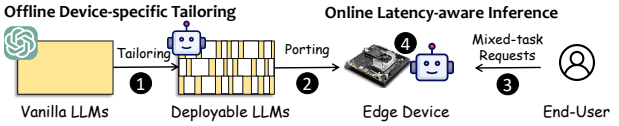


Figure 8: The overview of CLONE, including offline device-specific tailoring and online latency-aware inference.

the following main steps. ❶ CLONE leverages edge hardware profiles and LLM architecture using an encoder-evaluator-decoder framework to reframe LLM compression as a data-driven generative task. This approach addresses memory constraints, meets latency and energy requirements, and preserves generation performance. Plug-and-play LoRA adapters further fine-tune the tailored LLM for diverse edge applications. ❷ The deployable model is then ported onto the edge device. ❸ During runtime inference, to handle stochastic and mixed-task user requests, CLONE integrates a MoE router to dynamically select optimal LoRA adapters, enhancing model performance. ❹ At the system level, CLONE incorporates a learning-based DVFS controller to adjust device voltage (V_{DD}) and frequency ($Freq$) dynamically at layer boundaries during per-token autoregressive inference. This minimizes energy consumption while ensuring target latency is met. Furthermore, in order to fully leverage the synergistic advantages of these algorithms, we develop a specialized 28nm scalable hardware accelerator system designed to further boost energy efficiency through customized hardware integration.

4.2 Offline Device-specific Tailoring

According to *Motivation 1*, LLMs demonstrate considerable parameter redundancy, especially within their intermediate layers. This variance in contribution across layers not only affects performance but also system efficiency, facilitating tailored adaptations for edge deployment. To meet heterogeneous hardware constraints, CLONE redefines LLM pruning as a generative task, surpassing traditional manually designed discrete heuristic optimizations to identify and remove non-essential structures in a system-efficient manner. Specifically,

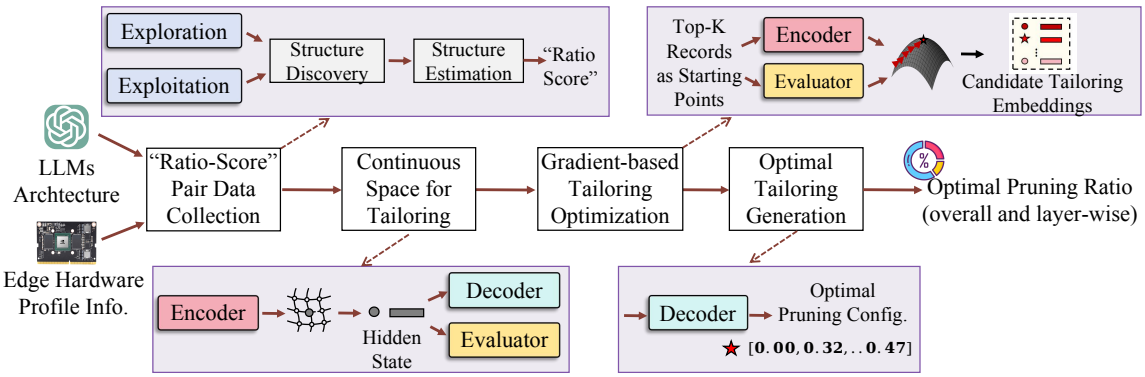


Figure 9: Hardware-aware tailor workflow. Four key components: "ratio-score" data collection, continuous space for tailoring, gradient-based tailoring optimization, and optimal tailoring generation.

it employs a data-driven tailor that identifies the optimal pruning configuration through gradient-based optimization within a continuous representation space. As illustrated in Figure 9, the generative tailor comprises four key modules:

1) *"Ratio-score" Data Collection.* Considering the memory constraints of the edge device and the dimensions of the model to be deployed, we initially establish the overall reduction ratio for the LLMs. For fine-grained optimization, we develop an exploration-exploitation strategy to determine layer-wise level pruning ratios r_i. This approach utilizes heuristic-based pruning methods [8, 30, 94] to produce high-quality ratios and incorporates random pruning ratios as part of the exploration process. In contrast to prior works [45, 55, 102] primarily focus on reducing the model size and computational overhead, CLONE integrates broader criteria including generation ability, inference latency, and energy cost. We define a holistic metric function f to characterize overall performance for a given r_i, aiming for optimization suitable for resource-constrained edge devices:

$$s_i = f(r_i) = \frac{1}{ppl_i} \times \left(\frac{E}{e_i}\right)^{\mathbf{1}(E<e_i)\times\alpha} \times \left(\frac{T}{t_i}\right)^{\mathbf{1}(T<t_i)\times\beta}, \quad (1)$$

where ppl is the zero-shot perplexity [14], quantifies the generative capabilities, with lower values indicating more precise model predictions. T and E denote the latency and energy budgets specific to edge, respectively. t_i and e_i represent the latency and energy consumption for a given ratio r_i. $\mathbf{1}(x)$ is an indicator that returns 1 if condition x holds, and 0 otherwise. Thus, configurations that exceed the thresholds T and E are penalized by developer-specified factors α and β, both set to 2 in our implementation. Finally, we obtain comprehensive ratio-score pairs, denoted as $\mathcal{P} = (r_i, s_i)$.

2) *Continuous Space for Tailoring.* Traditional approaches [8, 30, 94] often employ discrete pruning spaces, resulting in heuristic and incomprehensive pruning ratios. In order to characterize the continuous tailoring optimization space of edge devices, we implement an encoder-evaluator-decoder framework. This architecture includes a single-layer LSTM network [49] functioning as both encoder and decoder, alongside a feed-forward neural network serving as the evaluator. This setup effectively embeds ratio-score pairs (r_i, s_i) into a continuous representation space Θ.

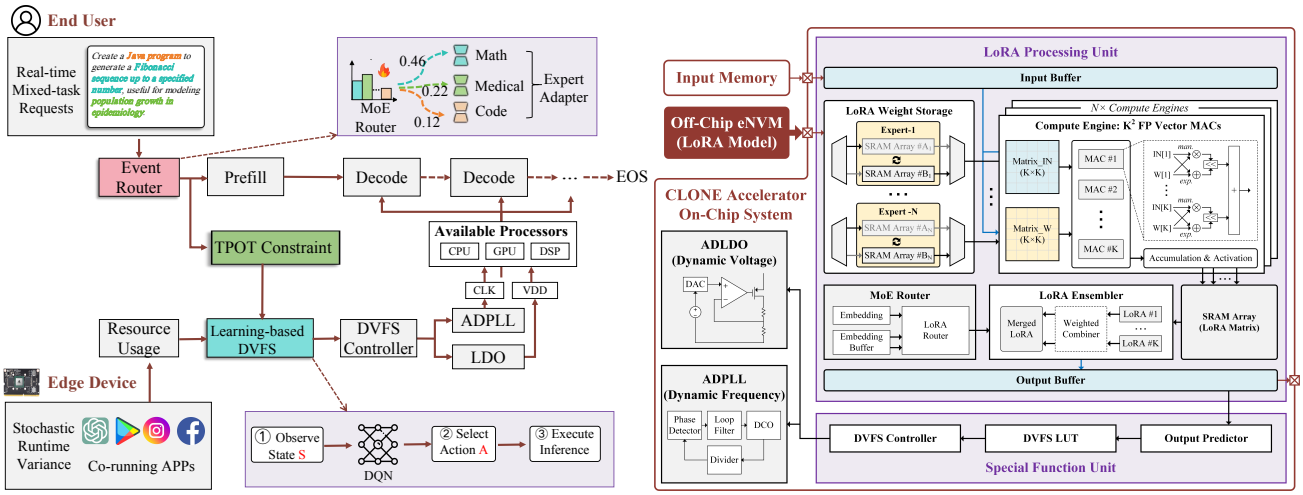
3) *Gradient-based Tailoring Optimization.* With the well-trained representation space Θ, we utilize a gradient-based optimization method to identify the optimal pruning configuration. We select top-K collected pairs serving as starting points to ensure effective initialization. Denoting such starting points as $\mathcal{E}r$, we initiate the optimization process along the gradient direction driven by the evaluator π:

$$\mathcal{E}_r^* = \mathcal{E}_r + \eta \frac{\partial \pi(\mathcal{E}_r)}{\partial \mathcal{E}_r}, \quad (2)$$

where $\mathcal{E}_r^*$ denotes the optimal pruning configuration representation, and η is step size used in the gradient update.

4) *Optimal Tailoring Generation.* Based on the optimal pruning representation $\mathcal{E}_r^*$, the optimal pruning configuration r^* is identified by the trained decoder ξ, denoted as $r^* = \xi(\mathcal{E}_r^*)$. To generate r^* iteratively without pre-specifying the ratios, we employ a beam search strategy [33], allowing for a systematic exploration of potential configurations to achieve the best possible performance. The generation process continues until the stop token $\langle$EOS$\rangle$ is encountered, enhancing the adaptiveness of the model configuration. Finally, remove non-essential groups through structural pruning, guided by the generated optimal pruning ratio.

According to *Motivation 2*, the patterns of user application usage and the inter-dependencies among tasks offer opportunities for optimization in downstream tasks, which can reduce the costs of fine-tuning and improve model performance. Therefore, plug-and-play LoRA adapters are utilized to enhance the tailored LLM generalization capabilities across diverse downstream applications. For n downstream applications, a set of LoRAs, $\Phi = \{\phi_1, \phi_2, \ldots, \phi_n\}$, is initialized, where each LoRA $\phi_i = BA$ is trained per task. The forward computation is:$y' = y + \Delta y = \mathbf{W}_0 x + BAx$, where $y' \in \mathbb{R}^d$ is the output, $x \in \mathbb{R}^k$ is the input, $B \in \mathbb{R}^{d\times r}$, $A \in \mathbb{R}^{r\times k}$ with



(a) Software (b) Hardware

Figure 10: Online latency-aware inference workflow. **Left (Software)**: CLONE employs a request-wise MoE router for dynamic LoRA configuration to manage layer-wise DVFS, optimizing energy efficiency within the specified latency targets. **Right (Hardware)**: The hardware accelerator system features a request-wise LoRA Processing Unit (LPU) designed for LoRA adaptation, and a Special Function Unit (SFU) tailored for pre-token DVFS at token layer boundaries.

$r \ll \min(d,k)$, and $\mathbf{W}_0$ are frozen LLM parameters. B is initialized to zero and A with Gaussian values.

4.3 Online Latency-aware Inference

Motivation 2 & 3 reveals that current approaches to real-time inference are constrained to workload-level resource adjustments, resulting in energy wastage. It identifies a crucial potential for system optimization via fine-grained management of performance and energy. While offline device-specific tailoring effectively reduces memory constraints and enhances downstream performance, the achieved latency can fluctuate significantly due to diverse user requests and runtime variability, potentially violating real-time latency constraints and increasing energy consumption. As shown in Figure 10, at the model level, a MoE-based router dynamically selects the optimal LoRA adapter based on stochastic user prompts, improving the accuracy of LLM-powered application responses. At the system level, CLONE integrates a token predictor to estimate the output token count, guiding a learning-based DVFS controller to adjust frequency and voltage for each token at layer boundaries. This minimizes per-request energy consumption while adhering to real-time latency targets. Additionally, CLONE embeds these efficient algorithmic enhancements into a 28nm on-chip accelerator architecture.

Request-wise MoE-based Router. In practice, end-user requests cover a diverse range of prompts, each associated with different tasks. Consequently, as shown in Figure 10 (a), an MoE (Mixture-of-Experts) [59, 117] router has been developed to dynamically and effectively merge LoRA modules for each mixed-task prompt. This development significantly extends the plug-and-play capabilities of CLONE. A set of experts $\mathbf{E} = (E_1, E_2, \ldots, E_N)$, where each expert E_i represents a tuned LoRA module. For input x_i, the output y_i:

$$y_i = \mathbf{W_o} x_i + \sum_{j=1}^{N} \omega_{ij} \cdot E_j(x_i), \tag{3}$$

where ω_{ij} modulates the contribution weights of each expert. Unlike traditional methods [131, 150] that incur extra computation and storage from trainable gate functions, we propose a parameter-free soft MoE method leveraging dynamic prompts. Using a sentence-embedding model, Γ (here adopting BGE [17]), the input embedding is formulated as $\Gamma(x)$. For each LoRA module ϕ, the embedding is obtained from randomly selected domain-specific samples k, expressed as $\Gamma(\phi) = \frac{1}{k}\sum_{i=1}^{k}\Gamma(k_{i\cdot\phi})$. To measure the similarity between the LoRA module ϕ and the prompt x, we leverage the cosine similarity, denoted as σ, as follows:

$$\sigma(x,\phi) = cos(\Gamma(x),\Gamma(\phi)), \tag{4}$$

Followed by a softmax function which takes an intermediate token representation as input and combines the output of each expert based on the gating weight $\Omega = (\omega_1, \ldots, \omega_N)$:

$$\Omega = \text{softmax}(\mathbf{s}_x), \tag{5}$$

Learning-based DVFS Controller. To further improve system effectiveness for LLM inference at the edge, a learning-based DVFS is developed to reduce per-generated token energy consumption while satisfying the real-time latency target at the layer-wise level. Among various reinforcement learning

(RL) methods [18, 62, 79, 85, 101], CLONE employs a simple two-layer MLP to efficiently learn policies within an episodic Markov decision process, ensuring minimal latency overhead. RL involves three core components:

State: Edge inference efficiency is heavily influenced by the processor intensity of co-running applications, denoted as S_{pro}. For runtime SLO constraints, inference efficiency is tightly coupled with frontend prefill time TTFT (T_{PRE}) and per-token decoding latency TPOT target (T_{DEC}).

Action: In RL, actions represent the adjustable control parameters of the system. For LLM inference at the edge, we define these actions as the selectable execution settings, specifically the energy-optimal supply voltage (V_{DD}) and the optimal running frequency (F_{req}) for each layer. Provided that the QoS constraints are met, it is feasible to lower the processor frequency, thereby conserving energy.

Reward: In RL, a reward models the optimization objective of the system. We encode energy optimization within the execution target as the reward, $\mathbb{R}_{energy}$, which is calculated using the power model [67, 70]:

$$\mathbb{R}_{energy} = \sum_{f} (P^{f}_{DEC} \times T_{DEC} + P^{f}_{PRE} \times T_{PRE}) \tag{6}$$

where P^{f}_{PRE} and P^{f}_{DEC} , representing the power consumption of the CPU/GPU at each frequency during the prefill and decoding states respectively, are determined through power measurements. These values are subsequently stored in a lookup table (LUT) within CLONE for efficient retrieval.

4.4 Hardware Accelerator System.

As illustrated in Figure 10 (b), acknowledging the limitations inherent in purely software-based approaches, CLONE introduces a scalable 28nm hardware accelerator specifically designed for edge-based LLM inference, effectively translating software optimizations into concrete performance improvements. This system integrates a LoRA Processing Unit (LPU) for dynamic adapter hot-swapping, enabling adaptive model performance optimization via dedicated data paths. Complementing this, a Special Function Unit (SFU) is equipped to perform continuous, fine-grained DVFS adjustments. The LPU executes a request-wise MoE router algorithm to efficiently manage stochastic requests, mitigating the limitations of general-purpose processors and enhancing computational throughput. Unlike conventional LoRA weight stored in on-chip SRAM, which either require frequent reloads from DRAM during wake-up cycles or keep SRAM active, wasting leakage power [73, 77, 90], we utilize an embedded non-volatile memory (eNVM) buffer to retain LoRA modules, eliminating reload overhead upon power-on. Simultaneously, the SFU employs a learning-based DVFS mechanism to adapt to runtime variability, ensuring rapid convergence to the desired voltage (V_{DD}) and frequency (F_{req}) settings. The SFU employs a lightweight predictor and DVFS model implemented as lookup tables (LUTs) for efficient hardware operation and action determination. The generated voltage (V_{DD}) and frequency (F_{req}) are realized using a fast-switching low-dropout (LDO) voltage regulator [9, 16] and an all-digital phase-locked loop (ADPLL) [2, 51], ensuring rapid runtime adjustments. Communication between the LPU and SFU is facilitated via a custom-built bi-directional streaming channel. An AXI splitter arbitrates the control flow of instructions and data bound for the LPU and SFU AXI-slave partitions.

5 EVALUATION

5.1 Experimental Setup

Infrastructure. In order to comprehensively evaluate the efficiency and effectiveness of CLONE, We perform our experiments on two heterogeneous edge devices: Orin Nano [104] and Jetson Orin NX [106]. Table 1 summarizes their specifications. To emulate heterogeneous deployment models, we utilize Llama-7B [132], Llama2-7B [133], Llama2-13B and Vicuna-7B [160] as baseline models. To simulate the heterogeneous performance requirements of multiple tasks at the edge, we employ the Flanv2 [135] dataset, which comprises 46 tasks across 10 domains, serving as the edge downstream dataset. Devices run stochastic web-search to simulate runtime variance. CodeCarbon [24] is utilized to measure runtime duration and the power consumption.

Baselines. To evaluate the effectiveness of CLONE, we compare CLONE with 7 representative approaches, including: ***A. Vanilla:*** (1) *Vanilla* assess the effectiveness of the original LLMs as the baseline criterion. ***B. Model compression:*** (2) *Random* randomly selects certain groups for pruning. (3) *LLM-Pruner* [30] removes connected structures of dependent multi-head attention from LLMs based on gradient information. (4) *ShortGPT* [94] defines the Block Influence metric to guide redundant subset removal. (5) *SliceGPT* [8] replaces each weight matrix in the network with a smaller, dense matrix, thereby reducing the network embedding dimension. ***C. Computation optimization:*** (6) *FlexGen* [118] retains all model parameters on CPU DRAM. During inference, the demand weights are loaded from the CPU to the GPU. ***D. Small size model:*** (7) OpenLLaMA-3B [35], an open reproduction of Llama with 3B parameters, trained on 1T tokens.

Evaluation Metrics. In order to comprehensively evaluate the efficiency and effectiveness of CLONE, we conduct evaluations from three perspectives. ***A.Generation ability.*** We measure the zero-shot perplexity (PPL) analysis on WikiText2 [95] and PTB [93], two of the most commonly used evaluation datasets, to evaluate the generation capabilities of the customized model. Lower PPL values signify stronger generation ability. ***B. General-purpose task solving ability.*** To demonstrate the world knowledge and problem-solving skills of the customized model, we utilize three benchmarks:

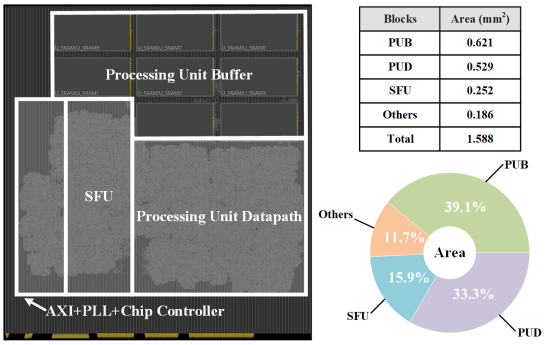


Figure 11: 28nm physical layout and area breakdown of the energy-optimal CLONE accelerator system.

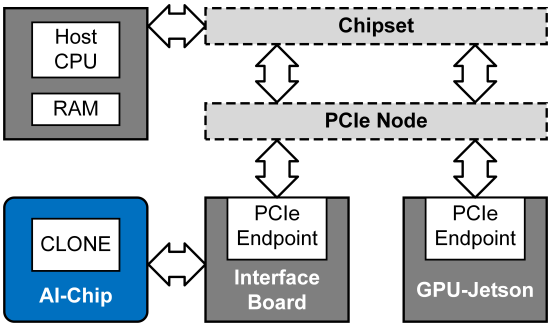


Figure 12: Test system. Integration of the CLONE accelerator into the Jetson platform via a PCIe interface to facilitate high-performance data exchange.

1) BBH [123], which includes 23 challenging tasks such as Q&A, natural language reasoning, and sentiment analysis. 2) MMLU [46], covering 57 tasks across diverse domains like mathematics, history, law, and ethics. 3) Commonsense reasoning tasks used in Llama paper [132], featuring BoolQ [22], PIQA [12], OBQA [100], ARC-c [23], ARC-e [23], WinoGrande [116], and HellaSwag [151]. ***C. System effectiveness.*** We evaluate the system effectiveness of CLONE from two perspectives: latency and energy consumption, which are defined as the duration and energy cost to infer the WikiText2 [95].

Hyperparameter and Reproducibility. For offline device-specific tailoring, we first execute classic approaches for 100 epochs to collect training data, and use 25× randomly shuffled client selection as data augmentation for tailoring training. Both Encoder and Decoder use a single-layer LSTM configuration, while the Predictor has a dual-layer feed-forward setup. Hidden state dimensions are 64 for Encoder and Decoder, and 200 for the Predictor, with all layers having an embedding size of 32. The hyperparameters include a batch size of 1024, a learning rate of 0.001, and $\eta = 0.8$. For optimization, we start with the top 25 model pruning ratio records. The LoRA rank r and the scaling hyperparameter α are set to 8 and 16, respectively, with 3 training rounds.

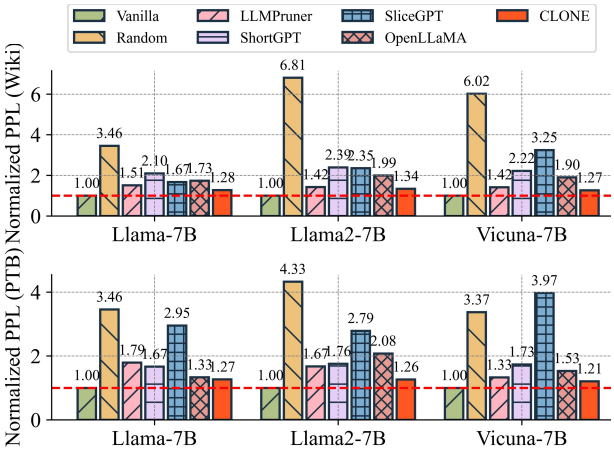


Figure 13: Generation ability comparison of various schemes on WikiText2 and PTB dataset across 3 representative LLMs.

5.2 Test System

Figure 11 presents the layout of the energy-optimal 28nm CLONE accelerator system. After place-and-route, the core footprint is only 1.588 mm^2; the figure also delineates module boundaries and reports the post-layout power- and area-breakdowns. Due to the high cost and time requirements of a full tape-out, the CLONE accelerator was validated through post-layout simulation. The physical layout was generated from Cadence Innovus synthesis results, following a complete design flow: behavioral modeling, RTL design, synthesis, P&R, and physical verification, ensuring simulation fidelity to actual fabrication. Figure 12 illustrates the integration of the CLONE accelerator into the Jetson platform via a PCIe interface to facilitate high-performance data exchange. The data flow initiates with the AI-Chip (CLONE), which interfaces with the Jetson GPU through a PCIe connection. An interface board bridges the PCIe endpoints of the CLONE accelerator and the Jetson GPU. This architecture is supported by the host CPU and RAM, where the CPU handles top-level data flow control, ensuring that requests are efficiently managed and routed. The CLONE is responsible for executing tasks based on user inputs, such as selecting the most efficient LoRA group and performing edge real-time DVFS control to optimize performance. By bypassing system software dependencies, this hardware-centric setup ensures seamless communication and data transfer between the CLONE and the Jetson device, achieving high levels of performance and energy efficiency.

5.3 Evaluation Results and Analysis

Generation Ability. To evaluate the customized model generation ability, we use the WikiText2 and PTB datasets to measure the zero-shot PPL (lower values correspond to stronger text generation capability) across different models on the

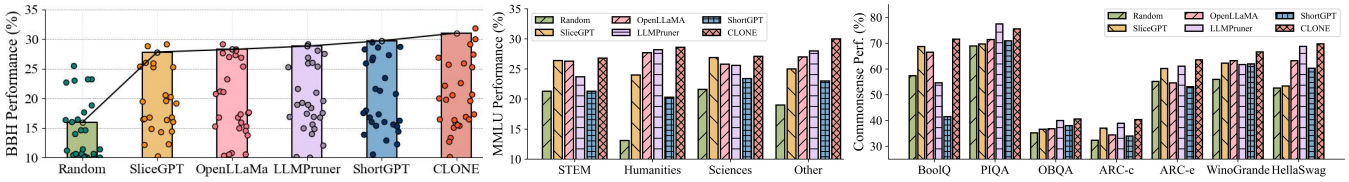
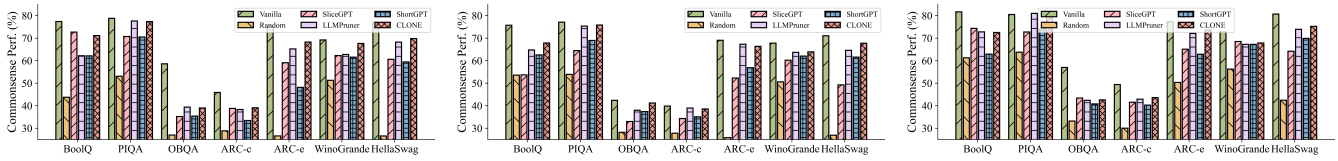


Figure 14: Downstream task performance of different customization approaches of Llama-7B on 3 representative benchmarks, including BBH (zero-shot), MMLU (3-shot), and Commonsense (zero-shot).



(a) Llama2-7B (b) Vicuna-7B (c) Llama2-13B

Figure 15: Zero-shot performance (commonsense) of Llama2-7B, Vicuna-7B and Llama2-13B.

Orin NX. Figure 13 shows the zero-shot PPL values that compare CLONE with the baselines. In order to effectively demonstrate the characteristics of different schemes, we normalize the measured PPL values to the vanilla model. Compared to baselines, CLONE preserves the most generation ability, achieving up to 5.1× generating capacity of *Random* on WikiText2 and up to 3.4× on PTB. This can be attributed to CLONE using PPL as a pruning metric to construct a continuous pruning selection space. It then utilizes gradient-based optimization to identify the optimal pruning configuration that minimally impacts generation ability.

Downstream Task Performance. Figure 14 illustrates the downstream task performance of CLONE across three representative benchmarks (BBH, MMLU, and Commonsense), covering 87 tasks. We observe that CLONE significantly outperforms other baselines across various domains and tasks. Specifically, for BBH, CLONE improves the average accuracy by 15.1% over Random and by 2.37% over others on average; for MMLU, it increases average accuracy by 6.0% over Random and by 2.96% over others on average; for commonsense tasks, CLONE enhances average accuracy by 10.1% over Random and by 6.1% over others on average. The principle as discussed in §3, CLONE implements task-aware tailoring for specific devices by considering end-user usage patterns and task specificity and relevance. It selects appropriate types and quantities of LoRA for fine-tuning. Moreover, we note that baseline performance varies across different downstream tasks. For instance, in BBH, ShortGPT is the sub-optimal model, whereas in Commonsense, LLMPruner is the sub-optimal. This variability stems from the heuristic and incomplete nature of baseline approaches. In contrast, CLONE utilizes a request-wise MoE-based router to effectively manage the properties of diverse real-world requests, resulting in superior LoRA module fusion. Overall, this experiment demonstrates CLONE's practical and robust ability to scale to complex workloads and applications.

Robustness and Scalability. Figure 15 details the zero-shot performance of the pruned model across a variety of downstream tasks, as well as its zero-shot perplexity on the WikiText2 and PTB datasets, utilizing diverse pruning configurations. Our observations confirm that CLONE consistently outperforms established baselines across all metrics.*1) Different types of models*: Comparing Llama2-7B and Vicuna-7B, the results indicate that CLONE exhibits a strong generalization capability, outperforming baselines by an average of 23.85%. *2) Different model sizes*: Evaluating Llama2-7B and Llama2-13B, it is evident from Figure 15 that CLONE maintains 91.13% of the performance of the unpruned (vanilla) models, underscoring its scalability. These results collectively affirm that CLONE is scalable to different LLMs in general, not limited to specific LLMs, so it is not a one-off "software-ASIC".

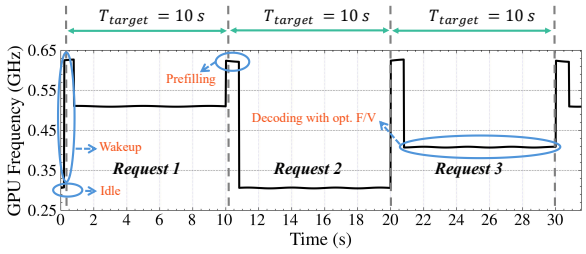


Figure 16: Simulations of LDO dynamic V_{DD} adjustments.

System Effectiveness. Then, we evaluate the system efficiency of CLONE from latency and energy consumption during the LLMs inference process on the WikiText2 dataset at the off-the-shelf devices Orin NX and Nano. As shown

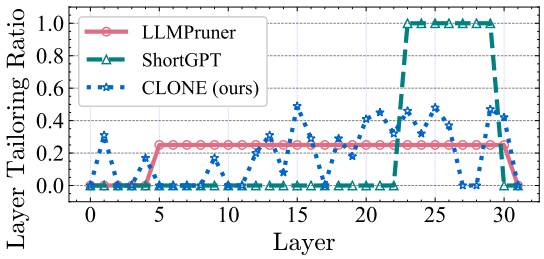


Figure 17: Comparison of the model pruning configuration.

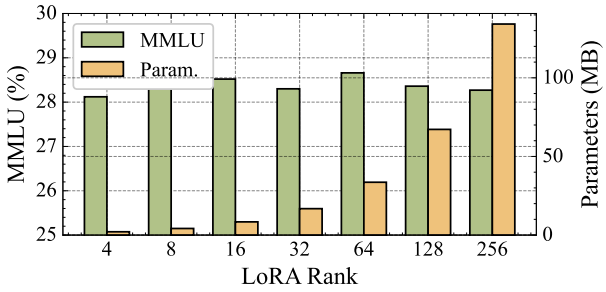


Figure 18: MMLU performance and parameter size across LoRA ranks.

in Table 3, we can see that CLONE effectively speeds up the inference process up to 11.92×, and achieves significant energy saving up to 7.36×. Figure 16 shows the spice-level simulation of the DVFS for runtime inference. There are two possible reasons 1) As shown in Equation 1, CLONE jointly considers the latency and energy budgets as metrics to generate optimal pruning configuration. 2) CLONE decouples prefill and decoding phases, applying request-level DVFS to boost effectiveness. In contrast, FlexGen trades high latency and energy for full-parameter LLMs, while methods like LLMPruner and ShortGPT focus on static, one-off compression without considering system-level optimization.

Table 3: System efficiency (latency and energy) comparison of various schemes to inference WikiText2 dataset.

Method	Energy (Wh)		Latency (s)	
	Jetson NX	Jetson Nano	Jetson NX	Jetson Nano
Random	7.27	8.26	842.40	1145.07
SliceGPT	5.47	7.54	661.65	929.39
OpenLLaMA	5.67	7.70	506.73	662.73
LLMPruner	6.01	6.91	622.92	1023.51
ShortGPT	5.67	8.56	555.14	698.02
FlexGen	21.12	26.04	3166.27	4674.42
CLONE^{-HW}	4.81	5.56	462.72	552.18
CLONE	**3.46**	**3.54**	**322.76**	**392.15**

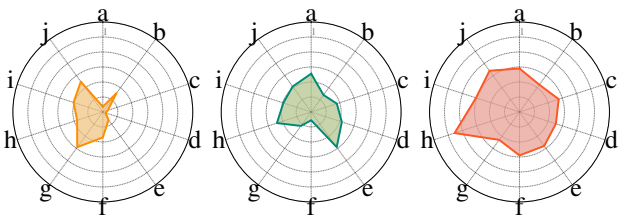
Study of the Generative Pruning Configuration. To examine how CLONE adaptively generates the optimal model pruning configuration, we compare the generative pruning approach to the layer-wise method used by ShortGPT and the uniform direct average method employed by LLMPruner on Llama-7B to illustrate the strategic disparities in their decision-making processes. As presented in Figure 17, LLMPruner remains static from the 5th to 30th layer. In contrast, ShortGPT uses only binary values of 0 or 1. We note that the layer-level pruning configuration determined by CLONE dynamically varies across layers. This adaptability reflects a more nuanced decision-making process, where the contributions of individual layers are recognized as heterogeneous, as discussed in *Motivation 1*. This indicates that CLONE's generative decision-making is more adaptive and tailored to the specific contributions of each layer.

Effectiveness of the Hardware Accelerator. The accelerator boosts efficiency by exploiting dedicated hardware resources; it does not itself alter generation quality or task-specific accuracy. Disabling it forces the workload onto general-purpose compute units, eliminating these hardware-level gains. As Table 3 shows, energy rises from 3.46Wh to 4.81Wh and end-to-end latency from 322.76s to 462.72s without the accelerator CLONE^{-HW}. Even in this regime, our offline, metric-guided generative pruning (Eq. 1) still surpasses all baselines, underscoring its robustness.

Effectiveness of Request-wise MoE Router. Figure 19 illustrates the performance of different LoRA adapter fusion methods during runtime inference for dynamic end-user requests. *w/o MoE* indicates direct averaging of all LoRA modules for all requests, which typically results in suboptimal performance due to the fact that not all stochastic multi-tasks benefit each other. *MoE (Top-1)* means based on the request to select the most similar LoRA for LLM inference. However, from Figure 19, we can see that *w/o MoE* Simply combining LoRAs results in worse performance because not all stochastic multi-tasks are beneficial to the others. Though *MoE (Top-1)* signifies that the most similar LoRA is selected based on the request for LLM inference. Although this approach enhances overall performance, it can still be sub-optimal in practice, as even a single request may involve multiple tasks, and merely considering task differences is insufficient. This observation highlights the critical importance of both the request-wise MoE router in maintaining CLONE performance within diverse data and task demands.

Impact of the Rank of LoRA Adapters. The rank r controls the number of trainable parameters, where larger values of r provide greater adapter capacity at the expense of increased training overhead. As illustrated in Figure 18, performance initially improves with higher ranks but eventually saturates, offering diminishing returns despite the rapidly growing parameter size.

Overhead Analysis. Offline tailoring utilizes a single-layer LSTM encoder-decoder and a dual-layer feed-forward network, adding minimal overhead and enhancing accuracy without affecting online inference. During online inference, the soft MoE router introduces no additional trainable parameters and performs a single probabilistic routing computation.



(a) w/o MoE (b) MoE (Top-1) (c) CLONE

Figure 19: Performance magnitudes of w/o MoE, MoE (Top-1) and CLONE. a-j represent different downstream tasks.

Implemented as a two-layer MLP with under 1K parameters, the DVFS controller imposes negligible overhead relative to billion-parameter LLMs. Both modules first execute concurrently (< 10ms) with the prefill stage (> 100 ms; see Figure 2), then the DVFS decision for token $t+1$ is generated while token t is being decoded, keeping the controller off the critical path. Meanwhile, the incremental energy cost of CLONE is on the milliwatt-hour scale, rendering it negligible compared with the watt-hour-level consumption of full-model inference.

6 Related Work

LLMs are highly memory-, compute-, and energy-intensive [5, 19, 96, 109, 132, 133], driving most "billion-parameter" inference to the cloud [6, 34, 81, 122, 124, 145, 149]. However, as edge devices become more powerful, there is increasing interest in executing LLM inference on the edge [4, 6, 34, 57, 74, 81, 107, 111, 122, 145, 149], which enhances data privacy and enables real-time service delivery. To address edge resource constraints, techniques such as model architecture search [54, 60, 88], quantization [26, 55, 57, 91, 143], pruning [27, 32, 91, 103], and knowledge distillation [41, 71] have been proposed. However, most approaches focus solely on model-level optimization, neglecting system-level trade-offs like storage and weight efficiency. Advances in LLM compilers and software stacks [56, 99, 113, 128] have enabled integration with co-processors and near-sensor processing [4, 43, 57, 74, 107, 111, 118, 122, 148, 155–158], but these often add computational and communication overhead, reducing edge device longevity and hindering concurrent application execution. Existing model customization methods [82, 126, 127, 153, 154] cannot directly address task-agnostic LLMs, where generalization is crucial. The unique characteristics of LLM decoder layers and stochastic outputs present untapped opportunities for hardware optimization. For system-level optimization, DVFS [10, 47, 48, 65, 112] has been widely used to dynamically adjust processor voltage and frequency. However, most DVFS strategies are designed for discriminative models like CNNs and RNNs, treating networks as black boxes. Generative LLMs, with their auto-regressive inference and stochastic prompt variability, remain underexplored in this context.

Edge–cloud collaboration offers a pragmatic middle ground between fully local and fully cloud-based inference. In practice, the edge should remain the first line of execution for latency-critical or privacy-sensitive workloads, running compact SLMs that fit the device's power and memory envelope. When a request exceeds local capacity—e.g., requires deeper reasoning, broader context, or larger knowledge—CLONE can transparently escalate the call to a cloud-resident LLM. This selective offloading preserves real-time responsiveness, keeps private data on-device whenever possible, and amortizes bandwidth and compute costs by invoking the cloud only for the fraction of tasks that truly need it.

7 Conclusion

CLONE integrates offline generative pruning and online latency-aware optimization to enhance edge deployment of LLMs. Offline, it dynamically generates device-specific pruning configurations via a generative framework. Online, CLONE employs a layer-wise DVFS strategy to optimize energy efficiency within latency constraints, using an MoE router with multiple LoRA adapters to support diverse tasks. Supported by a dedicated 28nm hardware accelerator, including specialized units for rapid LoRA adapter switching and fine-grained DVFS control, benchmarks confirm that CLONE significantly improves inference speed and energy efficiency while maintaining robust task performance, ideal for latency-sensitive, energy-constrained edge scenarios.

Acknowledgment

We sincerely thank the anonymous ATC'25 reviewers, and our shepherd, Dr. Joel Wolfrath, for their insightful suggestions. This work is supported in part by the Science and Technology Development Fund of Macau (0107/2024/RIA2), Joint Science and Technology Research Project with Hong Kong and Macau in Key Areas of Nansha District's Science and Technology Plan (EF2024-00180-IOTSC) and the Multi-Year Research Grant of University of Macau (MYRG-GRG2023-00211-IOTSC-UMDF, MYRG-GRG2024-00180-IOTSC). Please ask Dr. Li Li (llili@um.edu.mo) for correspondence.

References

[1] Rewind AI. Rewind: A better way to search your digital life, 2024.

[2] Tutu Ajayi, Sumanth Kamineni, Yaswanth K Cherivirala, Morteza Fayazi, Kyumin Kwon, Mehdi Saligane, Shourya Gupta, Chien-Hen Chen, Dennis Sylvester,

David Blaauw, et al. An open-source framework for autonomous soc design with analog block generation. In *2020 IFIP/IEEE 28th International Conference on Very Large Scale Integration (VLSI-SOC)*, pages 141–146. IEEE, 2020.

[3] Jay Alammar. The illustrated gpt-2 (visualizing transformer language models). *Jalammar. github. io. https://jalammar. github. io/illustrated-gpt2*, 2019.

[4] Keivan Alizadeh, Iman Mirzadeh, Dmitry Belenko, Karen Khatamifard, Minsik Cho, Carlo C. Del Mundo, Mohammad Rastegari, and Mehrdad Farajtabar. LLM in a flash: Efficient large language model inference with limited memory. *CoRR*, abs/2312.11514, 2023.

[5] Anthropic. Claude: A family of ai models, 2024.

[6] Apple. Apple intelligence. https://www.apple.com/apple-intelligence/, 2024.

[7] Apple. Apple siri: Virtual assistant, 2024.

[8] Saleh Ashkboos, Maximilian L. Croci, Marcelo Gennari Do Nascimento, Torsten Hoefler, and James Hensman. Slicegpt: Compress large language models by deleting rows and columns. *CoRR*, abs/2401.15024, 2024.

[9] Suyoung Bang, Wootaek Lim, Charles Augustine, Andres Malavasi, Muhammad Khellah, James Tschanz, and Vivek De. 25.1 a fully synthesizable distributed and scalable all-digital ldo in 10nm cmos. In *2020 IEEE International Solid-State Circuits Conference-(ISSCC)*, pages 380–382. IEEE, 2020.

[10] Soroush Bateni and Cong Liu. {NeuOS}: A {Latency-Predictable}{Multi-Dimensional} optimization framework for {DNN-driven} autonomous systems. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 371–385, 2020.

[11] Emily M. Bender, Timnit Gebru, Angelina McMillan-Major, and Shmargaret Shmitchell. On the dangers of stochastic parrots: Can language models be too big? In Madeleine Clare Elish, William Isaac, and Richard S. Zemel, editors, *FAccT '21: 2021 ACM Conference on Fairness, Accountability, and Transparency, Virtual Event / Toronto, Canada, March 3-10, 2021*, pages 610–623. ACM, 2021.

[12] Yonatan Bisk, Rowan Zellers, Ronan Le Bras, Jianfeng Gao, and Yejin Choi. PIQA: reasoning about physical commonsense in natural language. In *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020*, pages 7432–7439. AAAI Press, 2020.

[13] Rishi Bommasani, Drew A. Hudson, Ehsan Adeli, Russ B. Altman, Simran Arora, Sydney von Arx, Michael S. Bernstein, Jeannette Bohg, Antoine Bosselut, Emma Brunskill, Erik Brynjolfsson, Shyamal Buch, Dallas Card, Rodrigo Castellon, Niladri S. Chatterji, Annie S. Chen, Kathleen Creel, Jared Quincy Davis, Dorottya Demszky, Chris Donahue, Moussa Doumbouya, Esin Durmus, Stefano Ermon, John Etchemendy, Kawin Ethayarajh, Li Fei-Fei, Chelsea Finn, Trevor Gale, Lauren E. Gillespie, Karan Goel, Noah D. Goodman, Shelby Grossman, Neel Guha, Tatsunori Hashimoto, Peter Henderson, John Hewitt, Daniel E. Ho, Jenny Hong, Kyle Hsu, Jing Huang, Thomas Icard, Saahil Jain, Dan Jurafsky, Pratyusha Kalluri, Siddharth Karamcheti, Geoff Keeling, Fereshte Khani, Omar Khattab, Pang Wei Koh, Mark S. Krass, Ranjay Krishna, Rohith Kuditipudi, and et al. On the opportunities and risks of foundation models. *CoRR*, abs/2108.07258, 2021.

[14] Peter F Brown, Stephen A Della Pietra, Vincent J Della Pietra, Jennifer C Lai, and Robert L Mercer. An estimate of an upper bound for the entropy of english. *Computational Linguistics*, 18(1):31–40, 1992.

[15] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin, editors, *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020.

[16] Chaitanya K Chava and José Silva-Martínez. A frequency compensation scheme for ldo voltage regulators. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 51(6):1041–1050, 2004.

[17] Jianlv Chen, Shitao Xiao, Peitian Zhang, Kun Luo, Defu Lian, and Zheng Liu. Bge m3-embedding: Multilingual, multi-functionality, multi-granularity text embeddings through self-knowledge distillation, 2024.

[18] Yonghun Choi, Seonghoon Park, and Hojung Cha. Optimizing energy efficiency of browsers in energy-aware scheduling-enabled mobile devices. In Stephen A. Brewster, Geraldine Fitzpatrick, Anna L. Cox, and Vassilis Kostakos, editors, *The 25th Annual International Conference on Mobile Computing and Networking, MobiCom 2019, Los Cabos, Mexico, October 21-25, 2019*, pages 48:1–48:16. ACM, 2019.

[19] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, Parker Schuh, Kensen Shi, Sasha Tsvyashchenko, Joshua Maynez, Abhishek Rao, Parker Barnes, Yi Tay, Noam Shazeer, Vinodkumar Prabhakaran, Emily Reif, Nan Du, Ben Hutchinson, Reiner Pope, James Bradbury, Jacob Austin, Michael Isard, Guy Gur-Ari, Pengcheng Yin, Toju Duke, Anselm Levskaya, Sanjay Ghemawat, Sunipa Dev, Henryk Michalewski, Xavier Garcia, Vedant Misra, Kevin Robinson, Liam Fedus, Denny Zhou, Daphne Ippolito, David Luan, Hyeontaek Lim, Barret Zoph, Alexander Spiridonov, Ryan Sepassi, David Dohan, Shivani Agrawal, Mark Omernick, Andrew M. Dai, Thanumalayan Sankaranarayana Pillai, Marie Pellat, Aitor Lewkowycz, Erica Moreira, Rewon Child, Oleksandr Polozov, Katherine Lee, Zongwei Zhou, Xuezhi Wang, Brennan Saeta, Mark Diaz, Orhan Firat, Michele Catasta, Jason Wei, Kathy Meier-Hellstern, Douglas Eck, Jeff Dean, Slav Petrov, and Noah Fiedel. Palm: Scaling language modeling with pathways. *J. Mach. Learn. Res.*, 24:240:1–240:113, 2023.

[20] Jae-Won Chung, Jiachen Liu, Zhiyu Wu, Yuxuan Xia, and Mosharaf Chowdhury. ML.ENERGY leaderboard. https://ml.energy/leaderboard, 2023.

[21] Aidan Clark, Diego de Las Casas, Aurelia Guy, Arthur Mensch, Michela Paganini, Jordan Hoffmann, Bogdan Damoc, Blake A. Hechtman, Trevor Cai, Sebastian Borgeaud, George van den Driessche, Eliza Rutherford, Tom Hennigan, Matthew J. Johnson, Albin Cassirer, Chris Jones, Elena Buchatskaya, David Budden, Laurent Sifre, Simon Osindero, Oriol Vinyals, Marc'Aurelio Ranzato, Jack W. Rae, Erich Elsen, Koray Kavukcuoglu, and Karen Simonyan. Unified scaling laws for routed language models. In Kamalika Chaudhuri, Stefanie Jegelka, Le Song, Csaba Szepesvári, Gang Niu, and Sivan Sabato, editors, *International Conference on Machine Learning, ICML 2022, 17-23 July 2022, Baltimore, Maryland, USA*, volume 162 of *Proceedings of Machine Learning Research*, pages 4057–4086. PMLR, 2022.

[22] Christopher Clark, Kenton Lee, Ming-Wei Chang, Tom Kwiatkowski, Michael Collins, and Kristina Toutanova. Boolq: Exploring the surprising difficulty of natural yes/no questions. In Jill Burstein, Christy Doran, and Thamar Solorio, editors, *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, pages 2924–2936. Association for Computational Linguistics, 2019.

[23] Peter Clark, Isaac Cowhey, Oren Etzioni, Tushar Khot, Ashish Sabharwal, Carissa Schoenick, and Oyvind Tafjord. Think you have solved question answering? try arc, the ai2 reasoning challenge. *arXiv:1803.05457v1*, 2018.

[24] codecarbon. Track and reduce co2 emissions from your computing. https://codecarbon.io/, 2023.

[25] Dell. Llama 2: Inferencing on a single gpu. https://infohub.delltechnologies.com/zh-cn/t/llama-2-inferencing-on-a-single-gpu/, 2023.

[26] Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. Llm.int8(): 8-bit matrix multiplication for transformers at scale. In Sanmi Koyejo, S. Mohamed, A. Agarwal, Danielle Belgrave, K. Cho, and A. Oh, editors, *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*, 2022.

[27] Peijie Dong, Lujun Li, Zhenheng Tang, Xiang Liu, Xinglin Pan, Qiang Wang, and Xiaowen Chu. Prunerzero: Evolving symbolic pruning metric from scratch for large language models. In *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*. OpenReview.net, 2024.

[28] Begum Egilmez, Matthew Schuchhardt, Gokhan Memik, Raid Ayoub, Niranjan Soundararajan, and Michael Kishinevsky. User-aware frame rate management in android smartphones. *ACM Trans. Embed. Comput. Syst.*, 16(5s):131:1–131:17, 2017.

[29] Yasuhiro Endo, Zheng Wang, J. Bradley Chen, and Margo I. Seltzer. Using latency to evaluate interactive system performance. In Karin Petersen and Willy Zwaenepoel, editors, *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation (OSDI), Seattle, Washington, USA, October 28-31, 1996*, pages 185–199. ACM, 1996.

[30] Gongfan Fang, Xinyin Ma, Mingli Song, Michael Bi Mi, and Xinchao Wang. Depgraph: Towards any structural pruning. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR 2023, Van-*

couver, BC, Canada, June 17-24, 2023, pages 16091–16101. IEEE, 2023.

[31] Luciano Floridi and Massimo Chiriatti. Gpt-3: Its nature, scope, limits, and consequences. *Minds and Machines*, 30:681–694, 2020.

[32] Elias Frantar and Dan Alistarh. Sparsegpt: Massive language models can be accurately pruned in one-shot. In Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett, editors, *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA*, volume 202 of *Proceedings of Machine Learning Research*, pages 10323–10337. PMLR, 2023.

[33] Markus Freitag and Yaser Al-Onaizan. Beam search strategies for neural machine translation. In Thang Luong, Alexandra Birch, Graham Neubig, and Andrew M. Finch, editors, *Proceedings of the First Workshop on Neural Machine Translation, NMT@ACL 2017, Vancouver, Canada, August 4, 2017*, pages 56–60. Association for Computational Linguistics, 2017.

[34] Daocheng Fu, Xin Li, Licheng Wen, Min Dou, Pinlong Cai, Botian Shi, and Yu Qiao. Drive like a human: Rethinking autonomous driving with large language models. In *Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision*, pages 910–919, 2024.

[35] Xinyang Geng and Hao Liu. Openllama: An open reproduction of llama. https://github.com/openlm-research/open_llama, May 2023.

[36] Amir Gholami, Zhewei Yao, Sehoon Kim, Coleman Hooper, Michael W. Mahoney, and Kurt Keutzer. AI and memory wall. *IEEE Micro*, 44(3):33–39, 2024.

[37] github. GitHub Copilot: Your AI pair programmer. https://github.com/features/copilot.

[38] Ian J. Goodfellow, Yoshua Bengio, and Aaron C. Courville. *Deep Learning*. Adaptive computation and machine learning. MIT Press, 2016.

[39] Google. Google assistant, 2024.

[40] Google. Ml kit smart reply, 2024.

[41] Yuxian Gu, Li Dong, Furu Wei, and Minlie Huang. Minillm: Knowledge distillation of large language models. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net, 2024.

[42] Peizhen Guo, Bo Hu, and Wenjun Hu. Mistify: Automating DNN model porting for on-device inference at the edge. In James Mickens and Renata Teixeira, editors, *18th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2021, April 12-14, 2021*, pages 705–719. USENIX Association, 2021.

[43] Pengyu He, Yuanzhe Zhao, Heng Xie, Yang Wang, Shouyi Yin, Li Li, Yan Zhu, Rui P Martins, Chi-Hang Chan, and Minglei Zhang. A reconfigurable floating-point compute-in-memory with analog exponent preprocesses. *IEEE Solid-State Circuits Letters*, 2024.

[44] Pengyu He, Yuanzhe Zhao, Heng Xie, Yang Wang, Shouyi Yin, Li Li, Yan Zhu, Rui Paulo Martins, Chi-Hang Chan, and Minglei Zhang. A 28nm 314.6 tflops/w reconfigurable floating-point analog compute-in-memory macro with exponent approximation and two-stage sharing td-adc. In *2024 IEEE Custom Integrated Circuits Conference (CICC)*, pages 1–2. IEEE, 2024.

[45] Yihui He, Xiangyu Zhang, and Jian Sun. Channel pruning for accelerating very deep neural networks. In *IEEE International Conference on Computer Vision, ICCV 2017, Venice, Italy, October 22-29, 2017*, pages 1398–1406. IEEE Computer Society, 2017.

[46] Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. Measuring massive multitask language understanding. *Proceedings of the International Conference on Learning Representations (ICLR)*, 2021.

[47] Sebastian Herbert and Diana Marculescu. Variation-aware dynamic voltage/frequency scaling. In *15th International Conference on High-Performance Computer Architecture (HPCA-15 2009), 14-18 February 2009, Raleigh, North Carolina, USA*, pages 301–312. IEEE Computer Society, 2009.

[48] Robert Hesse and Natalie D. Enright Jerger. Improving DVFS in nocs with coherence prediction. In André Ivanov, Diana Marculescu, Partha Pratim Pande, José Flich, and Karthik Pattabiraman, editors, *Proceedings of the 9th International Symposium on Networks-on-Chip, NOCS 2015, Vancouver, BC, Canada, September 28-30, 2015*, pages 24:1–24:8. ACM, 2015.

[49] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Comput.*, 9(8):1735–1780, 1997.

[50] Ari Holtzman, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi. The curious case of neural text degeneration. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net, 2020.

[51] Terng-Yin Hsu, Bai-Jue Shieh, and Chen-Yi Lee. An all-digital phase-locked loop (adpll)-based clock recovery circuit. *IEEE Journal of Solid-State Circuits*, 34(8):1063–1073, 1999.

[52] Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. LoRA: Low-Rank Adaptation of Large Language Models. *arXiv e-prints*, page arXiv:2106.09685, June 2021.

[53] Yi Huang, Zhiyu Chen, Dai Li, and Kaiyuan Yang. CAMA: energy and memory efficient automata processing in content-addressable memories. In *IEEE International Symposium on High-Performance Computer Architecture, HPCA 2022, Seoul, South Korea, April 2-6, 2022*, pages 25–37. IEEE, 2022.

[54] Yingbing Huang, Lily Jiaxin Wan, Hanchen Ye, Manvi Jha, Jinghua Wang, Yuhong Li, Xiaofan Zhang, and Deming Chen. New solutions on llm acceleration, optimization, and application. In *Proceedings of the 61st ACM/IEEE Design Automation Conference*, pages 1–4, 2024.

[55] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Quantized neural networks: Training neural networks with low precision weights and activations. *Journal of Machine Learning Research*, 18(187):1–30, 2018.

[56] HuggingFace Team. Transformers documentation, 2024.

[57] Mohamed Assem Ibrahim, Shaizeen Aga, Ada Li, Suchita Pati, and Mahzabeen Islam. Just-in-time quantization with processing-in-memory for efficient ml training, 2023.

[58] Mem Inc. Mem: Your ai-powered assistant, 2024.

[59] Robert A. Jacobs, Michael I. Jordan, Steven J. Nowlan, and Geoffrey E. Hinton. Adaptive mixtures of local experts. *Neural Comput.*, 3(1):79–87, 1991.

[60] Ganesh Jawahar, Muhammad Abdul-Mageed, Laks VS Lakshmanan, and Dujian Ding. Llm performance predictors are good initializers for architecture search. *arXiv preprint arXiv:2310.16712*, 2023.

[61] Kabilankb. Robot control using llama: Bridging ai and robotics. Medium, 2024. Oct 13, 2024.

[62] Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. Reinforcement learning: A survey. *Journal of artificial intelligence research*, 4:237–285, 1996.

[63] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models. *CoRR*, abs/2001.08361, 2020.

[64] Byeongho Kim, Sanghoon Cha, Sangsoo Park, Jieun Lee, Sukhan Lee, Shinhaeng Kang, Jinin So, Kyungsoo Kim, Jin Jung, Jong-Geon Lee, Sunjung Lee, Yoonah Paik, Hyeonsu Kim, Jin-Seong Kim, Won-Jo Lee, Yuhwan Ro, Yeongon Cho, Jin Hyun Kim, Joon-Ho Song, Jaehoon Yu, Seungwon Lee, Jeonghyeon Cho, and Kyomin Sohn. The breakthrough memory solutions for improved performance on LLM inference. *IEEE Micro*, 44(3):40–48, 2024.

[65] Wonyoung Kim, Meeta Sharma Gupta, Gu-Yeon Wei, and David M. Brooks. System level analysis of fast, per-core DVFS using on-chip switching regulators. In *14th International Conference on High-Performance Computer Architecture (HPCA-14 2008), 16-20 February 2008, Salt Lake City, UT, USA*, pages 123–134. IEEE Computer Society, 2008.

[66] Young Geun Kim, Minyong Kim, and Sung Woo Chung. Enhancing energy efficiency of multimedia applications in heterogeneous mobile multi-core processors. *IEEE Trans. Computers*, 66(11):1878–1889, 2017.

[67] Young Geun Kim, Minyong Kim, Jae Min Kim, Minyoung Sung, and Sung Woo Chung. A novel gpu power model for accurate smartphone power breakdown. *ETRI journal*, 37(1):157–164, 2015.

[68] Young Geun Kim, Joonho Kong, and Sung Woo Chung. A survey on recent os-level energy management techniques for mobile processing units. *IEEE Trans. Parallel Distributed Syst.*, 29(10):2388–2401, 2018.

[69] Young Geun Kim and Carole-Jean Wu. Autoscale: Energy efficiency optimization for stochastic edge inference using reinforcement learning. In *53rd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2020, Athens, Greece, October 17-21, 2020*, pages 1082–1096. IEEE, 2020.

[70] Youngsok Kim, Joonsung Kim, Dongju Chae, Daehyun Kim, and Jangwoo Kim. μlayer: Low latency on-device inference using cooperative single-layer acceleration and processor-friendly quantization. In *Proceedings of the Fourteenth EuroSys Conference 2019*, pages 1–15, 2019.

[71] Jongwoo Ko, Sungnyun Kim, Tianyi Chen, and Se-Young Yun. Distillm: Towards streamlined distillation for large language models. In *Forty-first International*

Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024. OpenReview.net, 2024.

[72] Brett Koonce and Brett Koonce. Resnet 50. *Convolutional neural networks with swift for tensorflow: image recognition and dataset categorization*, pages 63–72, 2021.

[73] Srivatsan Krishnan, Zishen Wan, Kshitij Bhardwaj, Paul Whatmough, Aleksandra Faust, Sabrina Neuman, Gu-Yeon Wei, David Brooks, and Vijay Janapa Reddi. Automatic domain-specific soc design for autonomous unmanned aerial vehicles. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 300–317. IEEE, 2022.

[74] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In Jason Flinn, Margo I. Seltzer, Peter Druschel, Antoine Kaufmann, and Jonathan Mace, editors, *Proceedings of the 29th Symposium on Operating Systems Principles, SOSP 2023, Koblenz, Germany, October 23-26, 2023*, pages 611–626. ACM, 2023.

[75] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In Jason Flinn, Margo I. Seltzer, Peter Druschel, Antoine Kaufmann, and Jonathan Mace, editors, *Proceedings of the 29th Symposium on Operating Systems Principles, SOSP 2023, Koblenz, Germany, October 23-26, 2023*, pages 611–626. ACM, 2023.

[76] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436–444, 2015.

[77] Haitong Li, Mudit Bhargava, Paul N Whatmough, and H-S Philip Wong. On-chip memory technology design space explorations for mobile deep neural network accelerators. In *Proceedings of the 56th Annual Design Automation Conference 2019*, pages 1–6, 2019.

[78] Li Li, Xiaorui Wang, and Feng Qin. Energydx: Diagnosing energy anomaly in mobile apps by identifying the manifestation point. In *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*, pages 256–266. IEEE, 2020.

[79] Yuxi Li. Deep reinforcement learning: An overview. *arXiv preprint arXiv:1701.07274*, 2017.

[80] Zhong-Zhi Li, Duzhen Zhang, Ming-Liang Zhang, Jiaxin Zhang, Zengyan Liu, Yuxuan Yao, Haotian Xu, Junhao Zheng, Pei-Jie Wang, Xiuyi Chen, Yingying Zhang, Fei Yin, Jiahua Dong, Zhiwei Li, Bao-Long Bi, Ling-Rui Mei, Junfeng Fang, Zhijiang Guo, Le Song, and Cheng-Lin Liu. From system 1 to system 2: A survey of reasoning large language models, 2025.

[81] Haicheng Liao, Zhenning Li, Huanming Shen, Wenxuan Zeng, Dongping Liao, Guofa Li, Shengbo Eben Li, and Chengzhong Xu. BAT: behavior-aware human-like trajectory prediction for autonomous driving. *CoRR*, abs/2312.06371, 2023.

[82] Edgar Liberis and Nicholas D Lane. Differentiable neural network pruning to enable smart applications on microcontrollers. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, 6(4):1–19, 2023.

[83] Chaofan Lin, Zhenhua Han, Chengruidong Zhang, Yuqing Yang, Fan Yang, Chen Chen, and Lili Qiu. Parrot: Efficient serving of llm-based applications with semantic variable. In Ada Gavrilovska and Douglas B. Terry, editors, *18th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2024, Santa Clara, CA, USA, July 10-12, 2024*, pages 929–945. USENIX Association, 2024.

[84] Ji Lin, Jiaming Tang, Haotian Tang, Shang Yang, Xingyu Dang, and Song Han. AWQ: activation-aware weight quantization for LLM compression and acceleration. *CoRR*, abs/2306.00978, 2023.

[85] Xue Lin, Yanzhi Wang, and Massoud Pedram. A reinforcement learning-based power management framework for green computing data centers. In *2016 IEEE International Conference on Cloud Engineering, IC2E 2016, Berlin, Germany, April 4-8, 2016*, pages 135–138. IEEE Computer Society, 2016.

[86] Jiahao Liu, Yuanzhe Zhao, Yan Zhu, Chi-Hang Chan, and Rui Paulo Martins. A weak puf-assisted strong PUF with inherent immunity to modeling attacks and ultra-low BER. *IEEE Trans. Circuits Syst. I Regul. Pap.*, 69(12):4898–4907, 2022.

[87] Pengfei Liu, Weizhe Yuan, Jinlan Fu, Zhengbao Jiang, Hiroaki Hayashi, and Graham Neubig. Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing. *ACM Comput. Surv.*, 55(9):195:1–195:35, 2023.

[88] Shu Liu, Asim Biswal, Audrey Cheng, Xiangxi Mo, Shiyi Cao, Joseph E Gonzalez, Ion Stoica, and Matei Zaharia. Optimizing llm queries in relational workloads. *arXiv preprint arXiv:2403.05821*, 2024.

[89] Xiao Liu, Kaixuan Ji, Yicheng Fu, Zhengxiao Du, Zhilin Yang, and Jie Tang. P-tuning v2: Prompt tuning

can be comparable to fine-tuning universally across scales and tasks. *CoRR*, abs/2110.07602, 2021.

[90] Zhi-Gang Liu, Paul N Whatmough, Yuhao Zhu, and Matthew Mattina. S2ta: Exploiting structured sparsity for energy-efficient mobile cnn acceleration. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 573–586. IEEE, 2022.

[91] Zichang Liu, Jue Wang, Tri Dao, Tianyi Zhou, Binhang Yuan, Zhao Song, Anshumali Shrivastava, Ce Zhang, Yuandong Tian, Christopher Ré, and Beidi Chen. Deja vu: Contextual sparsity for efficient llms at inference time. In Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett, editors, *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA*, volume 202 of *Proceedings of Machine Learning Research*, pages 22137–22176. PMLR, 2023.

[92] Daniel Lo, Taejoon Song, and G Edward Suh. Prediction-guided performance-energy trade-off for interactive applications. In *Proceedings of the 48th International Symposium on Microarchitecture*, pages 508–520, 2015.

[93] Mitchell P. Marcus, Beatrice Santorini, and Mary Ann Marcinkiewicz. Building a large annotated corpus of english: The penn treebank. *Comput. Linguistics*, 19(2):313–330, 1993.

[94] Xin Men, Mingyu Xu, Qingyu Zhang, Bingning Wang, Hongyu Lin, Yaojie Lu, Xianpei Han, and Weipeng Chen. Shortgpt: Layers in large language models are more redundant than you expect. *CoRR*, abs/2403.03853, 2024.

[95] Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. Pointer sentinel mixture models. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017.

[96] Thomas Mesnard, Cassidy Hardin, Robert Dadashi, Surya Bhupatiraju, Shreya Pathak, Laurent Sifre, Morgane Rivière, Mihir Sanjay Kale, Juliette Love, Pouya Tafti, Léonard Hussenot, Aakanksha Chowdhery, Adam Roberts, Aditya Barua, Alex Botev, Alex Castro-Ros, Ambrose Slone, Amélie Héliou, Andrea Tacchetti, Anna Bulanova, Antonia Paterson, Beth Tsai, Bobak Shahriari, Charline Le Lan, Christopher A. Choquette-Choo, Clément Crepy, Daniel Cer, Daphne Ippolito, David Reid, Elena Buchatskaya, Eric Ni, Eric Noland, Geng Yan, George Tucker, George-Christian Muraru, Grigory Rozhdestvenskiy, Henryk Michalewski, Ian Tenney, Ivan Grishchenko, Jacob Austin, James Keeling, Jane Labanowski, Jean-Baptiste Lespiau, Jeff Stanway, Jenny Brennan, Jeremy Chen, Johan Ferret, Justin Chiu, and et al. Gemma: Open models based on gemini research and technology. *CoRR*, abs/2403.08295, 2024.

[97] microsoft. Your Everyday AI Companion | Microsoft Bing. https://www.bing.com/new.

[98] Microsoft. Azure cognitive services - text analytics: Smart reply, 2024.

[99] Microsoft DeepSpeed Team. Deepspeed: Advancing the science of ai through efficient training of large models, 2024.

[100] Todor Mihaylov, Peter Clark, Tushar Khot, and Ashish Sabharwal. Can a suit of armor conduct electricity? A new dataset for open book question answering. In Ellen Riloff, David Chiang, Julia Hockenmaier, and Jun'ichi Tsujii, editors, *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing, Brussels, Belgium, October 31 - November 4, 2018*, pages 2381–2391. Association for Computational Linguistics, 2018.

[101] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

[102] Pavlo Molchanov, Arun Mallya, Stephen Tyree, Iuri Frosio, and Jan Kautz. Importance estimation for neural network pruning. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2019, Long Beach, CA, USA, June 16-20, 2019*, pages 11264–11272. Computer Vision Foundation / IEEE, 2019.

[103] Pavlo Molchanov, Stephen Tyree, Tero Karras, Timo Aila, and Jan Kautz. Pruning convolutional neural networks for resource efficient inference. *arXiv preprint arXiv:1611.06440*, 2016.

[104] nano. Nvidia jetson nano, 2023.

[105] Rajiv Nishtala, Vinicius Petrucci, Paul M. Carpenter, and Magnus Själander. Twig: Multi-agent task management for colocated latency-critical cloud services. In *IEEE International Symposium on High Performance Computer Architecture, HPCA 2020, San Diego, CA, USA, February 22-26, 2020*, pages 167–179. IEEE, 2020.

[106] NVIDIA. Nvidia jetson orin - autonomous machines - nvidia. https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-orin/, 2024.

[107] Hyungjun Oh, Kihong Kim, Jaemin Kim, Sungkyun Kim, Junyeol Lee, Du-seong Chang, and Jiwon Seo. Exegpt: Constraint-aware resource scheduling for LLM inference. In Rajiv Gupta, Nael B. Abu-Ghazaleh, Madan Musuvathi, and Dan Tsafrir, editors, *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2024, La Jolla, CA, USA, 27 April 2024- 1 May 2024*, pages 369–384. ACM, 2024.

[108] OpenAI. Chatgpt, 2022.

[109] OpenAI. GPT-4 technical report. *CoRR*, abs/2303.08774, 2023.

[110] Otter.ai. Otter: Transcription and note-taking with ai, 2024.

[111] Pratyush Patel, Esha Choukse, Chaojie Zhang, Íñigo Goiri, Aashaka Shah, Saeed Maleki, and Ricardo Bianchini. Splitwise: Efficient generative LLM inference using phase splitting. *CoRR*, abs/2311.18677, 2023.

[112] Leonardo Piga, Iyswarya Narayanan, Aditya Sundarrajan, Matt Skach, Qingyuan Deng, Biswadip Maity, Manoj Chakkaravarthy, Alison Huang, Abhishek Dhanotia, and Parth Malani. Expanding datacenter capacity with DVFS boosting: A safe and scalable deployment experience. In Rajiv Gupta, Nael B. Abu-Ghazaleh, Madan Musuvathi, and Dan Tsafrir, editors, *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1, ASPLOS 2024, La Jolla, CA, USA, 27 April 2024- 1 May 2024*, pages 150–165. ACM, 2024.

[113] PyTorch Contributors. Pytorch: An open source machine learning framework, 2024.

[114] Qualcomm Technologies, Inc. Llama-v2-7B-Chat Quantized Model. https://aihub.qualcomm.com/models/llama_v2_7b_chat_quantized, 2024. Qualcomm AI Hub, 2024.

[115] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

[116] Keisuke Sakaguchi, Ronan Le Bras, Chandra Bhagavatula, and Yejin Choi. Winogrande: An adversarial winograd schema challenge at scale. *arXiv preprint arXiv:1907.10641*, 2019.

[117] Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc V. Le, Geoffrey E. Hinton, and Jeff Dean. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017.

[118] Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Beidi Chen, Percy Liang, Christopher Ré, Ion Stoica, and Ce Zhang. Flexgen: High-throughput generative inference of large language models with a single gpu. In *International Conference on Machine Learning*, pages 31094–31116. PMLR, 2023.

[119] Ishika Singh, Valts Blukis, Arsalan Mousavian, Ankit Goyal, Danfei Xu, Jonathan Tremblay, Dieter Fox, Jesse Thomason, and Animesh Garg. Progprompt: Generating situated robot task plans using large language models. In *IEEE International Conference on Robotics and Automation, ICRA 2023, London, UK, May 29 - June 2, 2023*, pages 11523–11530, London, UK, 2023. IEEE.

[120] Karan Singhal, Shekoofeh Azizi, Tao Tu, S. Sara Mahdavi, Jason Wei, Hyung Won Chung, Nathan Scales, Ajay Kumar Tanwani, Heather Cole-Lewis, Stephen Pfohl, Perry Payne, Martin Seneviratne, Paul Gamble, Chris Kelly, Nathaneal Schärli, Aakanksha Chowdhery, Philip Andrew Mansfield, Blaise Agüera y Arcas, Dale R. Webster, Gregory S. Corrado, Yossi Matias, Katherine Chou, Juraj Gottweis, Nenad Tomasev, Yun Liu, Alvin Rajkomar, Joelle K. Barral, Christopher Semturs, Alan Karthikesalingam, and Vivek Natarajan. Large language models encode clinical knowledge. *CoRR*, abs/2212.13138, 2022.

[121] Chan Hee Song, Jiaman Wu, Clayton Washington, Brian M Sadler, Wei-Lun Chao, and Yu Su. Llm-planner: Few-shot grounded planning for embodied agents with large language models. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 2998–3009, 2023.

[122] Yixin Song, Zeyu Mi, Haotong Xie, and Haibo Chen. Powerinfer: Fast large language model serving with a consumer-grade gpu. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, pages 590–606, 2024.

[123] Aarohi Srivastava, Abhinav Rastogi, Abhishek Rao, Abu Awal Md Shoeb, Abubakar Abid, Adam Fisch, Adam R. Brown, Adam Santoro, Aditya Gupta, Adrià Garriga-Alonso, Agnieszka Kluska, Aitor Lewkowycz, Akshat Agarwal, Alethea Power, Alex Ray, Alex Warstadt, Alexander W. Kocurek, Ali Safaya, Ali Tazarv, Alice Xiang, Alicia Parrish, Allen Nie, Aman Hussain, Amanda Askell, Amanda Dsouza, Ameet Rahane, Anantharaman S. Iyer, Anders Andreassen, Andrea Santilli, Andreas Stuhlmüller, Andrew M. Dai,

Andrew La, Andrew K. Lampinen, Andy Zou, Angela Jiang, Angelica Chen, Anh Vuong, Animesh Gupta, Anna Gottardi, Antonio Norelli, Anu Venkatesh, Arash Gholamidavoodi, Arfa Tabassum, Arul Menezes, Arun Kirubarajan, Asher Mullokandov, Ashish Sabharwal, Austin Herrick, Avia Efrat, Aykut Erdem, Ayla Karakas, and et al. Beyond the imitation game: Quantifying and extrapolating the capabilities of language models. *CoRR*, abs/2206.04615, 2022.

[124] stanford. Ai index report. https://aiindex.stanford.edu/report/, 2024. Accessed: 2024-07.

[125] Statista Inc. Mobile ram usage worldwide from 1q-19 to 1q-21 (in gb per device). www.statista.com/statistics/1057679/mobile-ram-usage-worldwide-by-average-size-per-device/, 2021.

[126] Kahou Tam, Chunlin Tian, Li Li, Haikai Zhao, and ChengZhong Xu. Fedhybrid: Breaking the memory wall of federated learning via hybrid tensor management. In *Proceedings of the 22nd ACM Conference on Embedded Networked Sensor Systems*, pages 394–408, 2024.

[127] Thierry Tambe, Coleman Hooper, Lillian Pentecost, Tianyu Jia, En-Yu Yang, Marco Donato, Victor Sanh, Paul Whatmough, Alexander M Rush, David Brooks, et al. Edgebert: Sentence-level energy optimizations for latency-aware multi-task nlp inference. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 830–844, 2021.

[128] TensorFlow Contributors. Tensorflow: An open source machine learning framework for everyone, 2024.

[129] Arun James Thirunavukarasu, Darren Shu Jeng Ting, Kabilan Elangovan, Laura Gutierrez, Ting Fang Tan, and Daniel Shu Wei Ting. Large language models in medicine. *Nature medicine*, 29(8):1930–1940, 2023.

[130] Chunlin Tian, Li Li, Zhan Shi, Jun Wang, and ChengZhong Xu. Harmony: Heterogeneity-aware hierarchical management for federated learning system. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 631–645. IEEE, 2022.

[131] Chunlin Tian, Zhan Shi, Zhijiang Guo, Li Li, and Cheng-Zhong Xu. Hydralora: An asymmetric lora architecture for efficient fine-tuning. *Advances in Neural Information Processing Systems*, 37:9565–9584, 2024.

[132] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurélien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. Llama: Open and efficient foundation language models. *CoRR*, abs/2302.13971, 2023.

[133] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton-Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurélien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. Llama 2: Open foundation and fine-tuned chat models. *CoRR*, abs/2307.09288, 2023.

[134] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pages 5998–6008, 2017.

[135] Jason Wei, Maarten Bosma, Vincent Y. Zhao, Kelvin Guu, Adams Wei Yu, Brian Lester, Nan Du, Andrew M. Dai, and Quoc V. Le. Finetuned language models are zero-shot learners. *CoRR*, abs/2109.01652, 2021.

[136] Jason Wei, Yi Tay, Rishi Bommasani, Colin Raffel, Barret Zoph, Sebastian Borgeaud, Dani Yogatama, Maarten Bosma, Denny Zhou, Donald Metzler, Ed H. Chi, Tatsunori Hashimoto, Oriol Vinyals, Percy Liang, Jeff Dean, and William Fedus. Emergent abilities of large language models. *Trans. Mach. Learn. Res.*, 2022, 2022.

[137] Xiuying Wei, Yunchen Zhang, Yuhang Li, Xiangguo Zhang, Ruihao Gong, Jinyang Guo, and Xianglong Liu. Outlier suppression+: Accurate quantization of large

language models by equivalent and effective shifting and scaling. In Houda Bouamor, Juan Pino, and Kalika Bali, editors, *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing, EMNLP 2023, Singapore, December 6-10, 2023*, pages 1648–1665. Association for Computational Linguistics, 2023.

[138] Hao Wen, Yuanchun Li, Guohong Liu, Shanhui Zhao, Tao Yu, Toby Jia-Jun Li, Shiqi Jiang, Yunhao Liu, Yaqin Zhang, and Yunxin Liu. Autodroid: Llm-powered task automation in android. In *Proceedings of the 30th Annual International Conference on Mobile Computing and Networking (MobiCom '24)*, Washington D.C., USA, 2024. Association for Computing Machinery.

[139] Carole-Jean Wu, David Brooks, Kevin Chen, Douglas Chen, Sy Choudhury, Marat Dukhan, Kim Hazelwood, Eldad Isaac, Yangqing Jia, Bill Jia, et al. Machine learning at facebook: Understanding inference at the edge. In *2019 IEEE international symposium on high performance computer architecture (HPCA)*, pages 331–344. IEEE, 2019.

[140] Yebo Wu, Li Li, Chunlin Tian, Tao Chang, Chi Lin, Cong Wang, and Cheng-Zhong Xu. Heterogeneity-aware memory efficient federated learning via progressive layer freezing. In *2024 IEEE/ACM 32nd International Symposium on Quality of Service (IWQoS)*, pages 1–10. IEEE, 2024.

[141] Yebo Wu, Chunlin Tian, Jingguang Li, He Sun, Kahou Tam, Li Li, and Chengzhong Xu. A survey on federated fine-tuning of large language models. *arXiv preprint arXiv:2503.12016*, 2025.

[142] Yebo Wu, Chunlin Tian, Jingguang Li, He Sun, Kahou Tam, Li Li, and Chengzhong Xu. A survey on federated fine-tuning of large language models. *CoRR*, abs/2503.12016, 2025.

[143] Guangxuan Xiao, Ji Lin, Mickaël Seznec, Hao Wu, Julien Demouth, and Song Han. Smoothquant: Accurate and efficient post-training quantization for large language models. In Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett, editors, *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA*, volume 202 of *Proceedings of Machine Learning Research*, pages 38087–38099. PMLR, 2023.

[144] Zhenjie Yang, Xiaosong Jia, Hongyang Li, and Junchi Yan. Llm4drive: A survey of large language models for autonomous driving. *arXiv preprint arXiv:2311.01043*, 2023.

[145] Xinyu Ye, Zhe Wang, Haihao Shen, Yu Luo, and Hanwen Chang. Creating large language models on your laptop. Medium, 2023.

[146] Rongjie Yi, Xiang Li, Weikai Xie, Zhenyan Lu, Chenghua Wang, Ao Zhou, Shangguang Wang, Xiwen Zhang, and Mengwei Xu. Phonelm: An efficient and capable small language model family through principled pre-training. *arXiv preprint arXiv:2411.05046*, 2024.

[147] Wangsong Yin, Rongjie Yi, Daliang Xu, Gang Huang, Mengwei Xu, and Xuanzhe Liu. Elms: Elasticized large language models on mobile devices. *arXiv preprint arXiv:2409.09071*, 2024.

[148] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. Orca: A distributed serving system for transformer-based generative models. In Marcos K. Aguilera and Hakim Weatherspoon, editors, *16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2022, Carlsbad, CA, USA, July 11-13, 2022*, pages 521–538. USENIX Association, 2022.

[149] Jinliang Yuan, Chen Yang, Dongqi Cai, Shihe Wang, Xin Yuan, Zeling Zhang, Xiang Li, Dingge Zhang, Hanzi Mei, Xianqing Jia, et al. Mobile foundation model as firmware. *arXiv preprint arXiv:2308.14363*, 2023.

[150] Ted Zadouri, Ahmet Üstün, Arash Ahmadian, Beyza Ermis, Acyr Locatelli, and Sara Hooker. Pushing mixture of experts to the limit: Extremely parameter efficient moe for instruction tuning. *CoRR*, abs/2309.05444, 2023.

[151] Rowan Zellers, Ari Holtzman, Yonatan Bisk, Ali Farhadi, and Yejin Choi. Hellaswag: Can a machine really finish your sentence? In Anna Korhonen, David R. Traum, and Lluís Màrquez, editors, *Proceedings of the 57th Conference of the Association for Computational Linguistics, ACL 2019, Florence, Italy, July 28- August 2, 2019, Volume 1: Long Papers*, pages 4791–4800. Association for Computational Linguistics, 2019.

[152] Shichen Zhan, Yebo Wu, Chunlin Tian, Yan Zhao, and Li Li. Heterogeneity-aware coordination for federated learning via stitching pre-trained blocks. In *2024 IEEE/ACM 32nd International Symposium on Quality of Service (IWQoS)*, pages 1–10. IEEE, 2024.

[153] Yifan Zhao, Hashim Sharif, Vikram Adve, and Sasa Misailovic. Felix: Optimizing tensor programs with gradient descent. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pages 367–381, 2024.

[154] Yifan Zhao, Hashim Sharif, Peter Pao-Huang, Vatsin Shah, Arun Narenthiran Sivakumar, Mateus Valverde Gasparino, Abdulrahman Mahmoud, Nathan Zhao, Sarita Adve, Girish Chowdhary, et al. Approxcaliper: A programmable framework for application-aware neural network optimization. *Proceedings of Machine Learning and Systems*, 5, 2023.

[155] Yuanzhe Zhao, Pengyu He, Yan Zhu, Rui P Martins, Chi-Hang Chan, and Minglei Zhang. A 28-nm 3.32-nj/frame compute-in-memory cnn processor with layer fusion for always-on applications. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 2025.

[156] Yuanzhe Zhao, Yang Wang, Yuheng Wang, Heng Xie, Yan Zhu, RP Martins, Chi-Hang Chan, Shouyi Yin, and Minglei Zhang. A 28nm value-wise hybrid-domain compute-in-memory macro with heterogeneous memory fabric and asynchronous sparsity manager. In *2025 IEEE Custom Integrated Circuits Conference (CICC)*, pages 1–3. IEEE, 2025.

[157] Yuanzhe Zhao, Yuheng Wang, Zijian Wang, Yan Zhu, RP Martins, Chi-Hang Chan, and Minglei Zhang. A reconfigurable 0.69-1.02 nj/classification biomedical ai processor for intelligent health monitoring devices. In *2025 IEEE Custom Integrated Circuits Conference (CICC)*, pages 1–3. IEEE, 2025.

[158] Yuanzhe Zhao, Heng Xie, Zijian Wang, Chunlin Tian, Li Li, Yan Zhu, RP Martins, Chi-Hang Chan, and Minglei Zhang. A one-shot floating-point compute-in-memory macro featuring pvt robustness and mismatch tolerance for edge llms. In *2025 IEEE Custom Integrated Circuits Conference (CICC)*, pages 1–3. IEEE, 2025.

[159] Yuanzhe Zhao, Minglei Zhang, Pengyu He, Yan Zhu, Chi-Hang Chan, and Rui Paulo Martins. A double-mode sparse compute-in-memory macro with reconfigurable single and dual layer computation. In *IEEE Custom Integrated Circuits Conference, CICC 2023, San Antonio, TX, USA, April 23-26, 2023*, pages 1–2. IEEE, 2023.

[160] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric P. Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. Judging llm-as-a-judge with mt-bench and chatbot arena. In Alice Oh, Tristan Naumann, Amir Globerson, Kate Saenko, Moritz Hardt, and Sergey Levine, editors, *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*, 2023.

[161] Yuhao Zhu, Matthew Halpern, and Vijay Janapa Reddi. Event-based scheduling for energy-efficient qos (eqos) in mobile web applications. In *21st IEEE International Symposium on High Performance Computer Architecture, HPCA 2015, Burlingame, CA, USA, February 7-11, 2015*, pages 137–149. IEEE Computer Society, 2015.

[162] Ömer Bilgin Bilgili. Ai robotics case - controlling robots with llms (large language models). acrome, 2024. Acrome Robotics Blog, October 23, 2024.

WEAVER: Efficient Multi-LLM Serving with Attention Offloading

Shiwei Gao, Qing Wang, Shaoxun Zeng, Youyou Lu and Jiwu Shu*

Tsinghua University

Abstract

LLM serving platforms typically provide services for tens to hundreds of different models, where a small number of hot models receive the majority of the requests, while most other models remain cold. Yet, current serving systems can not efficiently handle such workloads: Using dedicated instances for hot and cold models makes the GPU memory underutilized, and multiplexing different models with model parallelism introduces communication overhead.

We propose a mechanism called *workload weaving*, which offloads attention operators of hot models to *running* cold models, achieving high GPU memory utilization with low communication cost. To mitigate the blocking caused by running cold models, we propose WEAVER with two key techniques: (i) GPU-driven dynamic control flow, which delegates the control logic of offloading to GPUs, letting the offloaded operators bypass pending kernels in the GPU hardware queue; (ii) operator splitting, which carefully divides the large kernels of cold models into smaller ones to mitigate the head-of-line blocking. Our evaluation using real-world LLM trace demonstrates that WEAVER improves the throughput of hot models by up to 77% while maintaining the same or lower TPOT. For the cold model, WEAVER incurs a modest overhead (3-5ms).

1 Introduction

Large Language Models (LLMs) [1, 16, 18, 29, 30] have transformed numerous industries, leading to rapid adoption and innovation. As LLMs continue to evolve, serving platforms (e.g., Hugging Face [5], Together.ai [11]) now commonly host dozens or even hundreds of models, creating a new paradigm of *multi-LLM serving*. Multi-LLM serving features a skewed access pattern [17], where a small number of hot models receive the majority of requests, while most models remain rarely accessed (i.e., cold). This skewed pattern is also demonstrated by our analysis of a popular MaaS platform (§2.2).

However, through analysis and experiments, we find that existing methods can not efficiently handle the skewed workloads in multi-LLM serving (§2.3). A simple method is allocating dedicated GPU instances for hot and cold models respectively. Yet, it leads to poor GPU memory utilization for cold models due to their low request load. Another method is multiplexing, which shares GPUs between hot and cold models using model parallelism. Examples include AlpaServe [22], which employs model parallelism to share GPUs across models, and MuxServe [17], which enhances model parallelism with NVIDIA MPS [7] for improved spatial-temporal sharing. While these multiplexing methods can improve GPU utilization, they introduce non-negligible communication overhead, thus impacting overall system performance.

We observe that the inefficiency of existing multiplexing methods stems from the lack of exploiting LLM structures: they are designed for general DNN models. Motivated by it, we propose *workload weaving*, a mechanism that offloads a portion of attention, which are non-parameterized parts in LLMs that can be efficiently remotely executed, to GPUs running cold models. By doing so, we can allow hot models to harvest the underutilized GPU memory of cold models, enabling larger batch size and thus higher throughput. Besides, workload weaving has lower communication overhead compared to multiplexing methods.

However, making workload weaving practical is challenging. Different from offloading attention to dedicated computing devices [13, 19, 20, 23, 24], in workload weaving, the destination of attention offloading, i.e., the cold instance, is also responsible for serving its model. This makes the offloaded attention operators experience head-of-line blocking from *pre-issued kernels* and *long-running kernels*. First, existing LLM serving systems typically pre-issue hundreds of kernels to the GPU hardware queue. Therefore, when an offloaded attention operator arrives at the cold instance, it can only run on the GPU if all the pre-issued kernels are drained. Second, even if we could prioritize the execution of the offloaded attention operator, it might still be blocked when the GPU is being occupied with a long-running kernel.

We design WEAVER, a multi-LLM serving system that achieves efficient workload weaving while overcoming the above challenges. We guarantee that offloaded attention operators will wait for *at most one small running kernel* to finish, which ensures the efficiency of WEAVER. This is achieved by two techniques we propose:

1) GPU-driven dynamic control flow. WEAVER delegates the control logic of issuing offloaded attention operators to GPUs. Specifically, WEAVER leverages cross-GPU shared memory capabilities to dispatch offload tasks. The GPU in the cold instance continuously polls new offload tasks and executes them immediately, bypassing pre-issued kernels in the GPU hardware queue.

2) Operator splitting. WEAVER addresses the long-running kernel problem by splitting a large kernel into smaller ones, so

*Jiwu Shu is the corresponding author (shujw@tsinghua.edu.cn).

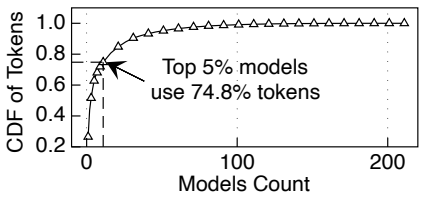


Figure 1: Model Workload Statictic of a MaaS Platform.

that the offloaded attention can be executed on GPUs within a bounded time. To mitigate the kernel fragmentation caused by splitting, we adopt a priority-based algorithm based on queuing theory analysis to generate the splitting plan.

We evaluate WEAVER on two real-world LLM traces (Azure-Conv, BurtGPT) and two hardware platforms (A100, L40S). Our results show that WEAVER improves the throughput of hot models by up to 77% against dedicated serving and 60% against multiplexing-based methods while maintaining the same or lower TPOT. For the cold model, WEAVER incurs a modest overhead (3-5ms) on average in the TPOT compared to dedicated serving.

2 Background and Motivation

2.1 Operators in LLM Serving

Due to the unique architecture of the Transformer, the LLM serving mainly involves two types of operators: parameterized *projection* and non-parameterized *attention*.

Projection operators. These operators include fully connected feed-forward networks (FFNs) and specialized linear transformations, such as query, key, value, and output projections in the multi-head attention mechanism. As a parameterized component, they rely on a fixed set of learned parameters to process input tokens or their intermediate representations.

Attention operators. Attention operators dynamically compute relationships between input tokens to capture contextual dependencies. Unlike projection operators, the attention mechanism is non-parameterized. Therefore, the attention computation does not require model parameters, but depends on the KV tensors of previous tokens. These KV tensors are typically saved in the form of *KV cache*, to avoid duplicated compuation in future token generation.

2.2 Multi-LLM Serving

LLMs have become popular across various industries, driving innovation and adoption at an unprecedented pace. As a result, the variety of available LLMs has grown rapidly, and thus LLM serving platforms now usually host tens to hundreds of LLMs, presenting the paradigm toward *multi-LLM serving*.

For instance, Model-as-a-Service (MaaS) platforms, such as Hugging Face [5], Together.ai [11], and OpenRouter [9], provide services of a broad range of models, catering to users with different requirements and use cases. Even single-model providers, e.g., OpenAI [8] and Gemini [3], host a family of models with varying capabilities, model sizes, and release versions, enabling users to select the most appropriate model for their specific tasks: OpenAI alone provides APIs for more than 20 variants of its chat family LLMs [8].

Batch Size	2 × Seperate (Token/s)	TP 2 (Token/s)	Slowdown
16	2024	1487	26.6%
32	3343	2645	20.9%
64	5392	4443	17.6%
128	8011	6606	17.5%

Table 1: Tensor Parallelism Overhead. *The results are tests on two A100-40GB with NVLink, Llama-3-8B, input 1024, output 128.*

Multi-LLM serving systems feature skewed access patterns [17]. We analyze a one-week multi-LLM serving statistic of a popular MaaS platform OpenRouter [9], which includes 209 models and more than 200 billion tokens. Figure 1 shows the CDF of request tokens: the top 5% models consume 74.8% tokens, exhibiting a skewed distribution. In other words, in multi-LLM serving, a small subset of hot models (e.g., Llama-3 or OpenAI's latest release) receives the majority of traffic, but many models are cold and see sparse usage; they receive a low but consistent request rate most of the time.

2.3 Existing Methods and Limitations

By analysis and experiments, we find that existing methods can not efficiently handle workloads with mixed hot and cold models, a remarkable feature of multi-LLM serving (§2.2).

Using dedicated instances. This method uses dedicated instances for hot and cold models respectively. For example, service providers can deploy hot models on dedicated GPUs [10], while dynamically allocating GPUs for cold models (e.g., leverage serverless). However, during processing requests for cold models, the associated GPUs suffer from low GPU memory utilization. We conduct an experiment, where a hot model (15 QPS, Llama-3-8B) and a cold model (1 QPS, Llama-3-8B) run on an A100-40GB GPU *each*, and measure their GPU memory utilization. While the hot model already utilizes all the GPU memory, the GPU memory utilization of the cold model remains at 43%.

Multiplexing. Another method is multiplexing GPUs for cold and hot models. AlpaServe [22] uses model parallelism for statistically multiplexing GPUs for multiple models. MuxServe [17] further improves it by incorporating NVIDIA MPS [7], thus achieving flexible temporal-spatial sharing. Micro-batching causes redundant LLM weight access [14], making pipeline parallelism less efficient and less commonly used for models that can fit on a single multi-GPU node. Both MuxServe and our work mainly focus on tensor parallelism as the primary model parallelism strategy. While tensor parallelism improves the GPU utilization, it also introduces overhead caused by communication and smaller (thus less efficient) operators. In Table 1, we evaluate the token generation throughput of the Llama-3-8B model with 2 separate instances and TP degree of 2 setting on the 2× A100-40GB GPU connected with NVLink. Results show that naively using TP can introduce up to 26% overhead even on a high

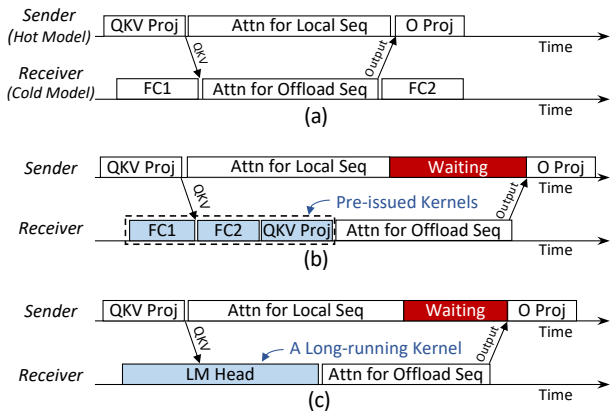


Figure 2: Workload Weaving and Challenges. *(a) Ideal case. (b) Blocked by pre-issued kernels. (c) Blocked by long-running kernels.*

interconnect bandwidth platform.

3 Approach and Challenges

Opportunity: attention offload. Attention offload is a technique that delegates attention operators to *dedicated* auxiliary computing devices [13,19,20,23,24], such as GPU, CPU, PIM, or CSD. The primary device (e.g., GPU) executes projection operators, and stores the sequence's KV cache on auxiliary devices. Attention offload provides a way to offload part of the computation and storage usage of LLM inference with low communication and memory overhead, this is because 1) for each attention computation, only the QKV tensor of the latest token needs to be transmitted to the auxiliary device, 2) the auxiliary device does not need to store model parameters (recall attention operators are non-parameterized, §2.1).

Our approach: workload weaving. Inspired by attention offloading, we propose *workload weaving* to handle multi-LLM serving workloads and overcome limitations of existing methods (§2.3). Workload weaving harvests the unutilized GPU memory of running cold models by offloading a portion of attention operators to them, and Figure 2a shows its workflow. During inference, the sender (i.e., a GPU running hot models) transmits the QKV tensor of the offloaded sequences's new tokens to the receiver (i.e., a GPU running hot models). The sender executes attention operators for the non-offloaded portion, while the receiver handles the offloaded portion and then returns results to the sender for final merging. The receiver also maintains the KV cache for the offloaded portion.

Workload weaving offers two advantages for multi-LLM serving. First, it can improve the throughput of hot models by utilizing the unused GPU memory of cold models, thereby enabling larger batch sizes. Second, it achieves lower communication overhead compared to model parallelism used in the multiplexing method.

3.1 Challenges

In the ideal case of workload weaving, the cold instance (i.e., receiver) can immediately execute the attention operators from the hot instance (i.e., sender), as shown in Figure 2a. Yet, realizing it is challenging, because the cold instance is also responsible for serving its model, making the offloaded attention operators suffer from head-of-line blocking from 1) pre-issued kernels and 2) long-running kernels.

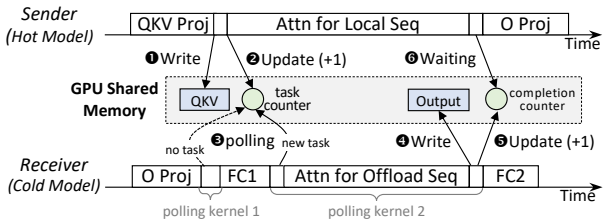


Figure 3: GPU-driven Dynamic Control Flow.

Blocked by pre-issued kernels. In existing LLM serving systems, CPUs perform the control flow logic, mainly including issuing kernels to the GPU. They typically pre-issue many kernels to the GPU (e.g., 387 kernels per iteration of Llama-3-8B in vLLM), each representing an operator. When these kernels enter the GPU hardware queue, the GPU hardware will execute them sequentially. Therefore, when an offloaded attention operator is launched by the receiver CPU, *there is no guarantee that it will be the next kernel to be executed*. For example, in Figure 2b, the offloaded operator is blocked by `FC2` and `QKV Proj`, two unexecuted but issued kernels.

Blocked by long-running kernels. Even if we can guarantee the offloaded attention operator is the next kernel to execute, it may still be blocked for a long time. This is because the receiver GPU may be executing a long-running kernel, as shown in Figure 2c. Long-running kernels are common in LLM serving: for example, the LM head of Llama-3-8B takes up to 961µs on an A100 GPU.

4 Design and Implementation

We propose WEAVER, a multi-LLM serving system that achieves efficient workload weaving while overcoming the associated challenges (§3.1).

WEAVER incorporates two key techniques, *GPU-driven dynamic control flow*(§4.1) and *operator splitting* (§4.2). The former delegates the control logic of attention offloading to the GPU, so that offloaded attention operators can bypass pre-issued kernels in the GPU hardware queue. The latter splits large operators into small ones using a priority-based algorithm backed by queueing theory, avoiding head-of-line blocking from a long-running kernel. Combining the two techniques, WEAVER can guarantee that offloaded attention will wait for at most one small running kernel to finish.

4.1 GPU-driven Dynamic Control Flow

To avoid the offloaded attention operator being blocked by pre-issued kernels, one strawman approach would be to issue kernels in a blocking manner: only when a kernel is complete, the receiver CPU will issue the next kernel (prioritize issuing the offload attention operator, if it exists). However, this

approach significantly impacts the receiver GPU's computational throughput. WEAVER proposes GPU-driven dynamic control flow to avoid the blocking problem of pre-issued kernels, while minimizing the impact on GPU performance. It delegates the control logic of issuing offloaded attention operators to the receiver GPU, and leverages cross-GPU shared memory capabilities [2, 4] for synchronization.

With GPU-driven dynamic control flow, the process of offloading an attention operator is shown in Figure 3. When the sender needs to offload an attention operator, it writes the latest token's QKV tensor to shared memory using one-sided writes (❶), and atomically increments a task counter (❷). The receiver GPU polls the task counter (❸). Upon detecting an increment, indicating a new task, the receiver GPU executes the attention operator, then writes the result tensor to shared memory (❹), and finally increments a completion counter (❺). Once the sender realizes that the completion counter has changed (❻), it obtains the result tensor from shared memory and then continues with the subsequent computation.

All the above operations of the receiver GPU, including polling, counter updates, data transfers, and attention execution, are encapsulated in a GPU kernel, which we call *polling kernel*. The receiver CPU pre-issues many kernels as in today's LLM serving systems to saturate the GPU hardware queue, but *each kernel is followed by a polling kernel*. In this way, the offloaded attention will only be blocked by the kernel that is executing, not other pre-issued kernels.

Of note, the updates of attention metadata (e.g., sequence page tables) are still executed on the CPU due to their complexity. These updates are infrequent (i.e., only one time per iteration) and thus do not create performance bottlenecks.

4.2 Operator Splitting

To avoid head-of-line blocking resulting from a long-running kernel, WEAVER introduces operator splitting. Its key idea is to split the kernel of a large operator into smaller ones, so that the offloaded attention can be executed on GPUs within a bounded time. However, it is non-trivial to generate a good splitting plan, considering that 1) the operators in LLM serving have diverse execution time, 2) excessive fragmentation due to operator splitting will induce performance overhead. To this end, we first employ queuing theory to model the sender's wait time, then adopt a priority-based algorithm to generate the splitting plan.

Modeling. We model the receiver as a polling system [28] with two logical queues. The first queue accommodates attention operators offloaded from the sender, arriving at intervals of M_i time units, each requiring a service time of N_i. The second queue accommodates the receiver's LLM operators, assumed to have an infinite backlog, with each operator requiring a service time of T_i. For the non-offloaded portion of the attention, the processing time in the sender is K_i.

Our objective is to minimize the conditional expected wait time experienced by offloaded attention operatiors in the first queue (i.e., W_i). Specifically:

$$\min E(W_i \cdot \mathbf{1}_{\{K_i < W_i + N_i\}})$$

Here, the condition $K_i < W_i + N_i$ means that: if the execution time of non-offloaded attention is less than the sum of receiver-side wait time and the execution time of offloaded attention, the sender will be blocked. To simplify the problem, we assume that the K_i and N_i remain constant (term as K and N), which can be obtained by periodic profiling. Based on statistical analysis, the completion position of each offloaded operator can be treated as a random variable uniformly distributed in $[0, T_i]$. Under this assumption, the wait time caused by each operator is:

$$W_i = \frac{\max\{T_i - (K-N), 0\}^2}{2T_i}$$

The probability of an operator being at the queue head is proportional to its time density in the system. Thus, the overall objective becomes:

$$min \frac{\max\{T_i - (K-N), 0\}^2}{2\sum T_i}$$

Priority-based operator splitting. The above formulation reveals that large operators contribute quadratically to the receiver's expected wait time. According to it, we propose a priority-based operator splitting algorithm, as described in Algorithm 1. We maintain a priority queue of all operators in the receiver and iteratively split the operator with the longest execution time into two halves. If the split operator still exceeds the $(K-N)$, it is reinserted into the queue. This process continues until the wait time is reduced below a pre-defined threshold $WT_{threshold}$ (we set it to 5% of the sender's average iteration time). With this algorithm, we can alleviate head-of-line blocking without generating many small kernels.

WEAVER executes the algorithm on the receiver-side CPU. WEAVER splits parameterized operators in the parameter dimension, and non-parameterized operators in the sequence dimension. Note that currently WEAVER fixes the proportion of offloaded attention. K and N are periodically updated via workload profiling.

4.3 Implementation Details

We implement WEAVER based on vLLM v0.6.0 [21]. We use the CUDA IPC for cross-GPU shared memory communication and FlashAttention [15] to execute the offloaded attention.

Request scheduling. The hot instance distributes the task of processing input sequences to itself and the cold instance, using a fixed offload ratio. After completing the prefill stage, the KV cache of each hot model's request is randomly assigned to either the hot or cold instance, where it remains until the request is complete. Since our work focuses on developing an efficient mechanism for serving multiple models, we leave the design of a cluster-level scheduler for future work. For instance, using request rate metrics, one could dynamically

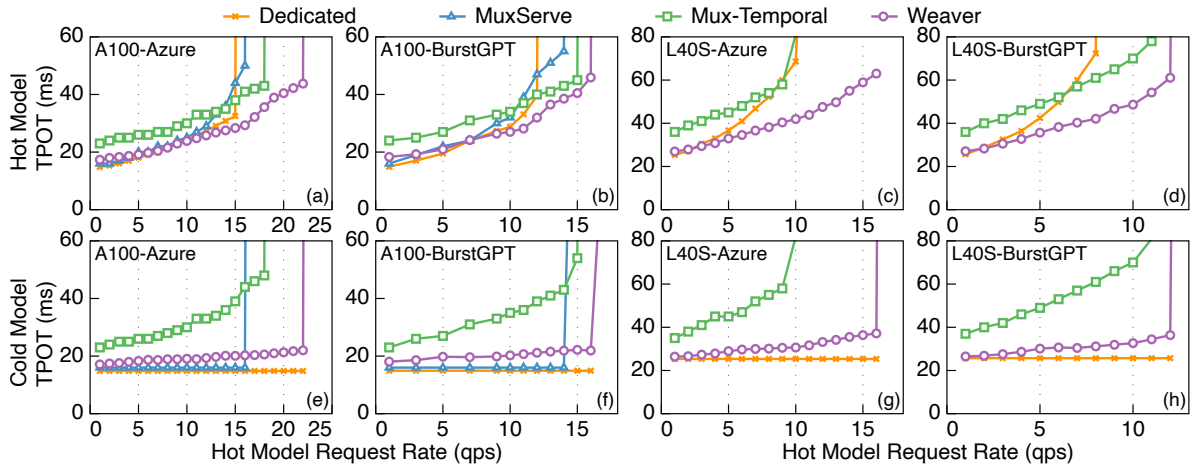


Figure 4: Average TPOT of Different Multi-LLM Serving Methods On Two Platforms with Real-world Workloads.

Algorithm 1 Priority-Based Operator Splitting.

Input: Priority queue Q, which contains all operators in reverse order of T_i; Wait time threshold $WT_{\text{threshold}}$.
Output: A queue of operators after splitting.
while Estimated wait time $> WT_{\text{threshold}}$ **do**
    $OP_{\max} \leftarrow Q$.front();
    if $T_{\max} > K - N$ **then**
        $OP_1, OP_2 \leftarrow$ split($OP_{\max}$);
        Q.pop();
        Q.push(OP_1, OP_2);
    else
        break
return Q

adjust the offload ratio, reducing the offload ratio when the cold model's load is high and increasing it when the load is low, to enable dynamic harvesting of cold model resources.

KV cache management. In WEAVER, the GPU memory of the cold model's instance holds both KV cache blocks for offloaded sequences from the hot model and ones for its own sequences. Following prior work [17], we build a unified KV block allocator that supports models with different numbers of layers and KV heads.

4.4 Discussion

Influence of the model size. For larger models, GPU memory, especially for the KV cache, becomes a more significant constraint, often limiting the maximum batch size. In such a situation, WEAVER allows the hot model to leverage the memory resources of the cold model's GPU instance. This enables larger effective batch sizes for the hot model, leading to higher maximum throughput. For small language models, its benefits could be less significant. WEAVER can not speed up the FFN part of the model. For small language models and larger batch sizes, the projection part could be more dominant for the end-to-end performance. We leave the exploration of workload offloading for the FFN part as future work.

5 Evaluation

Workloads and test models. We evaluate our system using two widely-used LLM workload traces: BurstGPT [31] (ChatGPT-4 split) and Azure-Conv [25]. We filter out sequences exceeding 2048 tokens, since some baselines do not support running such lengths. BurstGPT represents a balanced workload, with an average input length of 575 tokens and an output length of 340 tokens. In contrast, Azure-Conv represents an input-heavy workload, with an average input length of 749 tokens and an output length of 232 tokens. All experiments use the Llama-3-8B model by default. Following prior work [21], we sample the arrival time of requests with the Poisson distribution from the above traces. Unless otherwise stated, we set the request rate of the cold model to 1 request per second and the offload ratio of WEAVER to 45%.

Baselines. We compare WEAVER to the following baselines. ❶ Dedicated Serving, which uses dedicated GPUs to serve cold and hot models respectively, each running unmodified vLLM. ❷ MuxServe [17], which supports two modes: 1) spatial-temporal multiplexing with NVIDIA MPS enabled and 2) temporary multiplexing only (i.e., Mux-Temporal). For a fair comparison with MuxServe, we do not enable CUDA Graph in WEAVER and dedicated serving. Moreover, we also replace the attention kernel of MuxServe with more efficient FlashAttention, which is the same as WEAVER.

Testbeds. We used two test platforms with different interconnect bandwidths: (i) 4× A100-40GB with NVLink; (ii) a cloud server containing 4× L40S with PCIe as the interconnect. Following popular disaggregation serving design [34], each model (hot and cold) uses the 1p1d setup. We only multiplex the workload between decoding GPUs. On the L40S

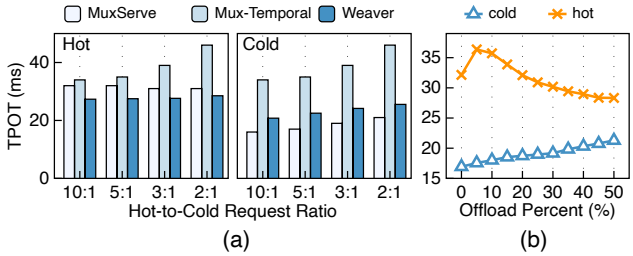


Figure 5: Sensitivity Analysis (I). *(a): Varying hot to cold request rate. (b): Varying the offload sequence percentage.*

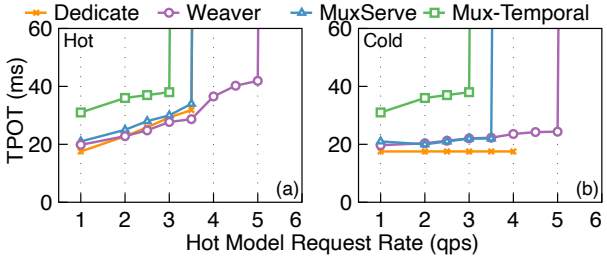


Figure 6: Sensitivity Analysis (II). *Results with longer output.*

testbed, we did not have root access, preventing NVIDIA MPS from being enabled. Therefore, we evaluated MuxServe using only the temporal multiplexing mode on this platform.

Metrics. Since in the evaluation WEAVER and baselines perform workload multiplexing only in the decode instance, the TTFT is out of the optimization scope of these systems (we present it in §5.3). Thus, we mainly report the TPOT (Time Per Output Token) of different models, which is crucial for the user experience.

5.1 Overall Performance

We evaluate the overall performance of WEAVER against baselines by varying the request rate of the hot model, ranging from low loads up to its maximum sustainable QPS (its saturation point), and respectively reporting the mean TPOT of the hot and cold model. Figure 4 shows the results.

Hot model (Figure 4a-d). WEAVER primarily aims to improve hot model performance. Compared to dedicated serving, WEAVER increases maximum throughput by up to 60%. This is because WEAVER utilizes available memory on GPUs of cold models, enabling larger batch size and thus higher throughput. At low request rates (less than 5 QPS), WEAVER shows slightly higher TPOT due to communication overhead. However, at higher request rates, WEAVER reduces the TPOT by up to 39%. When compared to multiplexing serving approaches, WEAVER achieves up to 22% higher maximum throughput for hot models on A100 GPUs. The improvement is more significant on L40S GPUs with lower interconnect bandwidth, where WEAVER achieves up to 77% higher maximum throughput, as tensor parallelism in multiplexing approaches introduces larger overhead on such platforms.

Cold model (Figure 4e-h). While improving the hot model's throughput, WEAVER has an acceptable impact on the cold model's performance. When load of the hot model is low, WEAVER achieves a 1.04× higher TPOT for the cold model compared to dedicated serving. Under the high load of the hot model, where more workload is offloaded, WEAVER incurs modest overhead and the TPOT increases 3-5ms on average. MuxServe shows slightly lower TPOT since it uses tensor parallelism with 2 GPUs for the cold model during inference, which comes at the cost of lower hot model throughput. Moreover, since each offloaded attention operator is small, it does not significantly interfere with the cold model's execution. As a result, the P99 TPOT of the cold model is not influenced: for example, 19-22ms on the A100 platform, which is 1-2 ms higher than the mean case.

Batch size and memory utilization We analyze the advantages of WEAVER in terms of memory utilization, using serving Azure-conv trace on A100 as an example. Under dedicated serving, at its maximum throughput of 15 QPS, the hot model achieves an average batch size of 118. This nearly saturates the KV cache memory of the hot GPU, reaching 88.9% utilization. In contrast, at the same QPS of 15, WEAVER significantly alleviates memory pressure on the hot GPU, reducing its KV cache utilization to 46.9%. This is achieved by actively leveraging the KV cache memory of the cold GPU, which exhibits 36.7% utilization. Furthermore, when the QPS is increased to 22, the maximum throughput supported by WEAVER, the average batch size of the hot model rises to 231. This effectively doubles the batching capacity compared to the dedicated serving approach.

5.2 Sensitivity Analysis

We analyze the sensitivity of WEAVER under two different configurations on the A100 platform with the BurstGPT trace.

Hot-to-cold request ratio. We evaluate the WEAVER's benefits with varying hot-to-cold request ratios, the request rate of the hot model is fixed at 10. As shown in Figure 5a, across all kinds of hot-to-cold request ratios, WEAVER shows consistently 15% lower TPOT than MuxServe and 20%-40% lower than Mux-Temporal for the hot model. For the cold model, MuxServe's TPOT is slightly lower than WEAVER, because it uses model parallesim. The Mux-Temporal suffers more from inefficient interleaved execution caused by temporal sharing, and the TPOT of WEAVER can be 56% lower than it.

Offloaded sequence ratio. We evaluate WEAVER's performance by varying the percentage of offloaded attention operators, with the hot model's request rate fixed at 15. Figure 5b shows the results. The cold model's TPOT increases almost linearly with the offload percentage, which is expected due to the proportionally increased workload. At low offload ratios (5%-10%), the hot model's performance slightly degrades due to communication overhead. When the offload ratio increases to 50%, the hot model's TPOT decreases by 13% while adding a 3.7ms TPOT overhead to the receiver's processing time. Our experiments show that in our workloads, a 45% offload ratio achieves optimal hot model's TPOT by

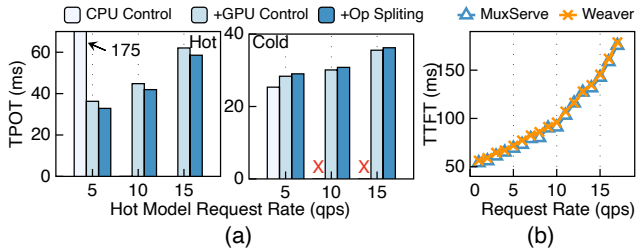


Figure 7: Ablation Study. *(a): Ablation of different techniques in* WEAVER. *(b): Ablation of different serving frameworks.*

providing a small grace period (i.e., $K-N$ in Algorithm 1) for possible blocking caused by a single small kernel.

Longer output length. We further evaluate WEAVER's performance on synthetic data featuring longer output sequences. The requests are synthesized with 512 placeholder tokens as the input and enforce the LLM to generate exactly 1024 tokens by setting max output length to 1024 and enabling ignore_eos. Experiments were conducted on the A100 testbed, with the cold model's QPS fixed at 1. Figure 6 presents the results. WEAVER achieves up to 42% higher max throughput compared to MuxServe. This improvement is more pronounced than observed with the Azure and BurstGPT datasets, primarily because WEAVER's attention offloading facilitates effective workload interleaving. With extended output lengths, the batch size is limited by the KV cache size. By offloading attention computation, which constitutes a larger portion of execution time, to the cold model instance, WEAVER increases the maximum possible overall batch size, and thus enables the hot model to achieve higher throughput with the same resources. Meanwhile, since the offloaded workload is heavier, the cold model's TPOT increases by up to 8ms than dedicated serving.

5.3 Ablation Study

Ablation of different techniques in WEAVER. We evaluate WEAVER's individual optimizations using the Azure-Conv dataset on L40S GPUs. Figure 7a shows our results. In the baseline (i.e., *CPU Control* in the figure), the CPU performs the control flow logic, which leads to significant blocking from pre-issued kernels. This blocking delays the offloaded attention computation, resulting in a high TPOT of 175ms for the hot model. After implementing GPU-driven dynamic control flow (i.e., *+GPU Control*), we observe a 4.83× reduction in the sender's TPOT, with only a minor 1.8ms increase in TPOT of the cold model due to the additional polling kernel. Adding operator splitting (i.e., *+Op Spliting*) further reduces the hot model's TPOT by up to 9.5% while increasing the cold model's TPOT by just 0.7ms. Together, these optimizations add only 2.5ms overhead to the cold model's original TPOT, representing less than a 10% increase — a reasonable trade-off given the performance improvements for the hot model.

Ablation of different serving frameworks. We compare the TTFT of MuxServe and WEAVER on the A100 platform, running the BurstGPT trace at various request rates. Figure 7b shows our results. The TTFT between WEAVER and MuxServe remains consistently close. The TTFT of WEAVER is slightly higher (at most 5.7%) due to more complex Python logic in our vLLM version. The results demonstrate that the performance comparison between the two frameworks is fair and the TTFT is not influenced.

6 Related Work

Multi-model serving. In the era of DNNs, numerous approaches have been proposed to support multi-model serving, such as TGS [33], PipeSearch [12], AlpaServe [22], MPS [7], and MIG [6], among others. They focus on achieving efficient model switching and spatio-temporal multiplexing with minimal overhead. These approaches mostly treat the model as a black box and omit the opportunity to take advantage of the model structure to serve multiple models.

For multi-LLM serving, several studies have explored scalable and efficient solutions. Llumnix [27] migrates KV cache to achieve load balancing across the LLM-serving system. The most closely related to ours is MuxServe [17], which employs tensor parallelism and leverages spatial-temporal multiplexing to concurrently serve multiple LLM requests.

Attention offloading. Prior work [13, 19, 20, 23, 24] has explored offloading attention operators in LLMs to different computing devices, including CPUs, GPUs, and PIM. LoongServe [32] speeds up inference of long sequences by splitting attention operators across multiple servers. Our work differs from these approaches: instead of using dedicated hardware, we offload attention to GPUs that are *running* cold models. This unique approach requires careful design to mitigate the blocking caused by the ongoing cold model serving.

7 Conclusion

This paper presents WEAVER, a multi-LLM serving system that introduces workload weaving to improve GPU memory utilization with low communication overhead. Our experimental results demonstrate its performance gain.

Acknowledgements

We sincerely thank our shepherd Muhammad Bilal and anonymous reviewers for their feedback and suggestions. This work is supported by the National Natural Science Foundation of China (Grant No. U22B2023, 62472242, 62332011), Beijing Natural Science Foundation (Grant No. L242016) and the Young Elite Scientists Sponsorship Program by CAST (Grant No. 2023QNRC001).

References

[1] ChatGPT. https://chatgpt.com/, 2024. [Website].

[2] CUDA Interprocess Communication. https://docs.nvidia.com/cuda/cuda-c-programming-guide/#interprocess-communication, 2024. [Website].

[3] Gemini API versions explained. https://ai.google.dev/gemini-api/docs/api-versions, 2024. [Website].

[4] hipipcgetmemhandle interface reference. https://rocm.docs.amd.com/projects/hipfort/en/docs-5.4.1/doxygen/html/interfacehipfort_1_1hipipcgetmemhandle.html, 2024. [Website].

[5] Huggingface Serverless Inference API. https://huggingface.co/docs/api-inference/index, 2024. [Website].

[6] NVIDIA Multi-Instance GPU. https://www.nvidia.com/en-sg/technologies/multi-instance-gpu/, 2024. [Website].

[7] Nvidia Multi-Process Service. https://docs.nvidia.com/deploy/mps/index.html, 2024. [Website].

[8] OpenAI API reference. https://platform.openai.com/docs/api-reference/chat, 2024. [Website].

[9] Openrouter: A unified interface for llms. https://openrouter.ai/, 2024. [Website].

[10] Together AI – Dedicated models. https://docs.together.ai/docs/dedicated-models, 2024. [Website].

[11] Together AI – The AI Acceleration Cloud. https://www.together.ai/, 2024. [Website].

[12] Zhihao Bai, Zhen Zhang, Yibo Zhu, and Xin Jin. PipeSwitch: Fast Pipelined Context Switching for Deep Learning Applications. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 499–514. USENIX Association, November 2020.

[13] Shaoyuan Chen, Yutong Lin, Mingxing Zhang, and Yongwei Wu. Efficient and Economic Large Language Model Inference with Attention Offloading, 2024.

[14] Rongxin Cheng, Yuxin Lai, Xingda Wei, Rong Chen, and Haibo Chen. Kunserve: Efficient parameter-centric memory management for llm serving, 2025.

[15] Tri Dao. FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning. In *International Conference on Learning Representations (ICLR)*, 2024.

[16] DeepSeek-AI. Deepseek-v3 technical report, 2024.

[17] Jiangfei Duan, Runyu Lu, Haojie Duanmu, Xiuhong Li, Xingcheng ZHANG, Dahua Lin, Ion Stoica, and Hao Zhang. MuxServe: Flexible Spatial-Temporal Multiplexing for Multiple LLM Serving. In *Forty-first International Conference on Machine Learning*, 2024.

[18] Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, and et al. The llama 3 herd of models, 2024.

[19] Jiaao He and Jidong Zhai. FastDecode: High-Throughput GPU-Efficient LLM Serving using Heterogeneous Pipelines, 2024.

[20] Xuanlin Jiang, Yang Zhou, Shiyi Cao, Ion Stoica, and Minlan Yu. NEO: Saving GPU Memory Crisis with CPU Offloading for Online LLM Inference, 2024.

[21] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 611–626, New York, NY, USA, 2023. Association for Computing Machinery.

[22] Zhuohan Li, Lianmin Zheng, Yinmin Zhong, Vincent Liu, Ying Sheng, Xin Jin, Yanping Huang, Zhifeng Chen, Hao Zhang, Joseph E Gonzalez, et al. {AlpaServe}: Statistical multiplexing with model parallelism for deep learning serving. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 663–679, 2023.

[23] Xiurui Pan, Endian Li, Qiao Li, Shengwen Liang, Yizhou Shan, Ke Zhou, Yingwei Luo, Xiaolin Wang, and Jie Zhang. InstInfer: In-Storage Attention Offloading for Cost-Effective Long-Context LLM Inference, 2024.

[24] Jaehyun Park, Jaewan Choi, Kwanhee Kyung, Michael Jaemin Kim, Yongsuk Kwon, Nam Sung Kim, and Jung Ho Ahn. AttAcc! Unleashing the Power of PIM for Batched Transformer-based Generative Model Inference. ASPLOS '24, page 103–119, New York, NY, USA, 2024. Association for Computing Machinery.

[25] Pratyush Patel, Esha Choukse, Chaojie Zhang, Aashaka Shah, Íñigo Goiri, Saeed Maleki, and Ricardo Bianchini. Splitwise: Efficient generative llm inference using phase splitting, 2024.

[26] Jiwu Shu. *Data Storage Architectures and Technologies*. Springer Press, 2024.

[27] Biao Sun, Ziming Huang, Hanyu Zhao, Wencong Xiao, Xinyi Zhang, Yong Li, and Wei Lin. Llumnix: Dynamic Scheduling for Large Language Model Serving, 2024.

[28] Hideaki Takagi. Queuing analysis of polling models. *ACM Comput. Surv.*, 20(1):5–28, March 1988.

[29] Gemini Team. Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context, 2024.

[30] Qwen Team. Qwen2.5 technical report, 2025.

[31] Yuxin Wang, Yuhan Chen, Zeyu Li, Xueze Kang, Zhenheng Tang, Xin He, Rui Guo, Xin Wang, Qiang Wang, Amelie Chi Zhou, and Xiaowen Chu. BurstGPT: A Real-world Workload Dataset to Optimize LLM Serving Systems, 2024.

[32] Bingyang Wu, Shengyu Liu, Yinmin Zhong, Peng Sun, Xuanzhe Liu, and Xin Jin. LoongServe: Efficiently Serving Long-Context Large Language Models with Elastic Sequence Parallelism. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, SOSP '24, page 640–654, New York, NY, USA, 2024. Association for Computing Machinery.

[33] Bingyang Wu, Zili Zhang, Zhihao Bai, Xuanzhe Liu, and Xin Jin. Transparent GPU Sharing in Container Clouds for Deep Learning Workloads. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 69–85, Boston, MA, April 2023. USENIX Association.

[34] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. DistServe: Disaggregating Prefill and Decoding for Goodput-optimized Large Language Model Serving. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 193–210, Santa Clara, CA, July 2024. USENIX Association.

Torpor: GPU-Enabled Serverless Computing for Low-Latency, Resource-Efficient Inference

Minchen Yu[†‡] Ao Wang[§] Dong Chen[‡] Haoxuan Yu[‡] Xiaonan Luo[‡] Zhuohao Li[‡]
Wei Wang[‡] Ruichuan Chen[*] Dapeng Nie[§] Haoran Yang[§] Yu Ding[§]
[†]*CUHK-Shenzhen* [‡]*HKUST* [§]*Alibaba Group* [*]*Nokia Bell Labs*

Abstract

Serverless computing offers a compelling cloud model for online inference services. However, existing serverless platforms lack efficient support for GPUs, hindering their ability to deliver high-performance inference. In this paper, we present `Torpor`, a serverless platform for GPU-efficient, low-latency inference. To enable efficient sharing of a node's GPUs among numerous inference functions, `Torpor` maintains models in main memory and dynamically swaps them onto GPUs upon request arrivals (i.e., late binding with model swapping). `Torpor` uses various techniques, including asynchronous API redirection, GPU runtime sharing, pipelined model execution, and efficient GPU memory management, to minimize latency overhead caused by model swapping. Additionally, we design an interference-aware request scheduling algorithm that utilizes high-speed GPU interconnects to meet latency service-level objectives (SLOs) for individual inference functions. We have implemented `Torpor` and evaluated its performance in a production environment. Utilizing late binding and model swapping, `Torpor` can concurrently serve hundreds of inference functions on a worker node with 4 GPUs, while achieving latency performance comparable to native execution, where each model is cached exclusively on a GPU. Pilot deployment in a leading commercial serverless cloud shows that `Torpor` reduces the GPU provisioning cost by 70% and 65% for users and the platform, respectively.

1 Introduction

The remarkable advances in machine learning (ML) and its widespread adoption in various domains have fueled a surging demand for cloud-based ML inference services [24, 32, 49, 65, 66]. Severless computing offers a compelling cloud model for inference serving [20, 58, 61]. In a serverless cloud, users publish ML models as inference functions, and delegate resource provisioning and scaling responsibilities to the cloud platform. Serverless computing is also economically appealing as users only pay for the resources consumed by their functions (i.e., pay-per-use billing), eliminating resource idling costs.

However, today's serverless computing platforms, such as AWS Lambda [6] and Alibaba Function Compute [1], lack efficient support for GPUs. They typically run an ML model in a container (or a microVM) and early bind it to a GPU before starting to serve requests. To avoid considerable startup overhead of on-demand GPU function provisioning (e.g., tens of seconds as shown in Table 1), an inference function is maintained as a long-lived, provisioned instance on a designated GPU to handle future requests [4, 7]. This approach essentially follows the "serverful" inference serving practice [47, 49, 65], requiring users to pay for the occupied GPUs even during function idling. Furthermore, our analysis in a production cloud demonstrates that inference functions exhibit varying request rates, with 85% functions being invoked no more than once per minute (Fig. 2). Early binding these functions to GPUs results in low utilization and imbalanced load across GPUs, making it inefficient for cloud providers.

We believe that an efficient serverless inference platform should provide four desirable properties. First, it should enable *pay-per-GPU-use billing* for users, with charges incurred only when the functions are invoked and running on GPUs. Second, the platform should achieve optimal GPU utilization through efficient *GPU sharing* for concurrent inference functions, minimizing resource provisioning costs for cloud providers. Third, the platform should be aware of the user-specified *latency SLOs* and strive to meet them for all inference requests, if feasible. Lastly, the platform should achieve the aforementioned three properties *without requiring detailed knowledge about inference models* due to intellectual property and business-critical confidentiality reasons. We notice that there have been several relevant systems developed in recent years [23, 32, 34, 36, 46, 47, 49, 58, 63], none of which, however, provide all of these properties for serverless inference. They often suffer from cost inefficiency, SLO violations, or necessitate model-specific knowledge (see §2.2 and §9).

In this paper, we present `Torpor`, a GPU-efficient serverless inference platform that achieves all four desirable properties and is readily-deployable onto real-world serverless platforms without intrusive changes. `Torpor` follows a late binding de-

sign principle, whereby idle inference models are maintained in host memory and dynamically swapped to GPUs upon request arrivals. Compared to GPU memory, host memory is less expensive and has a much larger capacity, making it an ideal storage for holding numerous idle functions. This approach naturally supports pay-per-GPU-use billing, as idle functions no longer occupy GPU resources. Furthermore, by dynamically swapping models from host to GPUs, it enables fine-grained GPU sharing among concurrent inference functions, substantially improving GPU utilization and load balancing across GPUs. Model swapping can also be efficiently performed through pipelined loading, yielding significantly lower latency compared to function cold starts. All these are achieved without detailed knowledge about inference models – a must-have in a commercial environment for intellectual property and confidentiality protection. These techniques, combined with intelligent request scheduling, enable the platform to optimize the SLO attainment for users.

Specifically, to realize late binding while being readily-deployable on real-world serverless platforms, `Torpor` leverages a GPU pooling architecture. In this design, each worker node manages a pool of local GPUs and allows its inference functions to access any of these GPUs freely through CUDA API redirection. This enables seamless model swapping within a GPU pool and is transparent to users. However, this approach also presents three key challenges.

First, GPU pooling and model swapping incur high communication overhead compared to native execution (i.e., executing a model directly on a GPU). To address this challenge, `Torpor` proposes *asynchronous API redirection* to avoid frequent synchronizations between the inference functions and the GPU pool, eliminating the high communication overheads for model inference. `Torpor` further utilizes *pipeline execution* to overlap the host-to-GPU model swapping and the inference execution, thereby reducing end-to-end latency. It also utilizes high-speed NVLink for fast model swapping between GPUs whenever feasible and beneficial. Combined with low-latency API redirection, `Torpor` can efficiently execute models on any available GPUs. `Torpor` is intentionally designed to be model-agnostic to meet the confidentiality requirements while being generally applicable to various models, including even large generative models where runtime states (e.g., KV cache) can be managed as part of the model.

The second challenge is that GPU pooling and model swapping necessitate an efficient GPU memory management system. `Torpor` designs such a system that automatically tracks the addresses of models as they are swapped across multiple GPUs and adjusts each memory access of CUDA APIs accordingly during inference execution. It also efficiently organizes and shares memory blocks to avoid high memory allocation overheads, improving the overall performance of model swapping. Additionally, `Torpor` offers two GPU runtime management modes—runtime sharing and runtime isolation—to meet various needs for resource efficiency and cross-model isolation.

The third challenge is that the platform should meet the latency SLOs for inference functions while maintaining low GPU costs. `Torpor` proposes three policies to achieve this objective. First, `Torpor` designs a request scheduling algorithm that minimizes model swapping overheads, resulting in reduced end-to-end inference latency. It categorizes models into two groups, heavy or light, based on whether these models incur high overhead during swapping via PCIe. `Torpor` then prioritizes NVLink over PCIe for transferring heavy models across GPUs, effectively reducing concurrent PCIe traffic. Second, `Torpor` globally manages GPU memory in the pool and leverages model heaviness to guide eviction. Together with request scheduling, this approach significantly minimizes model swapping overhead. Third, `Torpor` proposes an SLO-aware request queuing policy that prioritizes requests to functions that have a higher likelihood of meeting SLOs, effectively improving the SLO attainment.

We have implemented and evaluated `Torpor` through a pilot deployment in Alibaba Cloud [1], one of the world's largest commercial serverless platforms. Evaluation results show that `Torpor` achieves low-latency model inference, comparable with native executions. `Torpor` can share a single GPU across hundreds of inference functions and load-balance GPUs with model swapping, resulting in over 10× cost reduction compared with current GPU offering in Alibaba Cloud. With its efficient SLO-aware scheduling and queuing policies, `Torpor` can serve 480 functions on a 4-GPU worker node while achieving low tail latency and satisfying millisecond-scale SLOs for all functions. Cluster experiments further demonstrate that `Torpor` scales well with the number of inference functions at low resource cost and meets per-function latency SLOs for thousands of functions. `Torpor` has been beta-released in a pilot production cluster in Alibaba Cloud, saving 70% of user costs on average and 65% of GPU provisioning costs for Alibaba Cloud.

2 Background and Motivation

In this section, we first give an overview of serverless inference. We then describe the inefficiency of existing solutions to enabling GPUs in serverless platforms, and highlight four key requirements in this regard.

2.1 Serverless Inference

As a leading serverless platform with a global presence, our Alibaba Cloud has observed a growing adoption among enterprise customers who opt to deploy their inference services using serverless functions, known as *serverless inference*. In comparison to existing inference services based on a "serverful" cloud model, such as AWS SageMaker [5], serverless

[1] We have open-sourced `Torpor`'s single-node prototype at https://github.com/FCSLab/torpor.

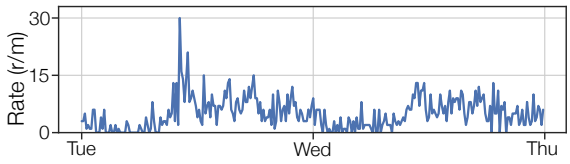


Figure 1: A two-day request trace of a typical GPU inference function in Alibaba Cloud.

inference significantly alleviates the burden of server management for cloud users. Specifically, the serverful approach requires users to manually configure various system-level parameters (e.g., VM types, GPUs, CPU cores, etc.) and manage resource provisioning (e.g., scaling the number of VMs up or down according to demand changes). In contrast, serverless inference enables users to simply publish models with inference code as functions, and then cloud providers automatically handle resource provisioning, autoscaling, scheduling, and fault tolerance. Furthermore, compared with the serverful approach, serverless inference also offers substantial cost savings as users do not pay for idle resources under the pay-per-use pricing model [20, 58, 61, 65]. In Alibaba Cloud, the requests to a function typically exhibit dynamic, bursty arrival patterns as shown in Fig. 1, consistent with previous research findings [24, 25, 32, 33, 41, 42, 47, 49, 67]. By leveraging the high elasticity of a serverless platform, inference functions can quickly scale in response to the changing workload, while users are billed based on the actual function runtime at a fine granularity, such as 1 ms [6, 8].

2.2 GPU Support in Serverless Platforms

Despite the benefits of the serverless inference model, existing serverless platforms, including Alibaba Cloud and other leading platforms, currently lack efficient support for GPUs, which impedes their ability to achieve high-performance serverless inference. Alibaba Cloud users also have expressed a compelling need to execute their models in GPU-enabled functions.

Existing solutions and their inefficiency. A number of recent systems have been proposed to support GPUs in serverless platforms [1, 27, 29, 58]. They, however, still follow the approach of existing serverful model serving systems (e.g., Nexus [49] and INFaaS [47]), and deploy inference models as long-running containers where each container, when created, is bound to a specific GPU (i.e., early binding). The deployed model remains in the memory of a designated GPU to handle future requests, and the occupied GPU resources can only be reclaimed after the model serving terminates.

However, the early-binding approach deviates from the serverless paradigm and is costly for both cloud users and providers. First, binding inference functions to GPUs occupies resources for extended duration, even when idling. Thus, users are obligated to pay for the allocated GPUs regardless of actual usage [3], leading to high expenses that undermine

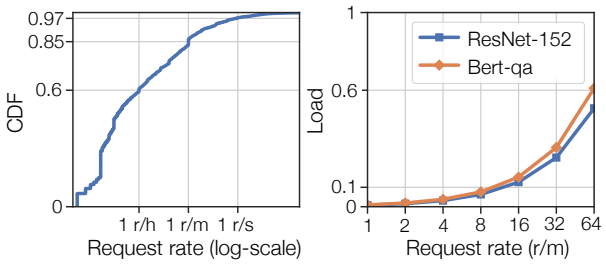


Figure 2: CDF of average function request rates from a one-week production trace (left) and the GPU load under various per-function request rates when running multiple functions on a `V100` GPU to saturate its 32 GB memory (right).

Table 1: Model startup times (s) under `Torpor` (runtime isolation mode in §4.5) and cold-starts. `Torpor`'s startup time is broken down into model loading and runtime resumption.

Model	Torpor		Cold-start	Mem. footprint
	Model	Runtime		
ResNet-152 [17]	0.03	0.26	8	1.6 GB
Bert-qa [26]	0.14	0.19	11	2.4 GB
Stable Diffusion [18]	0.24	1.5	25	5.1 GB
Llama3-8B [11]	1.6	1.4	48	13 GB
Qwen-14B [16]	2.1	1.5	57	20.1 GB
Llama2-13B [10]	2.5	1.9	61	24.5 GB

the cost-saving benefits of serverless inference. Second, this approach results in severe GPU underutilization, considering the low average request rates of most inference functions and the cross-GPU load imbalancing. Fig. 2 (left) depicts the distribution of the average request rates of Alibaba Cloud functions in a one-week trace, revealing that 85% (97%) of functions were invoked only once per minute (second) on average[2]. These findings align with the observations from other production traces [8, 48]. Fig. 2 (right) further illustrates that consolidating multiple models to fill GPU memory can still lead to low GPU load. Meanwhile, packing models into a GPU can cause temporary overloading due to the bursty request patterns (Fig. 1), thus inevitably leading to hotspots and load imbalancing in a multi-GPU setting. The impact of load imbalancing will be shown in Fig. 9 in §7.2.

To reduce costs, current systems need to frequently reclaim GPU resources when functions are inactive, avoiding charges for unused GPUs and allowing other functions to utilize idle resources. Unfortunately, this approach leads to frequent function cold starts, leading to significant overhead for model inference. Table 1 shows model startup times under cold starts, which need tens of seconds for GPU container setup, ML framework startup, GPU runtime creation, and model initialization[3]. Therefore, the cold-start overhead far exceeds the

[2]For confidentiality reasons, we depict the request rates of both CPU and GPU functions, which exhibit similar patterns (see Fig. 1).

[3]We exclude the delay of fetching a remote container image or a model file for cold starts, which can take extra seconds to minutes to complete [54]. A detailed discussion of `Torpor`'s performance is provided in §8.

Table 2: A comparison of Torpor and existing solutions that offer GPU support on serverless platforms.

Solution	GPU pay-per-use	GPU efficient	SLO compliant	Model agnostic
Alibaba Cloud [1]	×	×	×	✓
Molecule [27]	×	×	×	✓
DGSF [29]	×	×	×	✓
INFless [58]	×	×	*	×
Torpor	✓	✓	✓	✓

typical SLO requirement of model inference.

Requirements of serverless inference. Table 2 summarizes key requirements of serverless inference and compares Torpor with other existing solutions. Serverless users should be billed only when their functions are invoked and running on GPUs to achieve substantial cost savings (***pay-per-GPU-use***)[4]. Serverless platforms like Alibaba Cloud should serve as many inference functions as possible using a minimum number of GPUs, thereby attaining high GPU utilization (***GPU efficient***). The platform should allow users to specify their latency SLOs and strive to meet the latency SLOs for all functions (***SLO compliant***). For confidentiality reasons, the serverless platform should avoid inspecting detailed model structure, which can be of high business value (***Model agnostic***).

Compared with Torpor, none of existing solutions can meet all desired requirements. Alibaba Cloud and Alibaba Function Compute [1] are leading commercial serverless platforms with GPU supports; Molecule [27] introduces a serverless platform that supports GPUs and other hardware devices; DGSF [29] enables serverless functions to access GPUs in a remote cluster. These systems employ the early-binding approach as previously discussed, failing to enable pay-per-GPU-use billing and achieve high GPU efficiency. Moreover, they are oblivious to the semantics of model inference and unable to meet latency SLOs. INFless [58] presents a serverless inference system that early-binds functions to GPUs. While INFless proposes function scheduling and keep-alive schemes aimed at low-latency inference, it still leads to function cold starts and SLO violations (details in §7.3). Furthermore, INFless requires model knowledge for operator-level profiling. We leave more discussions on related work to §9.

3 Key Insight and Challenges

Key insight. As described in §2.2, the current early-binding approach of retaining inference models in GPU memory leads to high idling costs and underutilized resources. Therefore, an efficient serverless inference platform should enable *late binding*, where GPUs are managed as a resource pool and idle inference models reside in host memory, dynamically swapping into any available GPUs upon request. This approach should also be *easily deployable* on real-world serverless platforms without requiring intrusive changes. Late binding offers several key advantages in meeting the requirements in §2.2. **First**, keeping models in host memory eliminates GPU memory usage during idle periods, enabling pay-per-GPU-use billing and cost savings for cloud users. **Second**, host memory is significantly larger than GPU memory (e.g., a few TB vs. tens of GB), allowing for consolidation of multiple low-frequency functions onto a single GPU with improved GPU utilization. Late binding also facilitates load-balancing across multiple GPUs in a pool. **Third**, model swapping provides an efficient method to resume function execution compared to cold starts in the early-binding approach, thereby facilitating SLO compliance. **Finally**, late binding can be performed transparently to users within the GPU pool, which holds a holistic view of memory usage without requiring detailed model-specific knowledge.

Challenges. Implementing GPU pooling and late binding in the serverless platform presents three challenges. ***C1: Efficient GPU pooling and model swapping***. GPU pooling requires inference functions to synchronize with a remote GPU pool [28, 31], which introduces additional communication overhead compared to local executions and presents challenges in achieving low-latency inference. ***C2: GPU memory management.*** To enable seamless late binding, the platform should automatically monitor and manage memory usage without detailed model knowledge. This requires a unified and efficient GPU memory management system across the GPU pool. ***C3: SLO compliance and resource efficiency***. The platform should provide efficient request scheduling and model placement algorithms that effectively utilize the late binding mechanism to meet latency SLOs and enhance resource efficiency.

In the following sections, we present Torpor, a GPU-enabled serverless platform that addresses the aforementioned challenges and, importantly, is readily-deployable onto real-world serverless platforms without intrusive changes.

4 Torpor System Design

4.1 Architecture overview

Fig. 3 provides an overview of the architecture of Torpor, which comprises two main components: the cluster manager and worker nodes. The cluster manager handles cluster-level tasks, including request routing, node allocation, and resource scaling. It dynamically schedules function instances and routes inference requests to maintain load balancing across worker nodes and ensure fault tolerance (§ 4.5). At each worker node, Torpor employs GPU pooling, where a GPU server manages all local GPUs as a pool and allows functions to dynamically access any available GPUs. Within the GPU

[4] In our experiences, enterprise customers are willing to pay a nominal fee to retain idle functions in host memory for substantially improved performance (§8), similar to the function keep-alive charge meant to avoid cold starts [3, 7, 37].

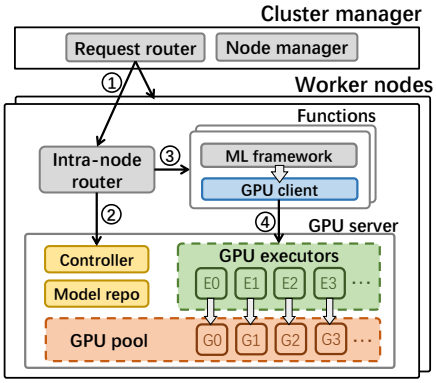


Figure 3: Architecture overview of Torpor. A request arriving at a Torpor cluster is first routed to a worker node hosting its target function ①. The router in the worker node synchronizes with the GPU server to query the executor for this request ②, and then routes it to the function instance with the target executor ID ③. The function instance next processes the request and uses a GPU client to automatically redirect CUDA API calls to this executor ④, and finally returns the result to the user after request completion.

server, a model repository manages models in host memory; GPU executors handle CUDA execution, perform necessary model swapping, and manage GPU memory on the associated GPUs; the controller maintains a global view of GPU memory and executor status, and decides how to schedule requests to executors. Additionally, each worker node runs an intra-node router to signal the GPU server about request arrivals and route requests to local inference functions. Once the target function receives a request, it interacts with the scheduled executor through a GPU client by remoting CUDA API calls. All components within the worker node—the GPU server, intra-node router, and functions—are deployed as containers.

Key to Torpor is to develop an efficient GPU server that enables low-latency model inference and addresses the challenges discussed in §3. Specifically, we will elaborate upon Torpor's designs to address the following questions: 1) how Torpor achieves low-latency GPU pooling (§4.2) and model swapping (§4.3); 2) how Torpor tracks the memory footprint of functions and manages GPU memory (§4.4); 3) how Torpor ensures isolation and handles failures (§4.5).

4.2 GPU Remoting

Asynchronous API redirection. Existing GPU remoting solutions [28, 45] introduce high communication overhead due to synchronizations for individual API calls during model inference (details in §7.1). Leveraging the computing pattern of model inference, Torpor proposes *asynchronous* API redirection to reduce synchronizations. Specifically, we observe that the intermediate steps in an inference execution are typically performed asynchronously on the GPU, where intermediate data is generated and consumed in GPU memory without requiring data transfer to the host until the execution completes. Consequently, a function can redirect intermediate CUDA calls to the GPU executor asynchronously without waiting for their results, and only perform synchronizations for the final output. This approach preserves the execution order of CUDA APIs, ensuring the correctness of the inference results.

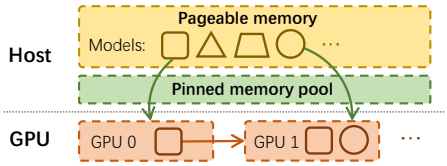


Figure 4: An example of model swapping. Models can be swapped from host to GPU through PCIe (green arrows), or across GPUs through NVLink (red arrow).

Following this insight, Torpor categorizes CUDA APIs into two groups based on their semantics: 1) *synchronous, blocking* APIs that require the host to await their completion before proceeding, such as cudaMalloc; and 2) *asynchronous, non-blocking* APIs that do not alter the host's runtime state, such as cudaLaunchKernel, which allows for asynchronous API redirection. Most APIs issued during inference fall into the asynchronous category, presenting opportunities to mitigate the synchronization overhead. Torpor can further batch consecutive CUDA API calls to enhance asynchronous API redirection (see APIs and batching details in our technical report [62]).

4.3 Model Swapping

As serverless platforms are constrained from examining the detailed model structures, it poses challenges to achieve seamless and efficient model swapping. Torpor overcomes the problem by leveraging two insights: 1) tracking general memory footprint of model inferences is feasible within a GPU pool, and 2) the memory access pattern of a model—the addresses and access order of model parameters—generally remains consistent across requests. Therefore, Torpor only tracks the first function execution (i.e., cold start), and applies the pattern to future request executions (see memory tracking in §4.4). Torpor performs model swapping on demand at the request level, and enhances performance through pinned memory pool and pipeline execution.

Model swapping and pipeline execution. Fig. 4 illustrates the model swapping in Torpor, where it utilizes pinned memory to enhance the host-to-GPU model loading and supports fast, cross-GPU model swapping via high-speed NVLink. Torpor's swapping can be generally applied to various types of models, including those that require maintaining runtime states (e.g., KV cache in LLMs) by treating these states as part of the model itself. Torpor further employs pipeline execution to enhance swapping performance, which is particularly effective for models computed in one forward pass, allowing the transmission of subsequent layers to overlap with the

computation of previous layers.

Key to pipeline execution is to judiciously group model parameters for swapping. Grouping too few parameters triggers a large number of transmissions and high synchronization overhead; conversely, grouping too many parameters impairs pipeline efficiency. Pipeline strategies in previous systems like PipeSwitch [22] and DeepPlan [36] do not suit serverless inference as they assume model structure is provided and require extensive model profiling. `Torpor` employs a model-agnostic approach to determine the group size for model pipelining. We observe that the transmission performance experiences an "elbow point" concerning group sizes: increasing the group size improves overall transmission throughput, but the improvement becomes marginal after a certain point. We therefore select this elbow point as the group size, which can achieve good swapping performance without substantially compromising pipeline efficiency. This group size depends only on hardware configurations such as PCIe, and can be easily determined by profiling various-sized data (e.g, about 2 MB in our testbed). This approach requires no detailed model knowledge and can be directly applied to various models.

Model eviction. We observe that model unloading from GPUs to host can result in considerable overhead and can interfere with concurrent inference executions. Therefore, `Torpor` always maintains a copy of the model in the host, and only invalidates its GPU memory region during eviction.

4.4 Memory Management

`Torpor` presents two key requirements for GPU memory management, as compared with other systems [12, 23, 44]. First, late binding requires a pool of GPUs to share the same logical memory space. This is because the inference functions do not recognize backend GPUs, and consistently access models using identical memory addresses even across different GPUs. Second, model swapping triggers frequently GPU memory (de)allocation, which leads to substantial overhead when using native methods like `cudaMalloc` (see Fig. 13). We therefore design a GPU memory management system that can effectively hide memory address differences between various GPUs and provide low-latency memory (de)allocations.

Memory address management. We observe ML frameworks like PyTorch typically organize data into blocks, each containing multiple parameters. `Torpor` leverages such memory layout to perform memory mapping at the block level. In particular, `Torpor` monitors memory blocks for each function and maintains a mapping to their actual physical addresses after model swapping. Since the internal data layout within each block remains unchanged (e.g., parameter offsets), `Torpor` can easily obtain the physical address of a parameter using its associated block address and offset. This approach eliminates the need for extensive metadata maintenance for individual data pointers, enabling the efficient address translation with low management overhead.

Memory block allocation. To mitigate the high overhead of native GPU memory allocation, `Torpor` reserves all GPU memory at bootstrap and internally manages memory blocks. This provides a shim layer to service memory requests from functions, without needing the native method. Key to this approach is to avoid memory fragmentation, which can decrease available GPU memory and harm overall efficiency. `Torpor` effectively addresses this issue by extending the Buddy memory allocation scheme [40] and leveraging unique characteristics of inference. It consolidates memory blocks from the same models to minimize fragmentation and enables sharing of common-sized blocks across different models (see details in our technical report [62]).

4.5 Isolation and Fault Handling

Resource isolation and GPU runtime management. `Torpor` provides container-level isolation for CPU and memory resources[5], similar to existing serverless platforms [1, 52, 60]. For GPUs, `Torpor` executes only one function on a GPU at a time and isolates GPU memory regions across functions. This is achieved through `Torpor`'s GPU server, which has full control over CUDA API execution and GPU memory access.

`Torpor` offers two isolation modes for GPU runtime: 1) runtime sharing, which runs a single runtime on a GPU for multiple models, for instance, in a more trusted environment, and 2) runtime isolation, which maintains a dedicated runtime for each model and suits better for a more untrusted environment. By default, `Torpor` employs runtime sharing to improve resource efficiency. When stricter isolation is necessary, `Torpor` can switch to runtime isolation mode. Indeed, in our pilot deployment (§8), `Torpor` runs in the runtime isolation mode. The overhead incurred by runtime isolation is generally acceptable, e.g., hundreds of milliseconds to 1.5 seconds as shown in Table 1, which is still over one order of magnitude latency improvement compared with cold starts.

Fault handling. `Torpor` sustains various system component failures. In case of function failures, `Torpor` restarts them to resume the execution. For executor failures or GPU runtime errors, `Torpor` migrates the affected models to other working GPUs (executors) via swapping, and then restarts the failed ones. When runtime isolation is employed, `Torpor` can ensure that buggy function executions do not affect others, achieving stronger fault isolation. The GPU server also persists runtime states (e.g., models and metadata) in local storage to allow fast recovery from an entire failure of the GPU server.

At the cluster level, `Torpor` persists metadata of individual nodes in a database, which enables the cluster manager to retain these states and recover from failures, aligning with current practices in Alibaba Cloud. It also keeps periodic health checks with the router on each worker node, and handles node

[5] `Torpor` makes no assumption on function sandboxes and can also support microVMs [19, 53].

Table 3: Latency (ms) of model pipelining execution when concurrently swapping other models through PCIe. The diagonal values indicate the latencies without concurrent models.

Model	DenseNet-169	ResNet-152	Bert-qa
DenseNet-169	**27**	27 (+0%)	27 (+0%)
ResNet-152	31 (+7%)	**29**	43 (+48%)
Bert-qa	166 (+11%)	240 (+61%)	**149**

failures by launching a new node and migrating all relevant functions.

5 Torpor Policy Design

We present how Torpor meets the latency SLOs and delivers resource efficiency (i.e., Challenge *C3* in §3). We start with the design overview, followed by individual policies.

5.1 Design Overview

Objective. The objective of Torpor's policy design is to meet latency SLOs for inference functions while minimizing the resource cost. We define a function to comply with latency SLOs if its tail request latency is not longer than a user-specified deadline, and meter the resource cost by the number of worker nodes. Key to achieving this goal is to *maximize the number of SLO-compliant functions* at each worker, such that Torpor can efficiently exploit per-worker GPU resources to host as many functions as possible, which in turn reduces the total number of workers required.

Challenges. Previous systems have proposed various schemes to meet latency SLOs [32, 58, 65, 67]; however, their policies do not apply to Torpor for two reasons. First, previous systems like INFless [58] and Shepherd [67] assume sufficient GPU memory and employ early binding, so they schedule model serving instances to GPUs and then batch and route requests to them. In contrast, Torpor focuses on late-binding the often lower-frequency or varying-demand functions to a pool of memory-constrained GPUs, requiring a joint design of model management (i.e., model swapping and eviction) and request scheduling. Second, previous systems assume a stable model inference latency [32, 47, 58, 65, 67] which, however, does not hold in our setting — model swapping can cause unpredictable performance due to PCIe bandwidth contention [21, 38]. For instance, as show in Table 3, concurrently swapping two models through PCIe increases individual model inference latency compared with running them alone, especially for large models (e.g., Bert-qa).

We propose three policies to address the aforementioned challenges. First, considering that packing many functions together can cause short-term overloading and request queueing, Torpor introduces a request prioritization policy to maximize the number of SLO-compliant functions (§5.2). Second, Torpor designs a request scheduling and model swapping policy to reduce bandwidth contention across concurrent models, thereby improving overall inference performance (§5.3). Lastly, by leveraging the characteristics of model swapping, Torpor proposes an effective model eviction policy to reduce bandwidth footprint in model swapping; combined with the request scheduling policy, Torpor minimizes the interference among concurrent model executions (§5.4).

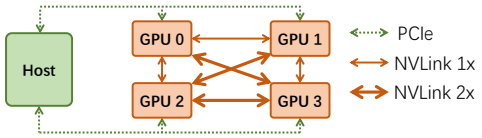


Figure 5: Topology of a 4-GPU worker node in Alibaba Cloud.

5.2 Request Queueing

To maximize the number of SLO-compliant functions, Torpor needs to monitor the SLO compliance of individual functions to determine their request executions. Intuitively, Torpor prioritizes functions with a higher probability to comply with SLOs. However, realizing this approach requires answering two questions: 1) how to quantify the likelihood of SLO compliance for a function, and 2) how to determine function execution order for improved SLO compliance.

Torpor proposes a metric, *required request count* (RRC), to measure the "degree of needed effort" to meet SLOs. RRC represents the expected request number that a function needs to successfully serve in order to satisfy SLOs. Let n be the current number of requests to a function, and m be the number of requests served within deadlines out of n. The RRC of this function is defined as $\frac{pn-m}{1-p}$, where p is the tail percentile specified in SLOs such as 98%. This is derived from the equation: $\frac{m+RRC}{n+RRC} = p$. Smaller RRC values indicate a higher likelihood of SLO attainment. Hence Torpor divides functions into high- and low-priority groups based on their RRCs, and prioritizes their requests accordingly. We develop an effective strategy to determine the boundary (i.e., RRC threshold) between the two groups that can dynamically adjust function prioritization based on the current load at a worker, thereby improving the overall SLO compliance (see details in our technical report [62]).

5.3 Scheduling and Model Swapping

While model execution latency is often stable, model swapping can incur unpredictable overhead due to PCIe bandwidth contention [21, 38]. Fig. 5 shows the topology of a worker node in Alibaba Cloud, where each pair of GPUs shares a PCIe switch and GPUs are inter-connected via NVLinks with various bandwidths[6]. The performance slowdown caused by

[6]Despite the presence of other GPU interconnects (e.g., NVSwitch in DGX A100), inter-GPU PCIe bandwidth sharing continues to necessitate interference mitigation.

Algorithm 1 Interference-Aware Request Scheduling

```
1:  function SCHEDULE(req r)
2:      A ← set of available GPUs                    ▷ A ≠ ∅, otherwise queueing r
3:      M ← set of GPUs hosting the target model
4:      if M ≠ ∅ then
5:          G ← M ∩ A
6:          if G ≠ ∅ then
7:              g ← any GPU in G
8:              Execute r on g                       ▷ No swapping
9:          else
10:             (g, m) ← GPU pair with fastest NVLink, g ∈ A, m ∈ M
11:             Execute r on g; Swap model from m    ▷ GPU-to-GPU
12:     else
13:         g ← a GPU whose neighbor is not loading models, g ∈ A
14:         if g not found then
15:             g ← a GPU whose neighbor is loading a light model, g ∈ A
16:         if g not found then
17:             g ← any GPU in A
18:         Execute r on g; Swap model from host     ▷ Host-to-GPU
```

bandwidth contention can vary among models as shown in Table 3, where larger models require more intensive data transmission and exhibit more pronounced performance degradation. Hence we propose interfere-aware scheduling to minimize PCIe contention, thereby reducing request latencies.

Interference-aware scheduling. `Torpor` exploits the direct NVLink connections between GPUs to reduce PCIe contention whenever possible. It prioritizes GPU-to-GPU over host-to-GPU model swapping to enable faster model transmission and avoid interference with concurrent PCIe traffic. When concurrent host-to-GPU swapping is unavoidable, `Torpor` avoids loading bandwidth-intensive models (e.g., Bert-qa in Table 3) simultaneously to minimize the impact of PCIe contention. Therefore, models are categorized as heavy or light based on their bandwidth requirements (see Table 4).

Algorithm 1 shows `Torpor`'s scheduling and swapping mechanisms. `Torpor` first checks whether the target model is loaded on an available GPU, and if so, directly executes it without swapping (line 8). If the model is hosted by busy GPUs, `Torpor` then schedules the request to perform GPU-to-GPU swapping, as the source and target GPUs should have a fast NVLink connection (line 11). Otherwise, `Torpor` resorts to the host-to-GPU swapping and prioritizes target GPUs whose neighbors are idle or running light models to reduce PCIe contention (line 18). Altogether, `Torpor` minimizes the interference and overhead of model swapping for each request, thus providing low inference latency.

5.4 Model Eviction Policy

Model eviction plays a critical role in reducing bandwidth contention and enhancing the overall inference performance, in conjunction with `Torpor`'s request scheduling policy. Unlike traditional cache eviction strategies which primarily aim to minimize the miss rates, `Torpor`'s model eviction policy considers the performance implications of model swapping for different models to facilitate future model loading.

We notice that swapping light models leads to negligible overhead for end-to-end performance compared with heavy ones (Table 3 and Table 4). Therefore, we tend to evict models that have little or no impact on performance when swapping. We employ two mechanisms following this insight. First, `Torpor` manages memory of all GPUs as a pool to globally optimize model placement, which ensures that each model can have up to one replica among GPUs when GPU memory is full. This allows for more efficient model caching and reduces host-to-GPU data transmission. Second, `Torpor` prioritizes light models in eviction, as swapping them leads to negligible or no PCIe bandwidth contention. When only heavy models remain, `Torpor` adopts Least-Recently-Used (LRU) policy to determine their eviction order.

6 Implementation

We have implemented `Torpor` for Alibaba Cloud, one of the world's leading commercial serverless platforms. `Torpor`'s GPU server and GPU client were implemented in 4k and 1.5k lines of C++ code, respectively. Intra-node router and cluster manager were implemented atop the relevant components in Alibaba Cloud. `Torpor`'s late-binding mechanism imposes no intrusive changes to standard cluster management logic, enabling the reuse of Alibaba Cloud's existing manager with minimal changes. In fact, `Torpor` has been successfully deployed in a real-world production environment of Alibaba Cloud (§8). We provide a container image as a function template based on PyTorch, where the original CUDA libraries (e.g., `libcudart.so`) are replaced by our GPU clients to enable GPU remoting. This requires no modification to the PyTorch framework.

7 Evaluation

In this section, we evaluate `Torpor` with the runtime sharing mode using production traces from Alibaba Cloud. We intergrate `Torpor` with runtime isolation mode into Alibaba Cloud's real-world production environment and report the results in §8.

Settings. We deploy `Torpor` at Alibaba Cloud following the realistic production specification of its serverless platform. `Torpor` runs in a cluster with up to 6 workers. Each worker node has 48 vCPU cores, 384 GB memory, and 4 NVIDIA V100 GPUs each with 32 GB memory. We use 8 popular ML models for evaluation, as shown in Table 4, and distribute them across inference functions in a round-robin manner. All functions are warmed up before running the test workloads. We compare `Torpor` against Native execution—the default approach in Alibaba Cloud—and INFless [58], a state-of-the-art serverless inference system.

Metrics. We focus on the ratio of functions meeting SLOs and

Table 4: Various models and their latencies (ms) with GPU remoting and model swapping. Underlined are heavy models where swapping via PCIe slows down the inference (see §5.3).

Model	Native	GPU remoting	Swap-PCIe	Swap-NVLink
DenseNet-169	30	25	27	26
DenseNet-201	36	28	30	30
Inception-v3	19	14	17	16
EfficientNet	17	12	13	13
ResNet-50	11	9	13	11
ResNet-101	20	14	22	16
ResNet-152	25	17	25	20
Bert-qa	42	43	144	45

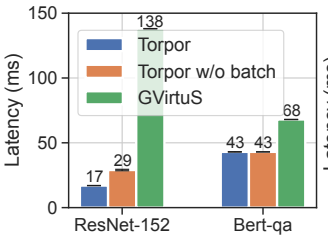


Figure 6: Inference latency with Torpor and other GPU remoting techniques.

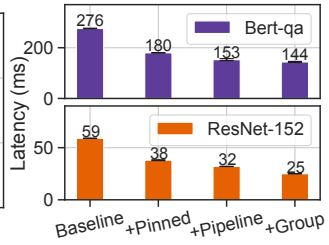


Figure 7: Performance breakdown of Torpor's model swapping via PCIe.

the GPU load in the evaluation. A function complies SLOs if its tail request latency is within a deadline. By default, we use 98^{th} tail latency, and set deadlines for CV models and Bert-qa to 80 ms and 200 ms, respectively. The GPU load is measured by the proportion of time during which the GPU is processing inference requests.

7.1 Overhead of Torpor's Late Binding

Table 4 compares the performance of 8 popular models under Native execution and Torpor with its GPU remoting (§4.2) and model swapping (§4.3). For GPU remoting, Torpor adopts efficient, asynchronous API redirection, leading to comparable performance to Native, or even better for CV models. This is because serving these models requires configuring many cuDNN descriptors where the relevant CUDA APIs are executed on the CPU side and do not require GPU resources; thus, redirecting these APIs effectively distributes CPU-side workloads across functions and the GPU server, enabling functions to access more CPUs and perform parallel computation. Note that, CPU resources are not the bottleneck in this scenario. According to our measurements, the CPU utilization of Torpor (native execution) for ResNet-152 and Bert-qa remains at 27.4% (8.8%) and 17.2% (9.2%), respectively. For model swapping, Torpor supports efficient pipeline execution through PCIe, and leveraing NVLink further improves performance.

Performance of Torpor's GPU remoting. To show the advantage of Torpor's GPU remoting, we compare it with GVirtuS [9, 31], a leading solution among publicly available GPU

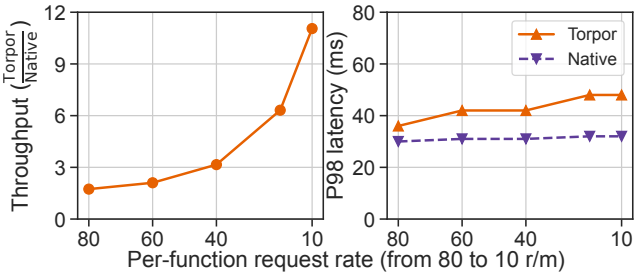


Figure 8: Performance of executing multiple ResNet-152 functions on a single GPU with Torpor's late binding (Torpor) and Native execution. We show the throughput of Torpor normalized to Native (left) and the tail latencies under varying per-function request rates (right).

remoting techniques. GVirtuS adopts a synchronous approach to API redirection. We also evaluate a variant of Torpor that disables call batching in asynchronous API redirection, i.e, "Torpor w/o batch". Fig. 6 shows the inference performance under various approaches. Torpor significantly outperforms GVirtuS and reduces latencies by 88% and 37% for ResNet-152 and Bert-qa, respectively. ResNet-152 triggers a large number of API calls during each inference, leading to high synchronization overhead for GVirtuS. Asynchronous API redirection (Torpor w/o batch) dramatically reduces the latency by 79%; with API call batching, Torpor further reduces the latency by 41%. Compared with ResNet-152, Bert-qa requires less communication in GPU remoting; therefore, the improvement from asynchronous API redirection is less but still quite significant.

Performance breakdown of model swapping. To illustrate how each of Torpor's model swapping designs contributes to performance improvement, we break down the inference performance of ResNet-152 and Bert-qa, as shown in Fig. 7. Specifically, "Baseline" directly performs model swapping and then executes inference; "Pinned" uses a pinned memory pool for improved swapping performance; "Pipeline" overlaps model swapping and execution at the granularity of individual model parameters; "Group" groups parameters for efficient pipeline execution. We note that enabling pinned memory reduces overall latencies by around 35%; parameter-level pipeline execution further reduces the latency by 15%. By grouping model parameters, Torpor achieves up to 22% performance improvement over "Pipeline", especially for ResNet-152 that consists of many small-sized parameters.

7.2 Benefits of Torpor's Late Binding

GPU efficiency for low-frequency functions. With late binding, Torpor substantially reduces per-function memory footprint, thereby enabling the consolidation of many low-frequency functions for improved GPU efficiency. We stress-test its performance by executing multiple ResNet-152 functions on a single GPU, varying request rates between 80

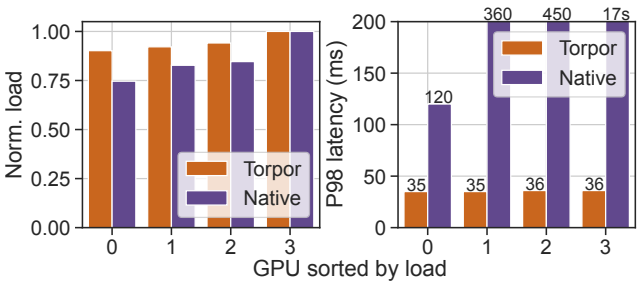


Figure 9: The normalized per-GPU load and the tail request latency with Torpor's late binding (Torpor) and Native.

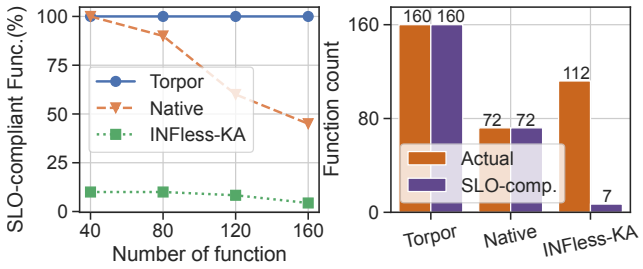


Figure 10: Performance comparison in terms of SLO compliance between Torpor, Native, and INFless-KA.

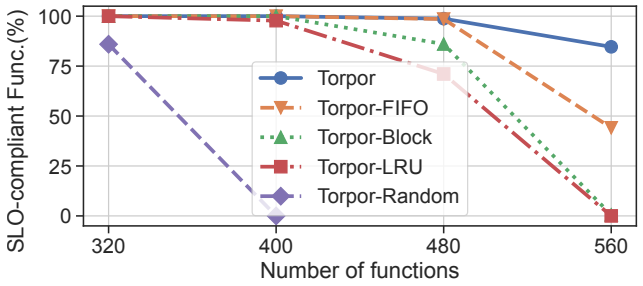


Figure 11: Ratio of SLO-compliant functions with the full Torpor and various different policies.

and 10 requests per minute (r/m)—a typical range of low-frequency functions in production traces (Fig. 2). In this setup, we execute sufficient concurrent functions on a GPU node to saturate GPU memory and ensure a high overall load. Fig. 8 compares normalized throughput and latency under Torpor and Native executions. Native's throughput declines as the per-function request rate decreases, due to its limited capacity to host many functions. In contrast, Torpor leverages host memory to accommodate many more functions, maintaining high throughput with efficient, request-level GPU sharing. For example, Torpor achieves over 10× higher throughput than Native at 10 r/m. Furthermore, even when model swapping is required, Torpor still keeps the tail latency below 50 ms.

Cross-GPU load balancing for high-frequency functions. We run 40 high-frequency ResNet-152 functions on a 4-GPU worker, where the average request rate is around 200 r/m. Fig. 9 shows the normalized per-GPU load and the tail request latency with Torpor and Native. Unlike Native, where GPUs hosting high-frequency functions can easily become overloaded due to bursts of requests, Torpor enables on-demand model migration for efficient load balancing. Therefore, Torpor achieves much less load variance across GPUs compared with Native, as shown in Fig. 9 (left). Moreover, Fig. 9 (right) shows the tail latency of requests executed on each GPU, where Native leads to extremely long tail latency (e.g., multi-seconds) due to high queueing delays. In contrast, Torpor consistantly delivers fast model inference, achieving a tail latency of around 35 ms on all GPUs.

7.3 Torpor at A Node

We next evaluate the performance of Torpor at a node. We use real-world workloads sampled from production traces (Fig. 2), where function request rates range from 5 to 30 r/m.

Performance comparison. We compare Torpor with two baselines, Native execution and INFless [58] — a state-of-the-art serverless inference system. INFless introduces a function keep-alive policy to set the lifespan of individual functions based on historical traces, denoted as INFless-KA. For a fair comparison, we implement the keep-alive policy of INFless (INFless-KA) in the Native system. Fig. 10 (left) shows the ratio of functions meeting SLOs in Torpor, Native, and INFless-KA. Fig. 10 (right) shows the numbers of functions being actually executed and being SLO-compliant, when hosting 160 functions. With fast model swapping, Torpor executes all 160 functions and also meets SLOs for all functions. In contrast, due to limited GPU memory, Native can only execute 72 out of 160 functions. INFless-KA can reclaim GPU memory via cleaning up idle functions, thereby enabling the execution of more functions (i.e., 112 functions) than Native; however, INFless-KA inevitably incurs function cold starts and results in only 7 functions being SLO-compliant.

Benefits of Torpor's policies. To understand the benefits of Torpor's policy designs, we compare the full Torpor with four baselines. 1) *Torpor-FIFO* uses a FIFO policy in request queueing rather than our SLO-aware policy (§5.2). 2) *Torpor-Random* disables our interference-aware scheduling and model swapping (§5.3), and randomly schedules a request to an idle GPU if the target model is not loaded, and then triggers model swapping. 3) *Torpor-LRU* adopts a LRU policy in model eviction rather than prioritizing models according to swapping overheads (§5.4). 4) *Torpor-Block* disables our block management policy (§4.4), and caches the released memory blocks in a single pool; when a new block is required, it directly returns a cached one in the pool if the requested size can be satisfied, otherwise it frees multiple blocks until the required memory space is available.

Fig. 11 shows the ratio of SLO-compliant functions. In particular, Torpor-FIFO is oblivious to SLOs and unable to properly prioritize functions, leading to serious SLO violations when the number of functions is large. Torpor-Block cannot reuse various-sized blocks and forces frequent memory allo-

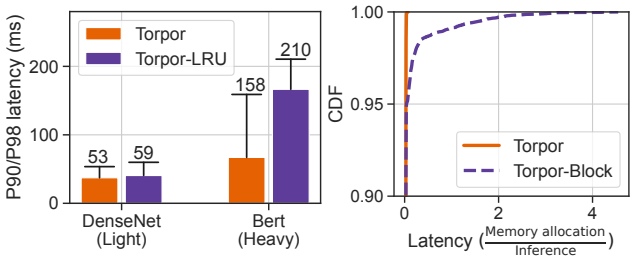


Figure 12: 90^{th} (bars) and 98^{th} (whiskers) tail latencies in `Torpor` and `Torpor`-LRU.

Figure 13: Latency CDF of memory allocation in `Torpor` and `Torpor`-Block.

cation via CUDA APIs, which incurs long delay in block allocation and harms overall performance. `Torpor`-LRU evicts heavy models often, leading to PCIe bandwidth contention during future model swapping. `Torpor`-Random leads to the worst performance due to its inefficient scheduling and model swapping policy, which does not exploit NVLink across GPUs and is oblivious to model heaviness. Compared with these baselines, `Torpor` successfully supports over 80% functions even with 560 functions, maximizing the number of SLO-compliant functions.

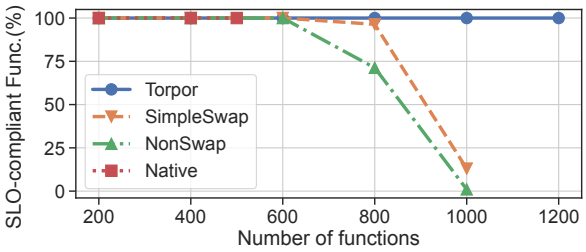
Efficiency of model heaviness. We evaluate the efficiency of model heaviness with `Torpor` and `Torpor`-LRU, using DenseNet-201 and Bert-qa as light and heavy models, respectively. Fig. 12 shows tail latencies of DenseNet-201 and Bert-qa in a mixed workload, where Bert-qa instances account for 60% of overall memory footprint. `Torpor`-LRU is agnostic to model heaviness and can evict heavy models frequently, leading to high tail latencies for Bert-qa that fail to meet its SLOs (200 ms). In contrast, `Torpor` effectively reduces tail latencies of Bert-qa without compromissing performance for DenseNet-201, achieving SLOs for both models.

Efficiency of memory allocation. Fig. 13 shows the distribution of the latencies of per-request memory allocation normalized to inference time, under `Torpor` and `Torpor`-Block. Due to efficient memory allocation and sharing (§4.4), `Torpor` incurs only negligible overhead. In contrast, `Torpor`-Block leads to high allocation overhead (e.g., over 4× than the actual inference time), harming the end-to-end performance.

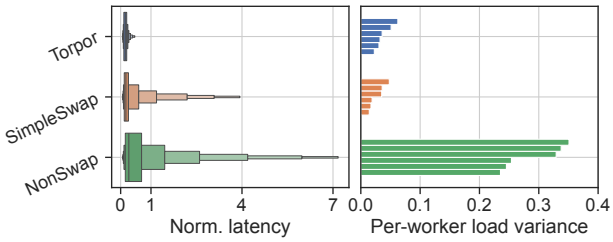
7.4 `Torpor` in A Cluster

We next evaluate `Torpor` in a cluster deployment with 6 GPU worker nodes. As running `Torpor` in a Alibaba Cloud cluster incurs additional system overhead, we set the SLOs for CV models and Bert-qa to 150 ms and 250 ms, respectively.

Baselines. We exclude INFless-KA from our cluster deployment due to its poor performance (see Fig. 10). We use three baselines: 1) *Native* uses native GPU containers bound to specific GPUs, which is the current practice in Alibaba Cloud. 2) *NonSwap* allows GPU remoting similar to `Torpor`, but disables model swapping, reducing memory footprint compared with Native. 3) *SimpleSwap* enables model swapping



(a) Ratio of SLO-compliant functions under `Torpor` and baselines.



(b) Distribution of per-request latencies normalized to deadlines (left) and the per-worker variance of GPU load normalized to the maximum (right), when running 1k functions. Boxes (left) depict the $1/128, 1/64, \ldots, 1/2, \ldots, 63/64, 127/128$ quantiles.

Figure 14: Cluster evaluation of `Torpor`.

compared with NonSwap. This approach only supports simple strategies as discussed in §7.3, including FIFO request queueing, random scheduling, and LRU model eviction.

Cluster evaluation. Fig. 14 compares `Torpor` with these three baselines. We first show the ratio of SLO-compliant functions under various number of functions. As shown in Fig. 14a, only `Torpor` can consistently meet per-function latency SLOs even with a large number of functions (e.g., over 1000). Native quickly saturates all GPU memory and only supports up to 500 functions, thus low GPU utilization. Compared with Native, NonSwap relaxes the constraint of GPU memory and enables more functions; however, it still fixes the binding between functions and GPUs, and causes GPUs overloaded by requests and leads to long tail latency. Moreover, while SimpleSwap outperforms NonSwap with model swapping, it still suffers from severe SLO violations with a large number of functions (e.g., 1k).

Fig. 14b compares the behaviors of `Torpor`, SimpleSwap, and NonSwap under 1k functions. We show the distribution of per-request latencies normalized to corresponding deadlines (left). In `Torpor`, almost every request can be served within its deadline, leading to a normalized latency less than 1. However, SimpleSwap and NonSwap suffer from long tail latency — over 4× and 7× of the respective deadlines. We also compare the per-worker GPU load of the three systems. For each worker node, we normalize the loads of its four GPUs to the maximum, and calculate the variance. Lower variance indicates better load balancing. Fig. 14b (right) plots the per-worker load variances of three solutions, each with 6 workers in total. Compared with NonSwap, `Torpor` and SimpleSwap

Table 5: Overview of Torpor's pilot production.

Metric	Value	Metric	Value
# of users	> 150	Users' cost savings	70% on avg.
# of GPUs	> 350	GPU savings	65%
# of daily requests	up to 465k		

can effectively balance GPU load across workers with model swapping, thus achieving much less load variance.

8 Torpor in Pilot Production

Torpor has been deployed in a pilot production cluster in Alibaba Cloud for beta testing. In this section, we present the testing results and our observations.

Overview of the pilot production. In the deployment, we employ a cost-efficient billing scheme in which users are charged only 10% of the GPU cost when their functions are inactive and retained in the host memory. This billing scheme aligns with Torpor's late binding mechanism and has attracted a variety of real-world inference workloads to Alibaba Cloud, including image processing, text generation, and image generation. Table 5 provides an overview of our pilot production system and the achieved savings. Currently, Torpor serves over 150 users in a cluster with more than 350 GPUs, handling up to 465k requests everyday. The system achieves an average cost savings of 70% for users compared to the previous approach of Alibaba Cloud that kept functions long-running on GPUs and billed users for the entire GPU time. Moreover, Torpor enables Alibaba Cloud to consolidate various functions for improved GPU utilization, resulting in a 65% reduction in the total number of required GPUs and cost savings for Alibaba Cloud.

Startup latency. Table 1 shows the performance of Torpor in the pilot production across various models, ranging from CNNs such as ResNet to LLMs like Llama. In the production environment, Torpor prioritizes user isolation and manages a dedicated GPU runtime for each function. Hence we also provide a detailed breakdown of the time required for model loading and runtime resumption in Table 1. Compared to cold starts in Alibaba Cloud, Torpor reduces model startup latencies by over an order of magnitude, e.g., cutting Llama2-13B's startup time to 4.4 seconds—a delay that is acceptable considering that generating all output tokens for a query can take tens of seconds [30, 68].

Case study. We next present a case study of a realistic GPU function in our pilot production, which involves text generation from input images. This function is invoked several thousands times per day and follows a request arrival pattern similar to Fig. 1. Without Torpor, this function would need to be kept long-running on a GPU for low inference latency, resulting in high GPU costs. Fig. 15 depict latency distribution and user cost of this function. For confidentiality reasons, we normalize the latency of model (and runtime) loading to the

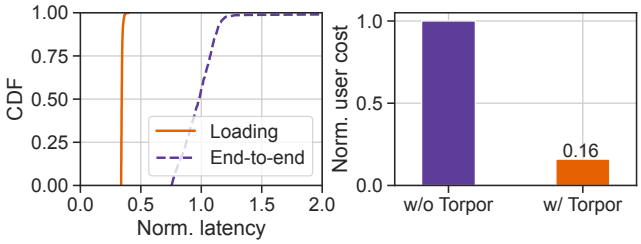


Figure 15: Latency distribution and user cost of a realistic GPU function in Alibaba Cloud.

mean of end-to-end times, and the cost to that of long-running GPU functions (i.e., w/o Torpor). The loading duration is consistent, as shown in Fig. 15 (left), which accounts for only ∼30% of the overall latency. Fig. 15 (right) illustrates the user cost, where Torpor reduces the total cost by 84% compared to the previous approach in Alibaba Cloud. Based on this real-world use case, Torpor achieves significant cost savings without compromising end-to-end performance, proving to be an effective solution for serverless inference.

Discussion. We have identified three challenges that require further investigation based on our experiences with the production cluster. *(1) Extremely infrequent functions:* We have observed that some functions are invoked extremely infrequently (e.g., a few requests per hour). However, these functions still result in substantial user costs due to their host memory usage. To address this, we plan to explore the utilization of cheaper storage for low-demand functions and investigate a multi-tier storage architecture that dynamically adapts to request patterns for improved resource and cost efficiency. *(2) Very large models:* While Torpor's late–binding design is versatile and has been applied to large models (Table 1), it currently does not support model-parallel execution across multiple GPUs. To accommodate very large models spanning multiple GPUs or nodes, it is crucial to enable efficient model partitioning and parallelization that integrates with Torpor's model-swapping mechanism. Therefore, we intend to extend Torpor to leverage various interconnects, including PCIe and NVLink (Fig. 5), for enhanced model parallel and pipeline execution. *(3) Highly bursty workloads:* While Torpor is primarily designed for efficient resource sharing among many low-frequency functions (Fig. 2), some functions may experience highly bursty workloads with short periods of extensive request arrivals. To handle such spikes, Torpor can proactively maintain host-memory-cached model replicas across multiple nodes, mitigating cold starts at the cost of increased memory usage. We plan to explore high-speed inter-node networks (e.g., RDMA) for efficient model loading between nodes, which can reduce memory footprint while maintaining low-latency model startup.

9 Related Work

Serverless Inference. In addition to the serverless platforms discussed in §2.2, there are several recent works on serverless

inference. StreamBox [56] and Dilu [43] support spatial GPU sharing for concurrent model execution; FaaSTube [55] improves data sharing within inference workflows. These works are orthogonal to Torpor and can be integrated for enhanced performance. ServerlessLLM [30] and Medusa [64] propose cold-start optimizations specialized for large language models, which is complementary to Torpor and can be used to accelerate those specific models.

Host-to-GPU data swapping. Many other systems have leveraged host-to-GPU data swapping in general deep learning and GPU workloads [23, 32, 34, 39, 46, 50, 57, 63]. For example, vDNN [46], Salus [63] and SwapAdvisor [34] leverage host memory for deep learning jobs with large GPU memory footprints; Batch-aware [39] and HUVM [23] optimize GPU memory access for general-purpose workloads; POS [35] supports efficient GPU checkpointing and restoring. Compared with Torpor, these systems are not specifically designed for model inference and do not account for its SLO attainment. Inference systems such as PipeSwitch [22] and DeepPlan [36] improve host-to-GPU model loading for fast switching. Unlike Torpor, these systems require detailed model-specific knowledge and do not target meeting model-level SLOs in a shared, multi-tenant serverless environment.

GPU remoting. GPU remoting techniques have been employed in different layers for GPU virtualization [28, 31, 45, 59]. Existing solutions like GVirtuS [31] and rCUDA [28] primarily focus on general-purpose workloads. Torpor applies GPU remoting in serverless inference, leveraging its characteristics for asynchronous, low-latency API redirection.

Spatio-temporal GPU sharing. Existing techniques have investigated the spatial and temporal GPU sharing to improve overall utilization [2, 13–15, 24, 33, 51, 56]. These techniques are orthogonal to Torpor and can be directly applied, which allow partitioning a physical GPU into multiple virtual instances to late-bind more functions.

10 Conclusion

This paper introduces Torpor, a serverless platform for SLO-aware and GPU-efficient model inference. Torpor employs a late binding approach, managing inference functions in host memory and dynamically swapping them to a pool of GPUs upon request arrivals. This approach enables pay-per-GPU-use billing and maximizes resource utilization. Additionally, Torpor proposes request scheduling and model management policies to meet latency SLOs for inference functions while minimizing resource costs. Torpor has been beta released in a large commercial serverless platform, successfully serving up to 465k requests per day and achieving 70% and 65% GPU cost savings for users and the platform, respectively.

Acknowledgments

We thank the anonymous reviewers and our shepherd, Somali Chaterji, for their insightful comments that helped improve this work. We also thank Bohui Wu and Zhexiang Zhang for their help in experiments. This work was supported in part by the Alibaba Innovative Research (AIR) Grant, RGC CRF Grant (Ref. #C6015-23G), RGC GRF Grants (Ref. #16217124 and #16210822), NSFC/RGC CRS Grant (Ref. #CRS_HKUST601/24), and CUHK-Shenzhen Research Grant (UDF01003466).

References

[1] Alibaba Cloud Function Compute. https://www.alibabacloud.com/product/function-compute.

[2] Aliyun cGPU. https://www.alibabacloud.com/help/en/container-service-for-kubernetes/latest/cgpu-overview.

[3] Aliyun Function Compute Billing Scheme. https://www.alibabacloud.com/help/en/function-compute/latest/billing-billing.

[4] Aliyun Function Compute Instance Types and Modes. https://www.alibabacloud.com/help/en/function-compute/latest/instance-types-and-instance-modes.

[5] Amazon SageMaker. https://aws.amazon.com/sagemaker/.

[6] AWS Lambda. https://aws.amazon.com/lambda/.

[7] AWS Lambda Provisioned Concurrency. https://docs.aws.amazon.com/lambda/latest/dg/provisioned-concurrency.html.

[8] Azure Functions. https://azure.microsoft.com/en-us/services/functions/.

[9] GVirtuS. https://github.com/gvirtus/GVirtuS.

[10] Llama2. https://www.llama.com/llama2.

[11] Llama3. https://www.llama.com/models/llama-3.

[12] Memory Management on Modern GPU Architectures. https://developer.download.nvidia.com/video/gputechconf/gtc/2019/presentation/s9727-memory-management-on-modern-gpu-architectures.pdf.

[13] Nvidia Multi-Instance GPU. https://www.nvidia.com/en-us/technologies/multi-instance-gpu/.

[14] Nvidia Multi-Process Service. https://docs.nvidia.com/deploy/mps/.

[15] Nvidia Virtual GPU. https://www.nvidia.com/en-us/data-center/virtual-solutions/.

[16] Qwen. https://github.com/QwenLM/Qwen.

[17] ResNet in PyTorch. https://pytorch.org/vision/stable/models/resnet.html.

[18] Stable Diffusion. https://huggingface.co/stable-diffusion-v1-5/stable-diffusion-v1-5.

[19] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. Firecracker: Lightweight virtualization for serverless applications. In *Proc. USENIX NSDI*, 2020.

[20] Ahsan Ali, Riccardo Pinciroli, Feng Yan, and Evgenia Smirni. BATCH: Machine learning inference serving on serverless platforms with adaptive batching. In *Proc. ACM/IEEE Supercomputing*, 2020.

[21] Marcelo Amaral, Jordà Polo, David Carrera, Seetharami Seelam, and Malgorzata Steinder. Topology-aware gpu scheduling for learning workloads in cloud environments. In *Proc. ACM SC*, 2017.

[22] Zhihao Bai, Zhen Zhang, Yibo Zhu, and Xin Jin. PipeSwitch: Fast pipelined context switching for deep learning applications. In *Proc. USENIX OSDI*, 2020.

[23] Sangjin Choi, Taeksoo Kim, Jinwoo Jeong, Myeongjae Jeon, Youngjin Kwon, Rachata Ausavarungnirun, and Jeongseob Ahn. Memory harvesting in multi-GPU systems with hierarchical unified virtual memory. In *Proc. USENIX ATC*, 2022.

[24] Seungbeom Choi, Sunho Lee, Yeonjae Kim, Jongse Park, Youngjin Kwon, and Jaehyuk Huh. Serving heterogeneous machine learning models on multi-GPU servers with spatio-temporal sharing. In *Proc. USENIX ATC*, 2022.

[25] Daniel Crankshaw, Xin Wang, Giulio Zhou, Michael J Franklin, Joseph E Gonzalez, and Ion Stoica. Clipper: A low-latency online prediction serving system. In *Proc. USENIX NSDI*, 2017.

[26] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[27] Dong Du, Qingyuan Liu, Xueqiang Jiang, Yubin Xia, Binyu Zang, and Haibo Chen. Serverless computing on heterogeneous computers. In *Proc. ACM ASPLOS*, 2022.

[28] José Duato, Antonio J. Peña, Federico Silla, Rafael Mayo, and Enrique S. Quintana-Ortí. rcuda: Reducing the number of gpu-based accelerators in high performance clusters. In *Proc. IEEE HPCS*, 2010.

[29] Henrique Fingler, Zhiting Zhu, Esther Yoon, Zhipeng Jia, Emmett Witchel, and Christopher J. Rossbach. Dgsf: Disaggregated gpus for serverless functions. In *Proc. IEEE IPDPS*, 2022.

[30] Yao Fu, Leyang Xue, Yeqi Huang, Andrei-Octavian Brabete, Dmitrii Ustiugov, Yuvraj Patel, and Luo Mai. ServerlessLLM: Locality-enhanced serverless inference for large language models. In *Proc. USENIX OSDI*, 2024.

[31] Giulio Giunta, Raffaele Montella, Giuseppe Agrillo, and Giuseppe Coviello. A gpgpu transparent virtualization component for high performance computing clouds. In *Proc. Euro-Par*, 2010.

[32] Arpan Gujarati, Reza Karimi, Safya Alzayat, Wei Hao, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace. Serving DNNs like clockwork: Performance predictability from the bottom up. In *Proc. USENIX OSDI*, 2020.

[33] Mingcong Han, Hanze Zhang, Rong Chen, and Haibo Chen. Microsecond-scale preemption for concurrent GPU-accelerated DNN inferences. In *Proc. USENIX OSDI*, 2022.

[34] Chien-Chin Huang, Gu Jin, and Jinyang Li. SwapAdvisor: Pushing deep learning beyond the GPU memory limit via smart swapping. In *Proc. ACM ASPLOS*, 2020.

[35] Zhuobin Huang, Xingda Wei, Yingyi Hao, Rong Chen, Mingcong Han, Jinyu Gu, and Haibo Chen. PARALLELGPUOS: A concurrent OS-level GPU checkpoint and restore system using validated speculation. *arXiv preprint arXiv:2405.12079*, 2024.

[36] Jinwoo Jeong, Seungsu Baek, and Jeongseob Ahn. Fast and efficient model serving using multi-gpus with direct-host-access. In *Proc. ACM EuroSys*, 2023.

[37] Zhipeng Jia and Emmett Witchel. Nightcore: Efficient and scalable serverless computing for latency-sensitive, interactive microservices. In *Proc. ACM ASPLOS*, 2021.

[38] Yimin Jiang, Yibo Zhu, Chang Lan, Bairen Yi, Yong Cui, and Chuanxiong Guo. A unified architecture for accelerating distributed dnn training in heterogeneous gpu/cpu clusters. In *Proc. USENIX OSDI*, 2020.

[39] Hyojong Kim, Jaewoong Sim, Prasun Gera, Ramyad Hadidi, and Hyesoon Kim. Batch-aware unified memory management in GPUs for irregular workloads. In *Proc. ACM ASPLOS*, 2020.

[40] Kenneth C. Knowlton. A fast storage allocator. *Commun. ACM*, 8(10):623–624, 1965.

[41] Jack Kosaian, K. V. Rashmi, and Shivaram Venkataraman. Parity models: erasure-coded resilience for prediction serving systems. In *Proc. ACM SOSP*, 2019.

[42] Yunseong Lee, Alberto Scolari, Byung-Gon Chun, Marco Domenico Santambrogio, Markus Weimer, and Matteo Interlandi. PRETZEL: Opening the black box of machine learning prediction serving systems. In *Proc. USENIX OSDI*, 2018.

[43] Cunchi Lv, Xiao Shi, Zhengyu Lei, Jinyue Huang, Wenting Tan, Xiaohui Zheng, and Xiaofang Zhao. Dilu: Enabling GPU resourcing-on-demand for serverless DL serving via introspective elasticity. In *Proc. ACM ASPLOS*, 2025.

[44] Xuan Peng, Xuanhua Shi, Hulin Dai, Hai Jin, Weiliang Ma, Qian Xiong, Fan Yang, and Xuehai Qian. Capuchin: Tensor-based GPU memory management for deep learning. In *Proc. ACM ASPLOS*, 2020.

[45] C. Reaño, A. J. Peña, F. Silla, J. Duato, R. Mayo, and E. S. Quintana-Ortí. Cu2rcu: Towards the complete rcuda remote gpu virtualization and sharing solution. In *Proc. IEEE HiPC*, 2012.

[46] Minsoo Rhu, Natalia Gimelshein, Jason Clemons, Arslan Zulfiqar, and Stephen W. Keckler. vDNN: Virtualized deep neural networks for scalable, memory-efficient neural network design. In *Proc. ACM/IEEE MICRO*, 2016.

[47] Francisco Romero, Qian Li, Neeraja J Yadwadkar, and Christos Kozyrakis. INFaaS: Automated model-less inference serving. In *Proc. USENIX ATC*, 2021.

[48] Mohammad Shahrad, Rodrigo Fonseca, Iñigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *Proc. USENIX ATC*, 2020.

[49] Haichen Shen, Lequn Chen, Yuchen Jin, Liangyu Zhao, Bingyu Kong, Matthai Philipose, Arvind Krishnamurthy, and Ravi Sundaram. Nexus: a GPU cluster engine for accelerating DNN-based video analysis. In *Proc. ACM SOSP*, 2019.

[50] Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Daniel Y. Fu, Zhiqiang Xie, Beidi Chen, Clark Barrett, Joseph E. Gonzalez, Percy Liang, Christopher Ré, Ion Stoica, and Ce Zhang. High-throughput generative inference of large language models with a single gpu. *arXiv preprint arXiv:2303.06865*, 2023.

[51] Foteini Strati, Xianzhe Ma, and Ana Klimovic. Orion: Interference-aware, fine-grained gpu sharing for ml applications. In *Proc. ACM EuroSys*, 2024.

[52] Huangshi Tian, Suyi Li, Ao Wang, Wei Wang, Tianlong Wu, and Haoran Yang. Owl: Performance-aware scheduling for resource-efficient function-as-a-service cloud. In *Proc. ACM SoCC*, 2022.

[53] Dmitrii Ustiugov, Plamen Petrov, Marios Kogias, Edouard Bugnion, and Boris Grot. Benchmarking, analysis, and optimization of serverless function snapshots. In *Proc. ACM ASPLOS*, 2021.

[54] Ao Wang, Shuai Chang, Huangshi Tian, Hongqi Wang, Haoran Yang, Huiba Li, Rui Du, and Yue Cheng. FaaSNet: Scalable and fast provisioning of custom serverless container runtimes at alibaba cloud function compute. In *Proc. USENIX ATC*, 2021.

[55] Hao Wu, Junxiao Deng, Minchen Yu, Yue Yu, Yaochen Liu, Hao Fan, Song Wu, and Wei Wang. Faastube: Optimizing gpu-oriented data transfer for serverless computing. *arXiv preprint arXiv:2411.01830*, 2024.

[56] Hao Wu, Yue Yu, Junxiao Deng, Shadi Ibrahim, Song Wu, Hao Fan, Ziyue Cheng, and Hai Jin. StreamBox: A lightweight GPU SandBox for serverless inference workflow. In *Proc. USENIX ATC*, 2024.

[57] Wencong Xiao, Shiru Ren, Yong Li, Yang Zhang, Pengyang Hou, Zhi Li, Yihui Feng, Wei Lin, and Yangqing Jia. AntMan: Dynamic scaling on GPU clusters for deep learning. In *Proc. USENIX OSDI*, 2020.

[58] Yanan Yang, Laiping Zhao, Yiming Li, Huanyu Zhang, Jie Li, Mingyang Zhao, Xingzhen Chen, and Keqiu Li. INFless: a native serverless system for low-latency, high-throughput inference. In *Proc. ACM ASPLOS*, 2022.

[59] Hangchen Yu, Arthur Michener Peters, Amogh Akshintala, and Christopher J. Rossbach. AvA: Accelerated virtualization of accelerators. In *Proc. ACM ASPLOS*, 2020.

[60] Minchen Yu, Tingjia Cao, Wei Wang, and Ruichuan Chen. Following the data, not the function: Rethinking function orchestration in serverless computing. In *Proc. USENIX NSDI*, 2023.

[61] Minchen Yu, Zhifeng Jiang, Hok Chun Ng, Wei Wang, Ruichuan Chen, and Bo Li. Gillis: Serving large neural networks in serverless functions with automatic model partitioning. In *Proc. IEEE ICDCS*, 2021.

[62] Minchen Yu, Ao Wang, Dong Chen, Haoxuan Yu, Xiaonan Luo, Zhuohao Li, Wei Wang, Ruichuan Chen, Dapeng Nie, Haoran Yang, and Yu Ding. Torpor: Gpu-enabled serverless computing for low-latency, resource-efficient inference. *arXiv preprint arXiv:2306.03622*, 2025.

[63] Peifeng Yu and Mosharaf Chowdhury. Salus: Fine-grained GPU sharing primitives for deep learning applications. In *Proc. MLSys*, 2020.

[64] Shaoxun Zeng, Minhui Xie, Shiwei Gao, Youmin Chen, and Youyou Lu. Medusa: Accelerating serverless LLM inference with materialization. In *Proc. ACM ASPLOS*, 2025.

[65] Chengliang Zhang, Minchen Yu, Wei Wang, and Feng Yan. MArk: Exploiting cloud services for cost-effective, SLO-aware machine learning inference serving. In *Proc. USENIX ATC*, 2019.

[66] Hong Zhang, Yupeng Tang, Anurag Khandelwal, Jingrong Chen, and Ion Stoica. Caerus: NIMBLE task scheduling for serverless analytics. In *Proc. USENIX NSDI*, 2021.

[67] Hong Zhang, Yupeng Tang, Anurag Khandelwal, and Ion Stoica. SHEPHERD: Serving DNNs in the wild. In *Proc. USENIX NSDI*, 2023.

[68] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. DistServe: Disaggregating prefill and decoding for goodput-optimized large language model serving. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, 2024.

ARTIFACT EVALUATED usenix ASSOCIATION AVAILABLE
ARTIFACT EVALUATED usenix ASSOCIATION FUNCTIONAL
ARTIFACT EVALUATED usenix ASSOCIATION REPRODUCED

TOPPINGS: CPU-Assisted, Rank-Aware Adapter Serving for LLM Inference

Suyi Li[†*], Hanfeng Lu[†*], Tianyuan Wu[†], Minchen Yu[‡], Qizhen Weng[◇], Xusheng Chen,
Yizhou Shan, Binhang Yuan[†], Wei Wang[†]
[†]*HKUST* [‡]*CUHK-Shenzhen* [◇]*TeleAI, China Telecom* *Huawei Cloud*

Abstract

Low-Rank Adaptation (LoRA) is a popular approach that adapts a base large language model (LLM) to domain-specific tasks by adding lightweight trainable adapters. In this paper, we present TOPPINGS, a system that efficiently serves many LoRA adapters derived from a common base LLM. TOPPINGS pins the base model on GPUs and dynamically loads the requested LoRA adapters from host memory as new requests arrive. In view of the high GPU loading overhead, which not only delays the time-to-first-token of the newly arrived request but also *interrupts* the ongoing decoding of *all* inflight queries when continuous batching is in use, TOPPINGS proposes a *CPU-assisted* LoRA serving approach. It simultaneously uses CPUs to compute the lightweight adaption for prefilling as the requested LoRA adapter is being loaded onto GPUs; it then switches to the GPUs after loading completes to resume the remaining computation. TOPPINGS develops a highly optimized synchronization mechanism and pipeline loading scheme to efficiently coordinate LoRA computation on the CPUs and GPUs. TOPPINGS further designs a *rank-aware scheduling algorithm* that optimally schedules heterogeneous LoRA requests to maximize the SLO attainment. Compared with the state-of-the-art LoRA serving systems, TOPPINGS improves the average request serving latency by up to 1.7× and achieves an SLO attainment of up to 99%.

1 Introduction

Large language models (LLMs) are making significant strides in generative AI [50, 60]. Deploying LLMs for domain-specific tasks requires specialization [3, 4, 12, 33, 34], which involves adapting a pre-trained base model to different downstream tasks. Low-Rank Adaptation [7, 12, 22] (LoRA) has emerged as a popular parameter-efficient fine-tuning (PEFT) approach. It preserves the base model's parameters and adds small trainable rank decomposition matrices to each Transformer layer, allowing the creation of numerous lightweight LoRA adapters from a single base model. As LoRA gains popularity, efficiently serving them with LLMs in a multi-tenant cloud becomes critically important [6, 34, 42, 46, 54].

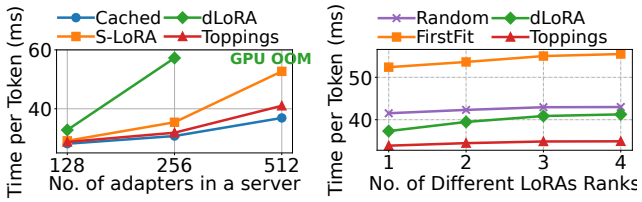


Figure 1: Time per token (TPT) performance on Llama2-7B. CACHED: the ideal case. **Left**: server node performance of existing works. **Right**: cluster-wide performance under different scheduling methods.

Typical LoRA serving systems maintain thousands of LoRA adapters on inference servers, offering APIs for end users to access specialized LLM inference [6, 35, 42, 46, 54]. Critical to the serving efficiency is *multiplexing a single base LLM* to facilitate multi-tenant LoRA serving [6, 42], where an inference server can serve requests to different LoRAs in a batch using one LLM. *Within a server*, the base LLM runs on GPUs and a stock of LoRAs reside in host memory. The server node takes two steps to serve new requests, adhering to the *continuous batching* paradigm [24, 26, 31, 56]. First, it interrupts the inference of running batch, loads demanded LoRA adapters onto GPUs, and executes the *prefill* computation for the new requests. Second, it includes the new requests into the running batch and executes the *decoding* computation for the entire batch. *At the cluster scale*, a scheduler receives users' requests and routes them across server nodes to maximize throughput [6] or ensure load balancing [54].

Despite the capability of batch serving and request orchestration, existing systems overlook two fundamental challenges in large-scale multi-tenant LoRA adapter serving [6, 42, 54].

First, on-demand loading LoRA adapters onto GPUs for new requests introduces high latency overheads. Specifically, when the desired LoRAs for new requests are not present on GPU, they must be fetched from host memory before *prefill* can commence, which can take tens of milliseconds

[*]Equal contribution

depending on the adapter size and PCIe bandwidth [15]. Consequently, this *delays* not only the time-to-first-token of the arriving request, but also the ongoing decoding of *all inflight requests*. Notably, LoRA loading is more than one-off overhead. As new requests keep arriving, the decoding of inflight requests frequently interrupts, *cumulatively* delaying token generations by 49% on average in our experiments (§2.3).

Second, in multi-tenant LoRA serving, users often request heterogeneous LoRA adapters with varying LoRA ranks [7, 12, 42]. Though heterogeneous adapters can be batched together to multiplex one base LLM using specialized CUDA kernels [6, 42], disregarding the heterogeneity and batching LoRAs blindly can increase the batch decoding latency by up to 1.5×. This increase is due to the high GPU memory I/O overheads introduced by large-rank LoRAs, which also impact the low-rank LoRAs computation in the same batch [6, 42]. Therefore, at the cluster level, the system should judiciously schedule requests of heterogeneous LoRAs to ensure that the heterogeneity does not compromise SLOs in latency. This necessitates the scheduler to accurately model the inference latency for a given batch of heterogeneous adapters, which is challenging due to the combinatorial complexity of batching heterogeneous adapters.

We introduce TOPPINGS[*], a new multi-tenant LoRA serving system to address these challenges. Compared to previous practices in LoRA serving, TOPPINGS improves efficiency and scalability of multi-tenant LoRA serving: it effectively scales to serve more LoRAs within a server and accommodate higher heterogeneity in a cluster (Fig. 1). TOPPINGS achieves this with two novel designs, which we elaborate as follows:

CPU-assisted LoRA serving. Similar to the existing LLM-multiplexing solutions [6, 42], TOPPINGS maintains the base LLM on GPUs and all adapters in host memory. When a new request arrives, TOPPINGS dynamically loads the desired adapter onto the GPU with a *loading pipeline* (§4.1). Meanwhile, it *concurrently* utilizes CPUs to compute *the LoRA adaption* to early-start the *prefill* computation. As a PEFT approach, LoRA computation is lightweight, with computational load in the order of 1 GFLOPs, making it feasible to run on CPUs. Once adapter loading completes, TOPPINGS switches to the GPU to handle adaption and the remaining prefill computation, if not finished. It then proceeds to decoding, together with other inflight requests that were previously interrupted by the new arrival under continuous batching [24, 26, 31, 56].

Nevertheless, implementing CPU-assisted LoRA serving poses several challenges. LLMs are constructed using the Transformer architecture, which consists of multiple attention layers [52]. During inference, the computed output of the base LLM needs to be synchronized with that of the LoRA models at each layer. Since these computations are split between the CPU and GPU, efficient layer-wise synchronization between the two processors is crucial to performance. Also, the frequent triggering of LoRA computations (e.g., 32 times per prefill in Llama2-7B [50]) leads to high invocation overheads, such as inter-process communication (IPC) and data transfer, increasing inference latency by 79.4% in our experiment. Moreover, CPUs' constrained parallel processing capabilities may create a new bottleneck when computing adaption for long prompts, especially in comparison to GPUs.

We address these challenges with a series of optimization designs. To efficiently coordinate on-GPU LLM computation and on-CPU LoRA computation, TOPPINGS develops a specialized CUDA operator that optimally overlaps the two computations by means of asynchronous memory copy and signaling. It also uses shared memory for fast data exchange between the base LLM process and multiple CPU LoRA processes without data copying and (de)serialization, reducing the LoRA invocation overhead to less than 1 ms. Furthermore, TOPPINGS designs a profiling-guided parallelization scheme to scale out LoRA computations across multiple CPUs, so as to eliminate the potential bottleneck. Together, TOPPINGS accelerates the request serving by 1.7× on average (§7.2).

Rank-aware request scheduling. We observe significant performance variations in decoding when batching different sets of heterogeneous LoRA adapters (§2.3). This highlights the need for intelligent request scheduling that takes into account the *rank heterogeneity* and its impact on decoding. To this end, we construct a performance model based on kernel profiling and characterization [6, 42]. This model can accurately predict the decoding latency for a given batch of heterogeneous adapters, despite the combinatorial complexity of batching heterogeneous adapters. With the performance model, we design a *rank-aware scheduler* to enhance cluster-wide serving performance and meet latency SLOs. Specifically, when a new request arrives, the scheduler examines all inference servers that possess the desired LoRA adapters and evaluates a *cost score* for each server using the performance model. This score measures the additional latency cost and SLO violation risk on the current ongoing requests if the new request were to be served on that server. The scheduler then routes the request to the server with the minimum cost score.

We have implemented TOPPINGS based on LightLLM [31] and evaluated its performance using Llama2-7B/70B [50] with requests generated from real-world and synthetic traces [40, 61]. Evaluation results show that TOPPINGS outperforms S-LoRA [42] and dLoRA [54], the state-of-the-art solutions, improving the average serving latency of inference requests by up to 1.7×. Even compared to the ideal, yet non-implementable baseline that caches all adapters on GPUs (no loading), TOPPINGS incurs only 7% overhead. We further tested TOPPINGS with long prompts of 2k to 30k tokens and observed consistent advantages. Additionally, we evaluated the rank-aware scheduling algorithm through testbed experiments and large-scale simulations. Compared to the popular scheduling policies used in existing systems [6, 54], TOP-

[*]"Topping" is a layer of food added to another food base to enhance flavor. Here, it metaphorically refers to a LoRA that adapts a base LLM's outputs.

PINGS reduces the average time per token by up to 57% and achieves an SLO attainment of 99%.

2 Background and Motivation

In this section, we give a primer to LLM inference and low-rank adaptation (LoRA). We also discuss the key challenges that arise when serving LoRA models in a multi-tenant cloud.

2.1 LLM Inference

Generative LLM inference. LLM inference is a process that involves generating a sequence of output tokens in response to an input prompt. This process consists of two phases: *prefill* and *decoding*. During the prefill phase, the LLM processes the prompt to generate the first response token and the key-value cache (KV cache) for each token; the decoding phase then uses the previous KV cache to generate new tokens iteratively and appends the KV cache accordingly. The decoding phase continues until a specified condition is met, such as emitting an end-of-sequence (<eos>) token.

LLM adaption. Parameter-efficient adaptation of LLMs enhances their performance for domain-specific tasks [33,34,46]. One popular approach is Low-Rank Adaptation or LoRA [22], which introduces an *adapter* to modify the intermediate LLM inference results while keeping the original LLM parameters unchanged. Specifically, given a pre-trained LLM weight matrix $\mathbf{W} \in \mathbb{R}^{H_1 \times H_2}$, an adapter consists of two low-rank matrices $\mathbf{A} \in \mathbb{R}^{H_1 \times r}$ and $\mathbf{B} \in \mathbb{R}^{r \times H_2}$, where r is the LoRA rank. LoRA adapts this weight matrix to $\mathbf{W}' = \mathbf{W} + \mathbf{AB}$. Let $\mathbf{y}$ be the original output of this layer given by $\mathbf{y} = \mathbf{xW}$. With LoRA adaption, the updated computation becomes

$$\mathbf{y}' = \mathbf{xW} + \mathbf{xAB} = \mathbf{xW}'. \quad (1)$$

The LoRA adapter is highly efficient in terms of parameter space because the rank $r \times (H_1 + H_2) \ll H_1 \times H_2$. Therefore, LoRA adaption is widely applied in the attention modules of transformer-based LLMs [22, 42]. When deploying LoRA-adapted models for inference, the computation load required by the LoRA adapter (**xAB**) is orders of magnitude smaller than that of the original weights **xW** in terms of floating-point operations, if we compute these two parts separately [6].

2.2 Multi-Tenant LoRA Serving

The need of LLM-multiplexing. A naive way to serve a LoRA adapter [22] is to *merge* its weights into the weights of the base LLM. However, this approach does not scale to multi-tenant LoRA serving: because one base model can only merge with one adapter at a time, serving n different LoRAs requires duplicating n copies of the base LLM, wasting GPU memory and missing opportunities for batch inference [26].

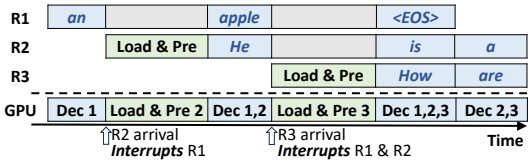


Figure 2: Continuous batching in which the decoding phase (Dec) is preempted to perform prompt processing upon a request arrival, which involves loading the requested LoRA adapter (Load) and prefilling (Pre).

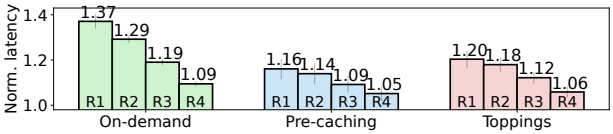


Figure 3: The normalized serving latency of first four identical LoRA requests (rank=128, 128 input tokens, 32 output tokens) with inter-arrival of 200ms under different approaches.

In practice, many LoRA adapters are developed based on common LLM series (e.g., Llama2 [50]), and multiple LoRA adapters originating from the same LLM can multiplex that LLM for GPU-efficient inference. This can be achieved by computing LoRA adaption **xAB** on the fly and adding this result back to the intermediate results **xW** before subsequent computations. As the computation of **xAB** is lightweight, multiple LoRA computations can be batched during inference.

Continuous batching. Existing LLM serving systems employ the *continuous batching* strategy optimized for LLM's iterative auto-regressive generation process [1, 24, 26, 31, 56]. Continuous batching operates at the iteration level, where completed requests are immediately removed from the running batch after each iteration and new requests can join the running batch in just one iteration, without waiting for the entire batch inference to complete. Continuous batching significantly improves the token generation throughput while minimizing the request queuing delays. Fig. 2 illustrates this batching process used in existing systems [24, 26, 31], where the decoding and prefill phases interleave as new requests arrive. Upon a request's arrival, the decoding phase (Dec) is preempted by prompt processing, which involves loading the requested LoRA adapter (Load) and prefilling (Pre). During this period, all the inflight requests are suspended, resulting in a decoding interruption. Once prompt processing completes, the new request joins the running batch, and the system combines them together to continue the decoding process.

Underutilized CPUs. LLM inference highly demands GPUs while CPUs are often underutilized. Recent LLM cluster analysis reveals that the median GPU utilization is 99%, whereas the CPU utilization is less than 10% for 75% nodes [23].

2.3 Challenges

However, simply enabling LLM-multiplexing and continuous batching is insufficient to achieve optimal performance for

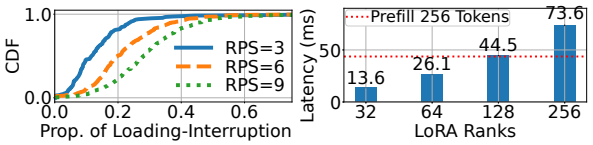


Figure 4: Left: The proportion of decoding-interruption overhead during the entire token generation of each request. Right: The latency of loading a single adapter of different ranks. The adapter applies to the projection matrices of a Llama2-7B. Prefilling 256/1k tokens takes 44ms/90ms, respectively, longer than 90%/99% of prompts in the real-world dataset [61].

multi-tenant LoRA serving, as it imposes two challenges.

C1: High cumulative decoding-interruption overhead. To save GPU memory, existing systems only cache the base LLM on GPU while keeping all its LoRA adapters in host memory [6, 42]. When a new request arrives, the corresponding adapter is loaded from the host to the GPU, initiating an adapter loading that must complete before the prefill phase can begin (Fig. 2). This results in a severe cold-start problem, where adapter loading can take between a few to tens of milliseconds depending on the adapter size* (Fig. 4-Right). Cold-start delays the time-to-first-token of the prefill request and becomes even more detrimental under continuous batching, where the arrival of a new request interrupts the ongoing decoding of all inflight queries (e.g., R3's arrival interrupts R1 and R2 in Fig. 2). Consequently, the cold-start latency adds to the decoding-interruption overhead, delaying all inflight requests. To illustrate this problem, we set up a simple experiment where five identical LoRA requests sequentially arrive, with the i-th arrival interrupting the decoding of the ongoing $i-1$ requests. Fig. 3 illustrates the output generation latency of the first four requests normalized by that of the last one with different serving approaches. Compared to pre-caching all adapters on GPU, the cold-start introduced by on-demand loading cumulatively extends the decoding interruption: earlier requests observe longer latency as they are delayed by more cold starts of late arrivals.

In a scenario where new requests keep arriving, the decoding of inflight requests is frequently interrupted, resulting in a significant *cumulative delay* in token generation. We empirically validate this by multiplexing a Llama2-7B model with 512 LoRA adapters (rank=64). These adapters have skewed popularity (Fig. 12-Left) following the Microsoft Azure Function (MAF) trace [40]. We configured Poisson request arrivals with various aggregate loads. Fig. 4-Left shows the proportion distribution of the cumulative decoding-interruption overhead, which, on average, accounts for 13%, 22%, and 29% of the entire request serving time when the aggregate load is 3, 6, and 9 requests per second (RPS), respectively. The median number of interruptions per request is 4 when RPS=3 and increases to 10 and 25 as the RPS increases to 6 and 9, respectively.

*Larger adapters are increasingly common as they usually allow LLMs to generate better response [7, 12, 51, 55].

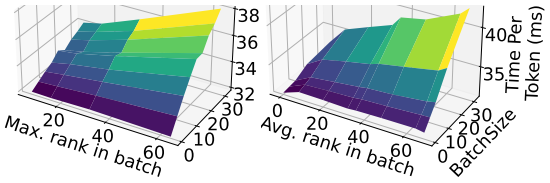


Figure 5: The decoding latency variation of batching heterogeneous LoRA adapters. Left: `BGMV`. Right: `MBGMV`.

To avoid adapter loading, a simple approach is to pre-cache all LoRA models in GPU. However, this approach is expensive: a single rank-64 adapter that adapts attention projection weights $\mathbf{W}_Q, \mathbf{W}_K, \mathbf{W}_V$ of the Llama2-7B demands approximately 100 MiB, equivalent to the size of a KV cache of 200 tokens. Pre-caching all adapters will swallow up GPU memory, degrade system throughput significantly [26], and cause GPU out of memory (OOM) errors (§7.2). A recent production trace [27] reveals that LoRA invocations follow a long-tailed distribution. Achieving a 50% hit rate requires caching over 200 LoRAs, which consumes more than 20 GiB of GPU memory for Llama2-7B. This is not well justified as most GPU memory should be reserved for KV-cache to maximize throughput [26].

C2: Cluster-level request scheduling for heterogeneous LoRA serving. In multi-tenant LoRA serving, users often request to use heterogeneous LoRA adapters with varying ranks [7, 42, 51, 55]. These heterogeneous adapters can be batched together to multiplex one base LLM using specialized kernels, such as the Batched Gather Matrix-Vector Multiplication (`BGMV`) kernel in Punica [6] or the Multi-size Batched Gather Matrix-Vector Multiplication (`MBGMV`) kernel in S-LoRA [42]. Rank heterogeneity in a batch of adapters directly affects their serving performance. Specifically, when batching a set of heterogeneous LoRA adapters, `BGMV` pads adapters of smaller ranks to the highest rank to perform batch operations, while `MBGMV` does not use padding [42]. As a result, `BGMV`'s performance is determined by the maximum rank in the batch, whereas `MBGMV`'s performance depends on the average rank. We measure the decoding latency of batch serving heterogeneous LoRAs using these two kernels with various batch configurations, and depict the results in Fig. 5. We observe significant performance variations when batching different sets of heterogeneous adapters. With the same batch size, rank heterogeneity can increase batch decoding latency by 28%. This highlights the need for *intelligent request scheduling* that considers the impacts of rank heterogeneity on performance to satisfy SLO.

To illustrate this point, we refer to a toy example shown in Fig. 6, where Instance 1 handles 24 requests with LoRA rank=32, and Instance 2 runs 16 requests with rank=64. Using Punica's `BGMV` kernel, the decoding latencies are 34.8 ms and 35.8 ms for Instances 1 and 2, respectively. With S-LoRA's `MBGMV`, the latencies are 35.3 ms and 35.9 ms for Instances 1 and 2, respectively. Assume a decoding latency SLO of 36ms,

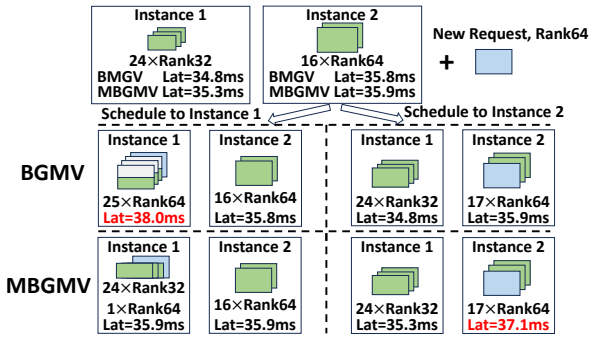


Figure 6: An toy example illustrating performance variations when scheduling a single request. With a decoding latency SLO of 36 ms, the new request should be routed to Instance 2 with BGMV, while Instance 1 with MBGMV.

System	Cumulative **loading** overhead	**(un)merging** overhead	Multi-LoRA **computation**	Cluster-wide **scheduling**
Punica [6]	High	Zero	Fast	Rank-agnostic
S-LoRA [42]	High	Zero	Fast	Rank-agnostic
dLoRA [54]	Zero	High	Slow	Rank-agnostic
TOPPINGS	Low	Zero	Fast	Rank-aware

Table 1: Comparison of existing systems on server efficiency (LoRA loading, merging, computing) and cluster scheduling.

and we need to determine the optimal schedule for a new request with rank=64. With the BGMV kernel, assigning the request to Instance 2 would meet the SLO, while sending it to Instance 1 would increase the maximum rank of the batched requests to 64, resulting in an SLO violation due to the processing of 25 higher-rank requests on Instance 1. However, with MBGMV, where latency is proportional to average LoRA ranks, scheduling the request to Instance 1 preserves the SLO, while routing it to Instance 2 leads to a violation.

However, modeling serving latency based on the rank heterogeneity is challenging due to the combinatorial complexity. With a common *maximum* serving batch size $B = 64$ and a small number of $N = 5$ different LoRA types, there are a total of $\sum_{b=1}^{B} \binom{N+b-1}{b} \approx 1.1 \times 10^7$ different rank combinations.

2.4 Concurrent Work and Their Inefficiency

Model serving system design primarily aims optimizing serving efficiency within a server and request scheduling across a cluster [20, 54, 59]. In Table 1, we compare concurrent adapter serving systems [6, 42, 54] in terms of adapter loading and computing within a server, and scheduling at the cluster scale.

Adapter loading. By default, both Punica [6] and S-LoRA [42] load adapters on demand, leading to high cumulative decoding-interruption overhead (**C1**), which we elaborate in evaluation (§7). Though S-LoRA suggests using predictive prefetching, it does not provide details, and frequent mispredictions are expected given the bursty inference request workloads [54, 59]. dLoRA [54] pre-caches all LoRA adapters into GPUs. Though pre-caching avoids loading overhead, it fall shorts in serving a large number of adapters, causing GPU OOM or degraded performance in our evaluation (§7.2).

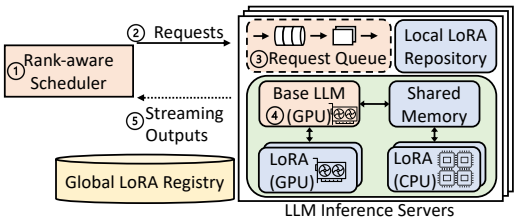


Figure 7: An architecture overview of TOPPINGS.

Multi-LoRA computation. Critical to the inference efficiency is LLM-multiplexing. Notably, Punica [6], S-LoRA [42], and TOPPINGS adopt efficient GPU kernels for the on-the-fly LoRA computation (§2.2). dLoRA [54] proposes a *dynamic batching* approach, which dynamically merges adapters with the base model to serve requests of the same adapter in a batch and unmerges adapters to serve requests of different adapters. However, deciding when to merge and unmerge adapters during inference introduces additional overhead [54] and results in slow computation, as evident in §7.2.

Scheduling. Despite the impact of heterogeneous LoRA adapters on request scheduling, the cluster scheduling policy in concurrent systems [6, 42, 54] are rank-agnostic, resulting in significant delays that violate SLOs (§7.4).

3 TOPPINGS Overview

In this section, we provide a high-level overview of TOPPINGS, a LoRA serving system that efficiently tackles the two challenges mentioned earlier. Within a server, TOPPINGS uses a *CPU-assisted* approach to hide the long cold-start latency. It uses CPUs to simultaneously execute the requested LoRA adapter while loading it onto the GPU, effectively overlapping the adapter loading with the *prefill* computation (§4) to mitigate the cumulative decoding-interruption overhead. At the cluster scale, TOPPINGS optimizes the scheduling of requests to heterogeneous LoRA adapters using a *rank-aware* scheduling algorithm, significantly enhancing cluster performance and SLO compliance (§5). Fig. 7 illustrates the system architecture, which consists of a cluster of LLM inference servers, a scheduler, and a global LoRA registry.

LLM inference server. Each LLM inference server maintains a long-running service of the base LLM on the GPU. It also stores a set of heterogeneous LoRA adapters in an in-memory *local LoRA repository*. During inference, the server coordinates LoRA computations on the CPU and GPU. Specifically, it adapts the BGMV kernel from [6] to perform LoRA computation efficiently on the GPU. For CPU-based LoRA execution, it utilizes four techniques to enhance its efficiency: 1) pipelined adapter loading, 2) asynchronous invocation, 3) shared memory data transfer, and 4) profiling-guided parallelization, which we elaborate in §4.

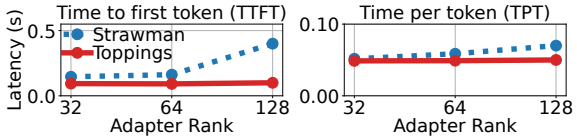


Figure 8: Performance of Llama2-7B on A100. The strawman solution increases TTFT by up to 4× and TPT by up to 1.4×.

Rank-aware scheduler. The scheduler receives requests and routes them to the appropriate servers to meet SLOs. To make informed scheduling decisions, it uses a performance model to predict the latency cost by considering the rank heterogeneity of the serving batch, which we explain in §5.

Global LoRA registry. It stores the metadata of all adapters, such as the LoRA ranks, the path to their weights file, etc.

Workflow. As illustrated in Fig. 7, new requests arrive at the scheduler (①), which uses the rank-aware scheduling algorithm described in §5 to route them to appropriate inference servers (②). Following the continuous batching strategy [56], the LLM inference server fetches requests from the request queue (③) and provides generative inference services using the corresponding LoRA adapters (④). New tokens generated by the LLM are then streamed back to the users (⑤).

4 CPU-Assisted LoRA Serving

In this section, we present TOPPINGS's design of CPU-assisted LoRA serving to address the high cumulative decoding-interruption overhead (§2.3). We start with a strawman solution. Then, we proceed to describe LoRA computation on GPU and CPU and discuss the challenges of efficiently combining the two executions to reduce the decoding-interruption overhead (§4.1). We then present optimization techniques that address these challenges (§4.2).

Strawman Solution. To reduce the cumulative decoding-interruption overhead, a strawman solution is to overlap adapter loading with ongoing decoding. That is, when a new request arrives, the system asynchronously loads its adapter onto the GPU without interrupting the ongoing decoding of inflight requests. When the adapter is fully loaded, the system then switches to the prefilling computation of the new request. Though the strawman solution can avoid decoding interruptions, it makes new requests suffer from longer loading overhead and degrades serving performance (Fig. 8), especially the time-to-first-token (TTFT), which is the time required to generate the first output token and is critical to user experience [11]. The strawman solution motivates TOPPINGS to early-start the *prefill* with LoRA computation on the CPU.

4.1 LoRA Computation on GPU and CPU

A parameter-efficient adapter, LoRA requires lightweight computation (§2.1) and can run on either GPU or CPU.

GPU LoRA. As the base LLM is "pinned" on GPU, running LoRA adapters on the same device saves the communication overhead and is usually more efficient than running them on CPU. To maximize the token throughput, LoRA computations (i.e., **xAB** in Eq. (1)) are batched in each attention layer during base LLM inference. This can be achieved with a specialized CUDA operator [6, 42]. In TOPPINGS, we adapt the Batched Gather Matrix-Vector Multiplication (`BGMV`) operator [6], which parallelizes the LoRA weight gathering and computation for efficient execution. The LoRA output is then added to the base output in the self-attention computation, following in Eq. (1). For an efficient implementation, we incorporate the operators of GPU LoRA computation into the base LLM inference process, as shown in Fig. 9.

CPU LoRA. LoRA computation, which is highly lightweight at approximately 1 GFLOPs (§2.1), can also be executed using CPUs. However, computing LoRA on CPUs requires *layer-wise synchronization* with the base LLM inference running on the GPU. Specifically, at each attention layer, the base inference process transfers the input tensor **x** in Eq. (1) from the GPU device memory to the host memory (Fig. 9). The CPU LoRA process then performs computation and transfers the result **xAB** back to the GPU device. Meanwhile, the base inference process proceeds to compute **xW**, which is finally adapted with the received LoRA output following Eq. (1). Though CPU LoRA requires synchronization, it can start immediately as the LoRA weights are already in memory. We hence utilize it to mitigate cold start and the cumulative decoding-interruption overhead in GPU LoRA.

CPU-assisted LoRA serving with pipeline loading. When a new request arrives and the corresponding adapter is not available on the GPU, the server fetches it from host memory and, in the meantime, early-starts its *prefill* computation using the CPU LoRA instead of waiting for the adapter loading to complete. As an adapter has a layered structure as the base LLM, we design a *pipeline loading* scheme to overlap communication and computation [5]. Given an adapter with N layers, we divide it into M layer groups. TOPPINGS uses the CPU to compute the first layer group, while in the meantime loading the second layer group onto the GPU. TOPPINGS carefully determines the group size to ensure that CPU computation and GPU loading of one layer group complete at around the same time by profiling the latency of CPU LoRA computation, adapter loading and GPU LoRA computation. Starting from the second group, the GPU pipeline is established, allowing TOPPINGS to parallelize GPU computation of the m-th group and GPU loading of the $(m+1)$-th group, where $m = 2, 3, \ldots, M-1$. In case of latency variations in LoRA loading and LoRA computation, Toppings can dynamically decide where to execute LoRA computation during inference. If the weight of the LoRA model layer has been loaded, Toppings executes the LoRA computation on the GPU; otherwise, it executes it on the CPU. Pipeline loading is particularly ef-

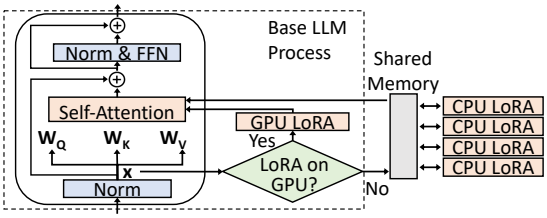


Figure 9: Illustration of coordinated LoRA computation on GPU and CPU per transformer block's attention layer.

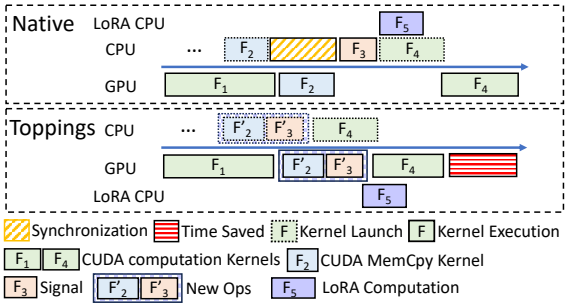


Figure 10: Execution timeline of Native LoRA Invocation using default CUDA synchronize primitives and LoRA Invocation with TOPPINGS's operator in base LLM process.

fective when serving large adapters (e.g., rank > 64), reducing the prefill latency by up to 15% compared with no pipelining ($M = 1$), as shown in §7.3.

Fig. 9 shows the orchestrated CPU and GPU LoRA computations, where CPU LoRAs run as isolated, concurrent processes for resource/failure isolation and better performance.

Challenges. Hosting LoRA computation in isolated CPU processes poses three challenges to system implementation. First, running LoRA in CPU processes requires *layer-wise synchronization* between the GPU-based LLM inference to ensure data validity. Second, frequent triggering of LoRA computation in each attention layer leads to high invocation overhead, such as inter-process data transfer. Third, using CPU to compute adaptation can be slow given its limited parallelization capability, especially when the prompt is long.

4.2 Efficient GPU-CPU LoRA Coordination

In this subsection, we tackle the system challenges mentioned earlier with three optimization techniques.

Sync-free CPU LoRA invocation. Most LLM serving systems achieve low latency through asynchronous GPU computation in PyTorch-like frameworks [26, 31, 42, 50]. By default, GPU operations (e.g., CUDA kernels) are executed asynchronously. They are first launched on CPU and enqueued to the GPU device queue, but not necessarily executed until later. For example, in Fig. 10, CUDA kernel F_2 is first launched on CPU, and then executed on GPU asynchronously. The order in which the kernels are added to the stream dictates their execution order. Hence, adapter serving requires careful coordination between base LLM inference running on GPU and LoRA invocation running on CPU to ensure correctness and good performance.

In native PyTorch, having the base LLM process invoke CPU LoRA requires explicit synchronization, which *blocks* subsequent kernels from launching. To illustrate this problem, we refer to Fig. 10-Top, which depicts the native PyTorch invocation timeline from the base LLM process's perspective. The CUDA kernel F_1 computes the input matrix $\mathbf{x}$. Meanwhile, the base LLM process launches F_2 on CPU, a CUDA `MemCpy` kernel to transfer the input matrix $\mathbf{x}$ to the host memory for CPU LoRA's access. After F_1 execution completes on GPU, the GPU executes the enqueued F_2 kernel. Once the data transfer completes, the base process uses a *signaling operator* F_3 to notify CPU LoRA processes to compute $\mathbf{xAB}(F_5)$. It then launches the next CUDA kernel F_4. This implementation requires explicit synchronization (shown as a yellow block with slashes) to ensure that the memory copy (F_2) completes before the signaling (F_3). However, the synchronization blocks the subsequent F_4 from launching, resulting in significant inference delay and GPU underutilization.

To address this issue, we introduce a customized operator that eliminates explicit synchronization by fusing an asynchronous `MemCpy` kernel with a CUDA signaling kernel. The CUDA signaling kernel writes a semaphore variable in host shared memory* to achieve signaling. As shown in Fig. 10-Bottom, instead of relying on synchronization, we fuse F_2 and F_3 into an *asynchronous* CUDA kernel $[F_2', F_3']$, where F_2' performs asynchronous `MemCpy` and F_3' asynchronously signals the intended CPU LoRA processes through changing the semaphore in the shared memory. As a result, the fused kernel $[F_2', F_3']$ can be added to the GPU device queue without waiting for the completion of F_1. Note that data validity and consistency is preserved in this case because CUDA device queue follows a sequential, strict first-in-first-out execution ordering. The semaphore variables employed will synchronize and ensure the correctness of memory read and write operations. Since the new operator requires no explicit synchronization, subsequent base model kernels, such as F_4, can launch without being blocked, eliminating unnecessary synchronization overhead. In §7.3, we show that our kernel can reduce the *prefill* latency by 15% compared to implementation with explicit synchronization. The design of the sync-free operation is not limited to LoRA computation and can be adapted to other workloads if necessary.

Shared memory data transfer. Transferring data and signals between the base LLM process and the isolated CPU LoRA processes requires inter-process communication (IPC). This is a one-to-N communication involving one base LLM inference process and multiple CPU LoRA processes. (We explain why multiple CPU LoRA processes later.) We create shared memory blocks in the host memory, i.e, DRAM, for fast inter-process signaling and data transfer, eliminat-

*The term "shared memory" refers to the shared memory in host memory.

ing the need for data copying and (de)serialization (Fig. 9). The CPU processes periodically poll from the semaphore in the shared memory. Once it changes, LoRA processes start reading the input matrix **x** from the shared memory and perform the computation **xAB**. They then write **xAB** back to the shared memory and notify the LLM inference process to incorporate the adaptation results (Eq. 1). Micro-benchmark evaluations (§7.3) demonstrate that the use of shared memory reduces data transfer overhead to less than 1 ms (Fig. 12-Right), substantially outperforming the message passing IPC employed by existing LLM frameworks [31].

Profiling-guided LoRA parallelization. Given that the CPU has lower computing power and limited parallelization capability compared to the GPU, performing LoRA adaptation using a single CPU is not scalable. Therefore, we propose a profiling-guided parallelization scheme to accelerate LoRA adaptation using multiple CPU cores. As discussed in §2.1, the adaptation computation is **xAB**, where $\mathbf{x} \in \mathbb{R}^{B \times L \times H}$ is the input matrix for B requests with L tokens, totaling $B \times L$ tokens. We first profile the performance achieved by a single core under varying workloads (Fig. 17-Left) and set the maximum number of tokens for a single CPU core to handle for computation. For example, if one core can handle c tokens, we allocate $\lceil \frac{L}{c} \rceil$ cores for computing the adaptation results of each request with weight matrix **W**. Each core is dedicated to an isolated CPU process to avoid interference. Specifically, the CPU process reads a slice of **x** from the shared memory region, performs the computation, writes the results back to the shared memory, and notifies the base LLM process accordingly. Compared to PyTorch's native multi-threading module [14], this approach achieves 1.4× speedup when using 16 CPUs for the same workload (Fig. 17-Right).

Putting it altogether, our design, as demonstrated in §7.2, can accelerate the request serving by 1.7× on average.

5 Rank-Aware Scheduling

In a multi-tenant LoRA serving system, user requests can trigger the use of heterogeneous LoRA adapters with varying ranks. As discussed in §2.3, the rank heterogeneity directly affects the performance of multi-tenant LoRA serving systems. Therefore, the scheduling strategy for handling these requests is crucial for enhancing system efficiency (**C2**): a sub-optimal strategy can drive the adapter heterogeneity in a server to a non-ideal setting that slows down token generation for both new and ongoing requests. An effective scheduler needs to be aware of the *heterogeneity-performance model* to make *rank-aware* request scheduling decisions and meet SLOs.

Performance modeling. TOPPINGS employs performance models to guide the scheduler in making informed decisions, ensuring that SLOs are met by mapping LoRA rank heterogeneity to serving performance. However, performance modeling is challenging (**C2**). Under continuous batching, when new requests are routed to a server, the server's running batch size increases, and the batch's rank heterogeneity changes as well, making the number of possible LoRA rank combinations in a batch grow exponentially.

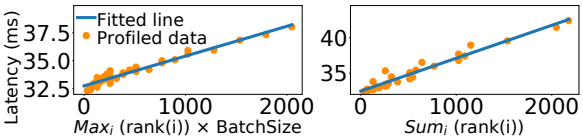


Figure 11: Performance models for BGMV (Left) and MBGMV (Right) kernels. Both linear regression models achieve a high coefficient of determination (R^2) of 0.96.

To address the challenge, TOPPINGS builds performance models based on the characterizations of the LoRA computation kernels, e.g., BGMV and MBGMV. We profile the kernels using Nvidia Nsight Compute [13] and observe that their performance is bounded by the GPU memory bandwidth, which is dominated by transferring adapter weights into L2 cache. Based on the insights, we develop generic performance models to predict the latency of a *prefill* or *decoding* iteration for a specific batch of heterogeneous adapters. These models are created through lightweight serving performance profiling, involving varying batch sizes and heterogeneous adapters on a specific GPU. We present the performance models tailored for both BGMV [6] and MBGMV [42]. For the padding-based BGMV kernel, where lower-ranked LoRAs require padding to match the highest rank, we observe that the serving performance of decoding latency is almost linear to the product of batch size and the *maximum* rank encountered in the batch (see Fig.11-Left). On the other hand, MBGMV [42] modifies the BGMV kernel to eliminate padding, improving performance with highly heterogeneous LoRA ranks but introducing additional performance overhead for computing homogeneous ranks. Through profiling, we find that under MBGMV, the serving performance scales linearly with the *sum* of LoRA ranks in a batch of heterogeneous adapters (Fig. 11-Right). Denoting the adapter rank of request i as $rank(i)$, we present performance models for these two kernels on a batch of requests $\mathcal{S}$ as two linear functions with parameters α and β, inspired by [29]:

$$\begin{aligned} \text{PERF}_{\text{BGMV}}(\mathcal{S}) &= \alpha_B \cdot |\mathcal{S}| \cdot \text{Max}_{i \in \mathcal{S}} rank(i) + \beta_B \\ \text{PERF}_{\text{MBGMV}}(\mathcal{S}) &= \alpha_M \cdot \text{Sum}_{i \in \mathcal{S}} rank(i) + \beta_M \end{aligned} \quad (2)$$

As shown in Fig. 11, our linear performance models accurately fit the profiled data. Both models achieve a high coefficient of determination (R^2) of 0.96, *suggesting the models can predict performance almost perfectly*: R^2=1 indicates a perfect fit.

The parameters α and β are specific to the hardware and LLM models. When the model or hardware changes, re-profiling is required to adjust these parameters. To construct the performance models in Fig. 11, we collect the data by profiling the serving of 7,618 requests with batch sizes of 4, 8, 16, and 32, and LoRA ranks of 8, 16, 32, and 64, which can complete in minutes. Besides, the model inference incurs

a negligible overhead in the order of 0.001ms.

Scheduling requests across inference servers. Using the established performance models, we develop a *rank-aware* scheduling algorithm (Algo. 1) for heterogeneous LoRA requests. Upon receiving a new request, the scheduler gathers information about ongoing requests from all available LLM inference servers. The scheduler identifies potential candidate servers by matching the base LLM, adapter, and GPU memory availability. If multiple candidates are found, the scheduler calculates a total cost score for each candidate server based on the performance model. This cost score measures the impact of the new requests on the performance of the server's ongoing requests. If serving the new request would cause a violation of the SLO, the cost score is assigned a large penalty. The scheduler then selects the server with the minimum cost score to handle the new request.

Algorithm 1: Rank-aware Scheduling Policy

Input: Performance models for *Prefill* and *Decoding*: $PrePerf(\cdot)$, $DecPerf(\cdot)$; average response length: avg_resp_len;

```
while True do
    Request i arrives;
    candidates ← available LLM inference servers
    for instance in candidates do
        running_batch, queue = instance.GetStats()
        cost = CalcCost(i, running_batch, queue)
        requests = len(running_batch) + len(queue)
        instance.total_cost = cost * requests
    best = min(candidates, key=lambda x: x.total_cost)
    best.serve(i)
Function CalcCost(req, running_batch, queue):
    exists = running_batch + queue
    # calculate additional prefilling time
    Δ_prefill = PrePerf(queue + req) - PrePerf(queue)
    # calculate additional decoding time
    Δ_decode = DecPerf(exists + req) - DecPerf(exists)
    cost_score = (Δ_prefill / avg_resp_len) + Δ_decode
    if DecPerf(exists + req) > SLO then
        penalty_score = float('inf')
        cost_score += penalty_score
    return cost_score
```

6 Implementation

LLM inference server. We implemented TOPPINGS's LLM Inference Server on top of LightLLM [31], an LLM serving framework based on PyTorch [37] and Triton [48]. We extended its Llama2 inference module to incorporate our LoRA adapters. This allows for easy integration with different LLMs and other popular LLM inference frameworks such as vLLM [26]. We implemented GPU LoRA adapters by adapting the `BGMV` kernels [6]. Regarding CPU LoRA, we implemented the custom CUDA kernel (§4.2) as a PyTorch Extension using `PyBind11`. Each CPU LoRA adapter runs as an isolated process, binding to one CPU core using the `numactl` command. To enable efficient batch inference, we utilize the request queue in LightLLM, which facilitates the continuous batching mechanism [26, 56].

We employ tensor parallel techniques [45] to support base LLMs that require multiple GPU devices. Tensor parallelism involves partitioning a weight matrix into multiple chunks, and each GPU device holds only one chunk to perform a portion of the computation in parallel [28]. To enable tensor parallelism for LoRA computation, TOPPINGS partitions the LoRA adapter weights (**B** in Eq.1) using the same strategy as that of the base LLMs. It performs the computation and incorporates the adaptation results into the inference intermediates in place, causing no extra communication overhead.

Scheduler & global LoRA registry. We implemented the scheduler with Algo. 1 using Python Flask. For the global LoRA registry, we utilized SQLite.

7 Evaluation

We evaluate TOPPINGS regarding the efficiency of an LLM inference server (§4) and the scheduler performance across multiple servers (§5). Our evaluation highlights include:

- TOPPINGS achieves efficient multi-tenant LoRA serving on Llama2-7B/70B, outperforming state-of-the-art baselines, e.g., S-LoRA [42] and dLoRA [54] (§7.2).
- TOPPINGS's optimizations in CPU LoRA execution are effectively illustrated by various micro-benchmarks (§7.3).
- TOPPINGS's scheduler achieves high SLO attainment and improves the performance as a cloud service (§7.4).

7.1 Experimental Setup

Model and server configs. We adopt Llama2 [50] with 7B and 70B parameters where LoRA adapters are applied to $\mathbf{W}_Q$, $\mathbf{W}_K$, and $\mathbf{W}_V$ in LLM's attention layers following the standard settings* [12, 22, 42]. We serve Llama2-7B with a single NVIDIA A100 GPU and Llama2-70B with four A100 GPUs using tensor parallelism, following [6, 32, 54].

Metrics. We use following metrics, which are considered essential in user-facing LLM serving [1, 11, 38, 42].

- ***Time-to-first-token (TTFT)*** measures how quickly users start getting the model's output after entering their prompts. It reflects the time required to process the prompt and then generate the first output token. Low waiting times for a response are essential in real-time interactions.
- ***Time-per-token (TPT)*** measures the time on average to generate an output token for each user. This metric corresponds with the perceived "speed" of the model. We calculate TPT by accounting for all tokens, including those from both the prefill and decoding phases.

*Following the setting in [6, 42], we use dummy weights (not zeros) for LoRA models, which do not affect system performance.

- ***E2E Latency*** measures the end-to-end time it takes for the model to generate the full response for a request.

Baselines. We consider the following baselines.

- CACHED represents an *ideal but impractical* method where all required LoRA adapters are *pre-cached* in *unlimited* GPU memory. It has no adapter loading overhead, thus achieving performance upper bound.
- S-LORA [42] represents a state-of-the-art multi-tenant LoRA serving framework, which is also built on top of LightLLM [31]. It loads LoRA adapters on demand and uses an adapted CUDA kernel for GPU LoRA computation.
- dLORA [54] is another LoRA serving system based on vLLM [26]. It pre-caches all LoRA adapters into GPU memory and dynamically merges/unmerges LoRAs with the base model during inference. It uses PyTorch's general matrix multiply (GEMM) kernel for LoRA computation.

Note that CACHED and TOPPINGS are built on top of LightLLM [31], the same as S-LORA [42]. We equip CACHED and TOPPINGS with the `BGMV` kernel [6] to perform GPU LoRA computation for a fair comparison. S-LORA and dLORA use their own kernels.

Workloads. We use both scaled production and synthetic workloads in our evaluation.

- ***Scaled production workload.*** We use the MAF trace [40] to generate a scaled production workload, which is widely used to emulate model serving workloads [30, 54, 59]. The trace contains invocation traffic of different functions, and we regard each function as one adapter. We randomly group adapters. Each inference server hosts a group of adapters and receives the aggregated request traffic from all the adapters it hosts. Within a group, adapters have varying probabilities of being invoked shown in Fig. 12-Left, proportional to their invocation frequency in the original trace.
- ***Synthetic workload.*** The aggregate request traffic to an LLM server follows Poisson processes with varying intensities, widely used in approximating simulated invocations [6,59]. Similar to [6,42], each request targets a distinct adapter and hence undergoes the adapter loading phase. Following [42], we place 200 adapters in an inference server.

Datasets. We refer two datasets to set each request's prompt and output length. The first is LMSYS-CHAT-1M [61], which contains input and output texts of one million real-world conversations. The second is QMSum [46, 62], which has prompts with lengths ranging from 2k to 30k tokens. QMSum comprises transcripts from real-world meetings, focusing on query-based summarization with long detailed answer [62].

7.2 LLM Inference Server Performance

In this section, we evaluate the performance of a *single* LLM inference server, which is subject to adapter loading and computing. For a fair comparison, we employ homogeneous LoRA ranks in this section, eliminating the potential performance differences that result from the way different LoRA computation kernels handle heterogeneous ranks [6, 42].

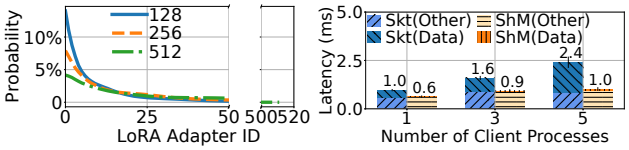


Figure 12: Left: LoRA invocation probability. X-axis: adapter ID sorted by invocation probability. Right: CPU LoRA computation time. Each process receives data of 16 tokens. *Skt*: Domain socket for inter-process communication (IPC). *ShM*: Shared memory for IPC. *Data*: Time for transfering data to other process via IPC. *Other*: Time for all other operations.

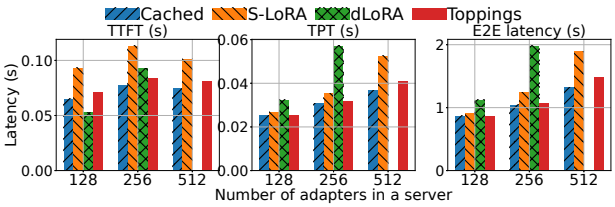


Figure 13: Performance with varying number of adapters under the scaled MAF workloads. dLORA causes GPU OOM when there are 512 adapters in the server.

Scaled production workloads. We first evaluate TOPPINGS on the scaled production workload. Fig. 12-Left illustrates the skewed distribution of adapter popularity. We evaluate each baseline with an increasing number of LoRAs served in an inference server. With more LoRA adapters, the server will receive higher request loads, and each new request is more likely to invoke a new LoRA adapter that needs to be loaded onto GPU on demand (Fig. 12-Left). Therefore, a larger number of LoRAs manifests higher cumulative loading overhead. The average aggregate *RPS* for 128/256/512 adapters is 1.5/3.6/7.7, scaled from the original trace. We set the LoRA *rank* = 64 in this setting.

We measure serving performance using the metrics defined in §7.1. See Fig. 13. When 128 LoRA adapters are in a server, the impact of cumulative adapter loading overhead is negligible because the request traffic is low, and most new requests do not require adapter loading. Compared to CACHED, S-LORA increases TTFT by 43%, TPT by 7%, and E2E latency by 5% on average, mainly due to the one-off adapter loading overhead introduced by new requests. dLORA achieves better TTFT as it pre-caches all LoRA adapters into GPU and dynamically merges LoRAs with the base LLM to accelerate *prefill* computation [54]. However, its frequent LoRA (un)merging operations impact the *decoding*, increasing the TPT by 29% and E2E latency by 29% compared to CACHED. TOPPINGS outperforms by only increasing TTFT by 9%.

However, as the number of LoRA adapters increases to 512, adapter loading incurs prohibitively high cumulative overhead, which can hinder S-LORA from scaling to efficiently serve

Setup	Model Size	Datasets	No. of LoRAs	Rank	RPS
S1	7B	LMSYS	200	32	9
S2	7B	LMSYS	200	64	6
S3	7B	LMSYS	200	64	9
S4	7B	QMSum	50	256	3
S5	70B	LMSYS	200	32	3
S6	70B	LMSYS	200	64	3

Table 2: Evaluation setups with the synthetic workload.

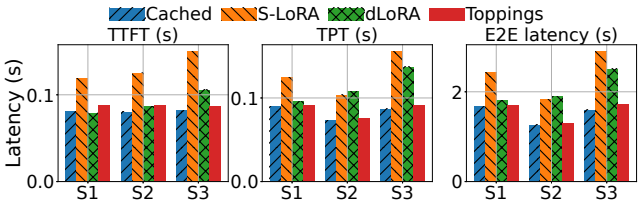


Figure 14: Serving performance. Setup: S1, S2, S3 in Tab. 2.

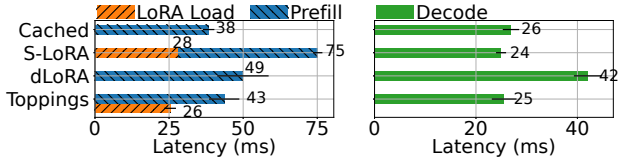


Figure 15: Prefill and decoding latency at LLM server.

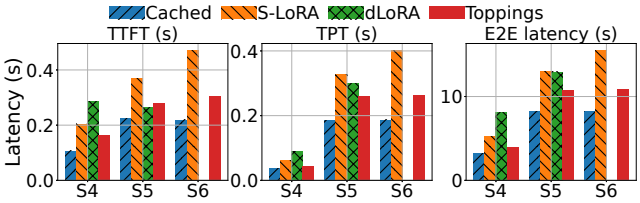


Figure 16: Serving performance. Setup: S4, S5, S6 in Tab. 2. dLoRA causes GPU OOM in S6.

a large number of adapters. Compared to the CACHED, S-LORA increases TPT by 43%, and E2E latency by 43% on average. Nevertheless, TOPPINGS can rival CACHED, accelerating S-LORA by 1.25×, 1.29×, 1.28× in terms of TTFT, TPT and E2E latency, respectively.

dLORA falls short in serving numerous LoRA adapters in a server. When 256 adapters are in a server, dLORA incurs significant computation overhead, increasing TPT/E2E latency by 86%/90%, due to its LoRA computation operator. Worse, deploying 512 adapters in a server can cause GPU OOM and the server cannot spin up (Fig. 13).

Synthetic workloads. We evaluate each baseline with various setups using the synthetic workloads, as shown in Tab. 2.

1) Setup S1, S2, and S3. We explore different LoRA ranks and request traffic in these three settings. As discussed in §2.3, two factors affect TOPPINGS's advantages (§2). The first is LoRA rank, which determines the size of a LoRA adapter: smaller rank leads to shorter loading latency. The second is the workload traffic, which determines the frequency of adapter loading (decoding interruption).

As shown in Fig. 14, TOPPINGS achieves the most performance advantage in **S3**, which has the largest LoRA rank and highest RPS. TOPPINGS rivals the performance of CACHED by introducing tolerable overheads, slightly increasing the TTFT by 6%, TPT by 6%, and the E2E latency by 7%. Compared to the CACHED baseline, S-LORA introduces high overhead due to the cumulative loading interruption, increasing TTFT by 82%, TPT by 80%, and E2E latency by 81% on average. dLORA incurs no adapter loading overhead as it preloads all adapters but falls short in calculating LoRA adaption (Eq. 1), degrading the computation performance. Besides, dLORA decides online whether to merge/unmerge the LoRA adapters with the base model to provide service [54], further increasing prefill latency when RPS is high. Compared to the CACHED baseline, dLORA increases TTFT by 29%, TPT by 58%, and E2E latency by 57% on average. We further evaluate TOPPINGS's performance under conditions of reduced CPU availability, where each CPU LoRA computes 4× more tokens. Despite the constrained CPU resources, TOPPINGS maintains consistent performance advantages over the baseline systems thanks to the pipeline loading design (§4.1). Compared to the CACHED baseline, TOPPINGS increases the TTFT by 17%, TPT by 11%, and E2E latency by 10%. The pipelined loading design allows TOPPINGS to promptly switch to GPU for LoRA computation, effectively mitigating the impact of reduced CPU availability.

S1 and **S2** represents less challenging settings, with smaller LoRA ranks or lower request traffic. As shown in Fig. 14, S-LORA/dLORA incurs smaller overhead than that in **S3** while TOPPINGS rival the performance of CACHED.

Fig. 15 explains TOPPINGS's advantage from the LLM inference server's side. The latencies of *decoding* iterations achieved by CACHED, S-LORA, and TOPPINGS are similar, while S-LORA have a long *prefill* iteration due to the adapter loading overhead. TOPPINGS leverages the CPU-assisted design (§4) to avoid the loading overhead. dLORA incurs no loading overhead as it pre-cache all adapters but slows down LoRA computing, leading to the long *decoding* latency.

Resource Utilization. Throughout the request serving in **S3**, the average GPU utilization of S-LORA/dLORA/TOPPINGS is 46%/81%/56%. dLORA's GPU utilization is much higher due to its LoRA (un)merging overhead and computation whereas S-LORA and TOPPINGS use high-performance kernels adapted from BGMV [6]. The average CPU utilization of S-LORA/dLORA/TOPPINGS is 4%/4%/46%, respectively, with each request consuming an average of 27ms for CPU-assisted LoRA computation. TOPPINGS exploits the underutilized CPUs [23] to accelerate adapter serving, which is overlooked by S-LORA and dLORA.

2) Setup S4. We stress test each baseline using the long prompts from the QMSum dataset [46, 62] that have lengths ranging from 2k to 30k tokens. We adopt Lloco's workflow [46] that serves these prompts with LoRA adapters. Given a lengthy prompt, Lloco first processes the prompt into token embeddings using the AutoCompressor [8, 46]. Subsequently, Lloco sends a request with the embeddings to query

the LLM and a designated LoRA adapter. As larger ranks contribute to higher response quality for long prompts [7], we use adapters with *rank* = 256. As shown in Fig. 16, TOPPINGS maintains superior with minimal overhead compared to CACHED. dLORA's slow LoRA computation is more salient with the long prompts in this setting.

3) Setup S5,S6. In these two settings, we evaluate each baseline with Llama2-70B using tensor parallelism [32]. We re-implement S-LORA by adapting CACHED since existing works [6, 42] have not released code in multi-GPU settings. We also emulate dLORA by importing its LoRA computation into CACHED to avoid the performance differences resulted from different backends (vLLM [26] v.s. lightLLM [31]), which are evident in the multi-GPU settings [10, 49, 64]. See Fig. 16. In **S5**, S-LORA/dLORA/TOPPINGS increases TTFT by 65%/18%/24%, TPT by 65%/59%/30%, and E2E latency by 61%/61%/33%. compared to CACHED. In **S6**, dLORA fails due to GPU OOM and TOPPINGS achieves consistent benefits, accelerating TTFT by 1.6×, TPT by 1.5×, and E2E latency by 1.4×, compared to S-LORA.

7.3 Microbenchmark Evaluations

We further evaluate the effectiveness of the CPU LoRA optimizations (§4) at the microbenchmark level.

Pipelined adapter loading. Pipelined adapter loading allows TOPPINGS to promptly prefill prompts on GPU instead of waiting for the adapter to be fully loaded (§4.1). To show its effectiveness, we measure the *prefill* performance w/ and w/o pipelined adapter loading, where the pipelined adapter loading accelerates the *prefill* computation by 1.2× and 1.1× when using adapter with *rank=128* and *rank=64*.

Sync-free CPU LoRA invocation. To analyze the performance of our optimized CPU LoRA invocation kernel (§4.2), we compare the *prefill* latency using PyTorch's native implementation and our optimized kernels. While *prefilling* 64 tokens, TOPPINGS's kernel gains a 10% speedup, reducing the latency from 67ms to 61ms. While *prefilling* 256 tokens, TOPPINGS's kernel gains a 15% speedup, reducing the latency from 145ms to 126ms.

Shared memory data transfer. We compare the latency of computing CPU LoRA with different IPC methods: shared memory and UNIX domain socket. We measure the time it takes to perform LoRA computation and the data round trip cost. Fig. 12-Right shows that as the number of receiver processes increases, the domain socket approach suffers from linear time increase in data transmission, whereas the shared memory approach obtains near-constant performance.

Multi-CPU computation. We first profile the LoRA computation performance in a *prefill* phase with a single CPU. We profile it with different workloads (number of tokens to process). We run the profiling on a Llama2-7B model with a A100 card using Intel Platinum 8369B CPU. As shown in Fig. 17-Left, the CPU has limited parallelism and does not scale to fit high workloads. We observe similar results using AMD EPYC 7R32 CPU. Fig. 17-Right illustrates the performance of *prefilling* a prompt of 256 tokens with TOPPINGS's multi-CPU design (§4.2) or the native multi-core utilization of PyTorch multi-threading module [14]. TOPPINGS's design achieves up to 1.4× speedup.

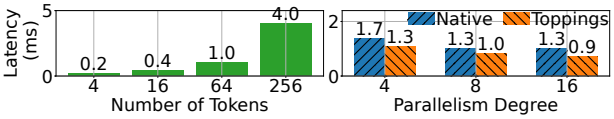


Figure 17: Left: CPU computation time of **xAB** in the *prefill* phase. Right: CPU computation for 256 tokens with different parallelism. Native: PyTorch native multi-threading [14].

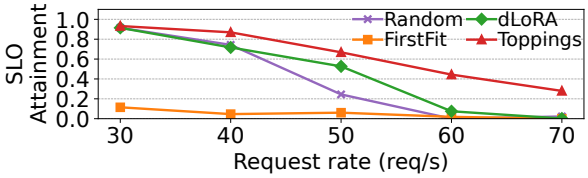


Figure 18: Scheduler performance under varying RPS.

7.4 Cluster Scale Performance

Following [59], we run cluster scale experiments in two settings: an 8-instance real-world testbed and a large-scale simulation. We use four different LoRA ranks in this section.

Baselines. Upon the arrival of new requests, we consider the following scheduling baselines for comparison:

- dLORA [54] enables reactive request migration to balance the workloads among servers.
- FIRSTFIT scheduler picks a server following the first-fit bin-packing strategy, which is also adopted by Punica [6].
- RANDOM scheduler randomly picks an inference server.

Performance model. The performance model can predict performance almost perfectly with a high R^2 of 0.96 (§5). Besides, the model takes 0.005ms for one inference.

Testbed evaluation. We set up a cluster testbed consisting of 8 Llama2-7B models to evaluate each scheduler under varying Poisson request traffic. TOPPINGS's LLM inference server is used as the LoRA serving backend. The SLO is set regarding TPT, as it reflects the perceived "speed" of the inference service. The SLO is set to be 1.5× higher than that achieved by LLM inference without LoRA. See Fig. 18. TOPPINGS's rank-aware scheduler outperforms other baselines by achieving the highest SLO attainment across all settings. When the request rate is low (RPS=30), the RANDOM and dLORA perform well as they tend to scatter the requests across servers, mitigating the impacts of rank heterogeneity. In contrast, the FIRSTFIT degrades as it tends to consolidate requests to heterogeneous LoRAs on a few servers, exacerbating rank heterogeneity. As the request rate increases, the performance of rank-agnostic

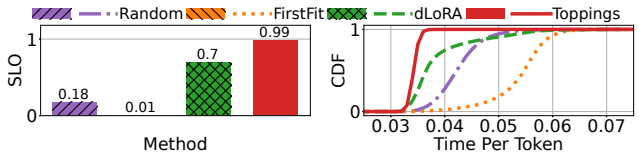


Figure 19: Scheduler performance across 60 instances.

schedulers, e.g., dLoRA, plummets, while Toppings maintains consistent advantages, improving SLO attainment by 21% and 26% when the RPS is 40 and 50, respectively.

Large-scale simulation. To ensure the fidelity of the simulator, we reuse the code from the LightLLM [31] and maintain the same serving logic. We obtain the *prefill* and *decoding* latency for the simulator by extensive profiling. We include all 40,000 functions from the MAF trace [40], with an aggregated RPS≈340, and use 60 server instances. The SLO is set regarding TPT, which is 1.5× higher than that of LLM inference without LoRA. Fig. 19 compares the SLO attainment of each scheduler and the CDF of requests' TPT. Toppings's rank-aware scheduler achieves the highest SLO attainment and speeds up the average TPT by 22/23/57% compared to dLoRA/Random/FirstFit. We see similar results in the simulation if replacing the `BGMV` in Toppings with `MBGMV`.

8 Related Work and Discussion

LLM inference. Optimizing LLM inference is the target of recent studies [2,16–19,21,25,30,38,47,57]. Orca [56] proposed iteration-level continuous batching to improve the throughput of LLM serving. Further, vLLM [26] addressed the issue of the GPU memory fragmentation resulting from LLM's KV Cache, improving serving throughput by high GPU efficiency. Sarathi-Serve executes chunked-prefills, which combines *prefill* and *decoding* computation to maximize serving throughput [1]. DeepSpeed-Ulysses enables efficient LLM training and inference for long prompts using sequence parallelism [25]. FlexGen [44] supported LLMs with limited GPU memory, maximizing serving throughput by efficiently processing tensors. Toppings is compatible with these optimizations, has already supported continuous batching and optimized GPU memory management mechanism [31].

Multi-tenant LoRA serving. In addition to the works discussed in §2.4, PetS [63] proposed a unified framework for serving various adapters in discriminative language models. However, it lacks iterative decoding and is orthogonal to Toppings. Recent works [43] have also explored the fairness issues in multi-tenant LLM serving, where some users may receive more resources than others. The primary goal of Toppings's rank-aware scheduling algorithm is to address performance variations caused by heterogeneous LoRA ranks, while not explicitly tackling fairness concerns. A straightforward approach to ensuring fairness is to limit the number of requests a user can make within a specified time frame, a strategy also adopted by OpenAI [36]. It is worth noting that fairness in LLM serving is not a challenge unique to LoRA serving. For instance, the Virtual Token Counter (VTC) scheduling algorithm [43] guarantees fairness not only for general LLM serving but also for adapter-based serving. Importantly, Toppings does not introduce additional unfairness to the system and is fully compatible with fairness strategies like VTC.

Multi-model inference serving. A series of systems optimize multi-model inference serving, including Clipper [9], MArk [58], Nexus [41], INFaaS [39], Clockwork [20], Shepherd [59], and AlpaServe [28]. They optimize batching, caching, and model placement of serving multiple models. However, they do not specially support generative LLMs and heterogeneous LoRA adapters, leading to optimization gaps.

Discussion. Toppings's design is versatile, supporting various LLM series and not restricted to a specific framework, as most LLMs share a backbone of Transformer blocks and utilize the attention mechanism [52]. The CPU-assisted design is compatible with existing GPU servers, which are typically equipped with abundant CPUs but have low CPU utilization [23, 53]. For example, the AWS g5.48xlarge instance has 192 vCPU cores. Besides a recent analysis of production traces from GPU clusters reveals significant underutilization of CPU resources [23]. This observation suggests that spare CPU cycles are readily available to support a CPU-assisted design without impacting overall cluster performance. For fault tolerance, our current implementation isolates CPU processes, with each bound to a specific CPU. Additionally, running redundant CPU processes enhances fault tolerance.

9 Conclusion

This paper presents Toppings, an efficient multi-tenant LoRA serving system. In a nutshell, Toppings multiplexes base model to serve many LoRA adapters in a batch, coordinates LoRA computation on CPU and GPU to avoid cold-start, and employs a rank-aware scheduler to meet SLOs. Compared to existing systems, Toppings significantly improves serving efficiency by reducing the request serving latency by up to 47% and achieves an SLO attainment of 99%.

Acknowledgement

We thank our shepherd, Liting Hu, and the anonymous reviewers for their valuable feedbacks that help improve the quality of this work. We also thank Xiaozhe Yao for his feedback and assistance in the early stage of this work. This work was supported in part by Huawei Cloud Research Grant, RGC CRF Grant (Ref. #C6015-23G), RGC GRF Grants (Ref. #16210822 and #16217124), and NSFC/RGC Collaborative Research Scheme (Ref. #CRS_HKUST601/24 and #CRS_PolyU501/23).

References

[1] Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav Gulavani, Alexey Tumanov, and Ramachandran Ramjee. Taming Throughput-Latency tradeoff in LLM inference with Sarathi-Serve. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, 2024.

[2] Reza Yazdani Aminabadi, Samyam Rajbhandari, Ammar Ahmad Awan, Cheng Li, Du Li, Elton Zheng, Olatunji Ruwase, Shaden Smith, Minjia Zhang, Jeff Rasley, and Yuxiong He. Deepspeed-inference: Enabling efficient inference of transformer models at unprecedented scale. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2022.

[3] AnyScale. Announcing anyscale private endpoints and anyscale endpoints fine-tuning. https://bit.ly/40eobib, 2024.

[4] Microsoft Azure. Customize a model with azure openai service. https://bit.ly/3WbehwU, 2024.

[5] Zhihao Bai, Zhen Zhang, Yibo Zhu, and Xin Jin. PipeSwitch: Fast pipelined context switching for deep learning applications. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020.

[6] Lequn Chen, Zihao Ye, Yongji Wu, Danyang Zhuo, Luis Ceze, and Arvind Krishnamurthy. Punica: Multi-tenant lora serving. In *Proceedings of Machine Learning and Systems 2024 (MLSys 2024)*, 2024.

[7] Yukang Chen, Shengju Qian, Haotian Tang, Xin Lai, Zhijian Liu, Song Han, and Jiaya Jia. LongLoRA: Efficient fine-tuning of long-context large language models. In *The Twelfth International Conference on Learning Representations*, 2024.

[8] Alexis Chevalier, Alexander Wettig, Anirudh Ajith, and Danqi Chen. Adapting language models to compress contexts. In *The 2023 Conference on Empirical Methods in Natural Language Processing (EMNLP 23)*, 2023.

[9] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J Franklin, Joseph E Gonzalez, and Ion Stoica. Clipper: A {Low-Latency} online prediction serving system. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 613–627, 2017.

[10] Cydia2018. Performance comparison between lightllm and vllm. https://github.com/ModelTC/lightllm/issues/116.

[11] Databricks. Llm inference performance engineering: Best practices. https://www.databricks.com/blog/llm-inference-performance-engineering-best-practices, 2023.

[12] Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. Qlora: Efficient finetuning of quantized llms. In *Advances in Neural Information Processing Systems (NeurIPS 23)*, 2023.

[13] Nvidia Developer. Nvidia nsight compute. https://developer.nvidia.com/nsight-compute, 2024.

[14] PyTorch Docs. Cpu threading and torchscript inference. https://pytorch.org/docs/stable/notes/cpu_threading_torchscript_inference.html, 2023.

[15] Jianbo Dong, Zheng Cao, Tao Zhang, Jianxi Ye, Shaochuang Wang, Fei Feng, Li Zhao, Xiaoyong Liu, Liuyihan Song, Liwei Peng, Yiqun Guo, Xiaowei Jiang, Lingbo Tang, Yin Du, Yingya Zhang, Pan Pan, and Yuan Xie. Eflops: Algorithm and system co-design for a high performance distributed training platform. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2020.

[16] Jiangfei Duan, Runyu Lu, Haojie Duanmu, Xiuhong Li, Xingcheng Zhang, Dahua Lin, Ion Stoica, and Hao Zhang. Muxserve: Flexible spatial-temporal multiplexing for multiple llm serving. In *ICML*, 2024.

[17] Yao Fu, Leyang Xue, Yeqi Huang, Andrei-Octavian Brabete, Dmitrii Ustiugov, Yuvraj Patel, and Luo Mai. Serverlessllm: Locality-enhanced serverless inference for large language models. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, 2024.

[18] Yichao Fu, Siqi Zhu, Runlong Su, Aurick Qiao, Ion Stoica, and Hao Zhang. Efficient LLM scheduling by learning to rank. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024.

[19] Waris Gill, Mohamed Elidrisi, Pallavi Kalapatapu, Ammar Ahmed, Ali Anwar, and Muhammad Ali Gulzar. Meancache: User-centric semantic cache for large language model based web services. In *2025 IEEE International Parallel and Distributed Processing Symposium (IPDPS 25)*, 2025.

[20] Arpan Gujarati, Reza Karimi, Safya Alzayat, Wei Hao, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace. Serving DNNs like clockwork: Performance predictability from the bottom up. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020.

[21] Cunchen Hu, Heyang Huang, Liangliang Xu, Xusheng Chen, Jiang Xu, Shuang Chen, Hao Feng, Chenxi Wang, Sa Wang, Yungang Bao, et al. Inference without interference: Disaggregate llm inference for mixed downstream workloads. *arXiv preprint arXiv:2401.11181*, 2024.

[22] Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. LoRA: Low-rank adaptation of large language models. In *International Conference on Learning Representations (ICLR 22)*, 2022.

[23] Qinghao Hu, Zhisheng Ye, Zerui Wang, Guoteng Wang, Meng Zhang, Qiaoling Chen, Peng Sun, Dahua Lin, Xiaolin Wang, Yingwei Luo, Yonggang Wen, and Tianwei Zhang. Characterization of large language model development in the datacenter. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, 2024.

[24] Huggingface. Text generation inference. https://github.com/huggingface/text-generation-inference.

[25] Sam Ade Jacobs, Masahiro Tanaka, Chengming Zhang, Minjia Zhang, Reza Yazdani Aminadabi, Shuaiwen Leon Song, Samyam Rajbhandari, and Yuxiong He. System optimizations for enabling training of extreme long sequence transformer models. In *Proceedings of the 43rd ACM Symposium on Principles of Distributed Computing*, PODC '24, 2024.

[26] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles (SOSP 23)*, 2023.

[27] Suyi Li, Lingyun Yang, Xiaoxiao Jiang, Hanfeng Lu, Zhipeng Di, Weiyi Lu, Jiawei Chen, Kan Liu, Yinghao Yu, Tao Lan, Guodong Yang, Lin Qu, Liping Zhang, and Wei Wang. Katz: Efficient workflow serving for diffusion models with many adapters. In *Proc. USENIX ATC*, 2025.

[28] Zhuohan Li, Lianmin Zheng, Yinmin Zhong, Vincent Liu, Ying Sheng, Xin Jin, Yanping Huang, Zhifeng Chen, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. AlpaServe: Statistical multiplexing with model parallelism for deep learning serving. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, 2023.

[29] Ashraf Mahgoub, Edgardo Barsallo Yi, Karthick Shankar, Sameh Elnikety, Somali Chaterji, and Saurabh Bagchi. ORION and the three rights: Sizing, bundling, and prewarming for serverless DAGs. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, 2022.

[30] Xupeng Miao, Chunan Shi, Jiangfei Duan, Xiaoli Xi, Dahua Lin, Bin Cui, and Zhihao Jia. Spotserve: Serving generative large language models on preemptible instances. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 24)*, 2024.

[31] ModelTC. Light llm. https://github.com/ModelTC/lightllm.

[32] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, Amar Phanishayee, and Matei Zaharia. Efficient large-scale language model training on gpu clusters using megatron-lm. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '21, 2021.

[33] OpenAI. Custom instructions for chatgpt. https://openai.com/blog/custom-instructions-for-chatgpt, 2023.

[34] OpenAI. Gpt-3.5 turbo fine-tuning and api updates. https://bit.ly/4j9QVS2, 2023.

[35] OpenAI. Openai documentation: Fine-tuning. https://platform.openai.com/docs/guides/fine-tuning, 2024.

[36] OpenAI. What are the best practices for managing my rate limits in the api? http://bit.ly/43Fk7e1, 2025.

[37] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: an imperative style, high-performance deep learning library. In *Proceedings of the 33rd International Conference on Neural Information Processing Systems (NeurIPS 19)*, 2019.

[38] Pratyush Patel, Esha Choukse, Chaojie Zhang, Aashaka Shah, Inigo Goiri, Saeed Maleki, and Ricardo Bianchini. Splitwise: Efficient Generative LLM Inference Using Phase Splitting . In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA 24)*, 2024.

[39] Francisco Romero, Qian Li, Neeraja J. Yadwadkar, and Christos Kozyrakis. INFaaS: Automated model-less inference serving. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, 2021.

[40] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, 2020.

[41] Haichen Shen, Lequn Chen, Yuchen Jin, Liangyu Zhao, Bingyu Kong, Matthai Philipose, Arvind Krishnamurthy, and Ravi Sundaram. Nexus: A gpu cluster engine for accelerating dnn-based video analysis. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 322–337, 2019.

[42] Ying Sheng, Shiyi Cao, Dacheng Li, Coleman Hooper, Nicholas Lee, Shuo Yang, Christopher Chou, Banghua Zhu, Lianmin Zheng, Kurt Keutzer, Joseph E. Gonzalez, and Ion Stoica. S-LoRA: Serving thousands of concurrent lora adapters. In *Proceedings of Machine Learning and Systems 2024 (MLSys 2024)*, 2024.

[43] Ying Sheng, Shiyi Cao, Dacheng Li, Banghua Zhu, Zhuohan Li, Danyang Zhuo, Joseph E. Gonzalez, and Ion Stoica. Fairness in serving large language models. In *Proc. OSDI*, 2024.

[44] Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Beidi Chen, Percy Liang, Christopher Ré, Ion Stoica, and Ce Zhang. Flexgen: High-throughput generative inference of large language models with a single gpu. In *International Conference on Machine Learning (ICML 23)*, 2023.

[45] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism, 2020.

[46] Sijun Tan, Xiuyu Li, Shishir G Patil, Ziyang Wu, Tianjun Zhang, Kurt Keutzer, Joseph E. Gonzalez, and Raluca Ada Popa. LLoCO: Learning long contexts offline. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, 2024.

[47] Yifan Tan, Cheng Tan, Zeyu Mi, and Haibo Chen. Pipellm: Fast and confidential large language model services with speculative pipelined encryption. *arXiv preprint arXiv:2411.03357*, 2024.

[48] Philippe Tillet, Hsiang-Tsung Kung, and David Cox. Triton: an intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, 2019.

[49] tonylin52. lightllm vs vllm. https://github.com/ModelTC/lightllm/issues/79.

[50] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. Llama 2: Open foundation and fine-tuned chat models, 2023.

[51] Undi95. Undi95/llama2-13b-no-robots-alpaca-lora. https://huggingface.co/Undi95/Llama2-13B-no_robots-alpaca-lora, 2024.

[52] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems (NeurIPS 17)*, 2017.

[53] Qizhen Weng, Wencong Xiao, Yinghao Yu, Wei Wang, Cheng Wang, Jian He, Yong Li, Liping Zhang, Wei Lin, and Yu Ding. MLaaS in the wild: Workload analysis and scheduling in Large-Scale heterogeneous GPU clusters. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, 2022.

[54] Bingyang Wu, Ruidong Zhu, Zili Zhang, Peng Sun, Xuanzhe Liu, and Xin Jin. dLoRA: Dynamically orchestrating requests and adapters for LoRA LLM serving. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, 2024.

[55] YeungNLP. Yeungnlp/longqlora-llama2-7b-8k-lora. https://huggingface.co/YeungNLP/LongQLoRA-Llama2-7b-8k-lora, 2024.

[56] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. Orca: A distributed serving system for Transformer-Based generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, 2022.

[57] Zhongkai Yu, Shengwen Liang, Tianyun Ma, Yunke Cai, Ziyuan Nan, Di Huang, Xinkai Song, Yifan Hao, Jie Zhang, Tian Zhi, Yongwei Zhao, Zidong Du, Xing Hu, Qi Guo, and Tianshi Chen. Cambricon-llm: A chiplet-based hybrid architecture for on-device inference of 70b llm. In *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2024.

[58] Chengliang Zhang, Minchen Yu, Wei Wang, and Feng Yan. {MArk}: Exploiting cloud services for {Cost-Effective},{SLO-Aware} machine learning inference serving. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 1049–1062, 2019.

[59] Hong Zhang, Yupeng Tang, Anurag Khandelwal, and Ion Stoica. SHEPHERD: Serving DNNs in the wild. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, 2023.

[60] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, et al. Opt: Open pre-trained transformer language models. *arXiv preprint arXiv:2205.01068*, 2022.

[61] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Tianle Li, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zhuohan Li, Zi Lin, Eric Xing, Joseph E. Gonzalez, Ion Stoica, and Hao Zhang. LMSYS-chat-1m: A large-scale real-world LLM conversation dataset. In *The Twelfth International Conference on Learning Representations (ICLR 24)*, 2024.

[62] Ming Zhong, Da Yin, Tao Yu, Ahmad Zaidi, Mutethia Mutuma, Rahul Jha, Ahmed Hassan Awadallah, Asli Celikyilmaz, Yang Liu, Xipeng Qiu, and Dragomir Radev. QMSum: A New Benchmark for Query-based Multi-domain Meeting Summarization. In *North American Association for Computational Linguistics (NAACL 21)*, 2021.

[63] Zhe Zhou, Xuechao Wei, Jiejing Zhang, and Guangyu Sun. PetS: A unified framework for Parameter-Efficient transformers serving. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, 2022.

[64] zzb610. Comparison between pageattention and tokenattention. https://github.com/ModelTC/lightllm/issues/379.

QFactory: Accelerating Quantized Large Language Model Serving with Qtile Graphs

Qihao Zhang Mingshu Zhai Rui Sun Jidong Zhai

Tsinghua University

Abstract

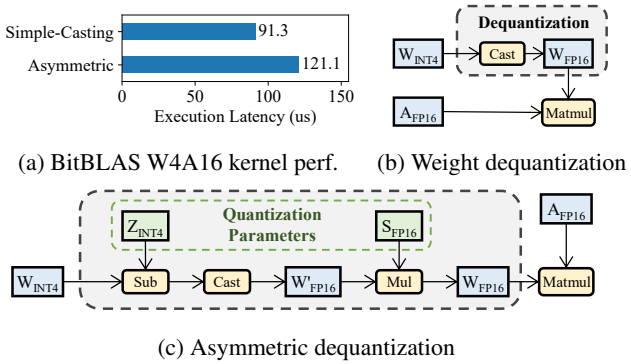
Quantization is a critical technique for accelerating large language models. To achieve tangible speedups, weight dequantization must be performed on-the-fly, necessitating tailored quantized kernels for various quantization algorithms and precision formats. Existing methods typically rely on a static eager execution paradigm for dequantization operations, which overlooks a broader range of potential optimizations, leading to suboptimal performance.

In this paper, we present QFactory, an efficient compilation framework designed to generate high-performance quantized kernels. QFactory introduces a novel Qtile abstraction that facilitates the representation of quantized tensors, transforming the traditional tensor computation graph into a Qtile-graph (Qgraph). Leveraging this QGraph abstraction, QFactory first explores graph-level Qtile computation transformations to generate equivalent QGraphs, thereby expanding the search space for optimizations. Subsequently, QFactory employs operator-level Qtile scheduling to identify optimal memory loading strategies for each Qtile within the QGraph before generating the final code. Experimental results demonstrate that QFactory achieves an average performance improvement of 1.66× over existing systems and delivers 1.23× end-to-end generation speedup when integrated into state-of-the-art large language model serving systems.

1 Introduction

Quantized large language model (QLLM) serving is increasingly promising, as it facilitates the deployment of powerful and effective LLMs within limited resource environments. Many quantization techniques [6, 10, 11, 14, 22, 23, 42, 43, 47] have been proposed to reduce the deployment cost of leading LLMs [1, 2, 32, 34–36].

The core of quantization techniques is eliminating memory pressure in LLM serving. Due to the autoregressive nature of LLMs, the decoding phase exhibits memory-bound computational characteristics. The computational bottleneck



(a) BitBLAS W4A16 kernel perf. (b) Weight dequantization

(c) Asymmetric dequantization

Figure 1: Quantized kernels for serving QLLMs.

mainly lies in the loading of model parameter matrices from GPU memory [14, 20]. To address this problem, quantization algorithms compress model parameters and store them in low-precision formats (e.g., from 16-bit FP16 to 4-bit INT4), significantly reducing DRAM memory requirements. Additionally, the quantized computation kernels [23, 30, 42] only load the compressed weights and dequantize them to higher precision before performing computations with the activation tensor, as shown in Figure 1b. Thus, the decoding speed is boosted because of the reduced memory access.

As the model bit-width becomes extremely low, **fine-grained quantization** algortihms [17, 30, 47] have been proposed to better preserve model accuracy, which typically applies an **asymmetric** and **group-wise** method. A representative fine-grained quantization algorithm is GPTQ [10]. It performs *asymmetric* quantization by compressing model weights with two auxiliary quantization parameters, *zero point Z* and *scaling factor S*. Compared to the simple type-casting dequantization in Figure 1b, asymmetric dequantization in Figure 1c introduces extra computation and memory overhead. Additionally, it is a *group-wise* quantization algorithm since these auxiliary parameters are shared among a small group of parameters within a weight tensor.

Although fine-grained quantization algorithms hold promis-

ing in preserving model capability at extremely low bit-width, the inefficient execution of dequantization hurts their effectiveness. The issue stems from the use of custom kernels that incorporate on-the-fly dequantization operations [12, 23], namely quantized kernels, which are poorly supported by the current deep learning compilers [4, 31, 39]. For instance, as shown in Figure 1a, the state-of-the-art low-precision deep learning compiler BitBLAS [39] experiences a significant performance degradation when compiling asymmetric quantized kernels. Compared to the simple type-casting W4A16 kernel, asymmetric quantized kernel compiled by BitBLAS exhibits 30% higher latency. Even worse, the performance issue becomes more pronounced when scaling to lower bit-widths. As the bit-width of weight parameters decreases and more complex quantization algorithms are adopted, the additional memory access introduced by dequantization becomes relatively significant, emerging as a new performance bottleneck.

Existing systems fail to process the extra dequantization in quantized kernels efficiently due to their **eager execution paradigm** for dequantization operations. Upon encountering quantized values during execution, this static approach rigidly converts them immediately according to the quantization algorithm's definition, limiting the potential for optimization. In contrast, our insight is that a **deferred execution paradigm** of dequantization can improve end-to-end performance. The dequantization operations can be deferred to the following tensor in the computation graph instead of applying to the original quantized tensor. Compared to this novel deferred paradigm, the conventional eager paradigm has two following limitations.

Limited Searching Space Quantized programs introduce new dequantization operations, which alter the original computation graph. These changes increase the graph's complexity and create additional opportunities for graph-level transformations. Our investigation reveals that exploring this transformation space can yield improved kernel performance. However, existing approaches remain unaware of these changes in the computation graph and eagerly execute dequantization operations on the original graph, thereby failing to exploit this potential optimization space.

Underutilized Memory Bandwidth Quantization algorithms introduce additional quantization parameters associated with weight parameters that are computed during dequantization. These quantization parameters are often shared across a certain scope of weight parameters. However, existing approaches fail to leverage this shared property and instead perform dequantization independently for each weight parameter, leading to underutilized memory bandwidth. Our findings show that by differentiating the data loading for various tensor tiles, overall bandwidth utilization can be improved.

To address these challenges, we propose QFactory, an efficient compilation framework for generating high-performance quantized kernels. The core design of QFactory is Qtile, an annotated tensor expression for quantized tensors. With this abstraction, QFactory discovers that Qtiles can be propagated along the computation graph from operator inputs to operator output. In this way, tensor dequantization is deferred to later computation, enabling a more flexible placement of dequantization operations.

To compile a quantized kernel, QFactory first transforms the user-specified quantized program into a Qtile-graph (Qgraph) by replacing the quantized tensors in computation graph with Qtiles. Then QFactory optimizes the kernel generation of quantized programs at two levels. Firstly, QFactory applies *Qtile Computation Transformation* to deduce mathematically equivalent Qgraphs, exploring a wider range of graph transformation opportunities. Subsequently, QFactory takes all discovered Qgraph candidates and generates code for them. QFactory identifies data reuse as a crucial key factor of kernel performance, and performs *Differentiated Qtile Scheduling* to maximize the overall data reuse possibilities. Finally, QFactory employs a template-based kernel generation and a machine learning based hyperparameter selector for efficient auto-tuning.

To evaluate the effectiveness of QFactory, we conduct experiments on GPUs across different generations, ranging from NVIDIA V100 to newest NVIDIA H100 GPU. Our evaluation shows that QFactory outperforms state-of-the-art low-precision deep learning compiler by 1.66× on single kernel performance, and can accelerates end-to-end decoding speed of various scale QLLMs by 1.23× when integrated into the popular LLM serving framework vLLM [21].

2 Background

2.1 Quantization of Large Language Models

While the advancements in large language models have yielded impressive results, their substantial resource consumption presents a significant challenge for model serving. Current LLMs generate tokens in an autoregressive manner, resulting in low batch sizes and leading to a memory-bound computational scenario [30, 42].

Researches have found quantization as a promising solution for reducing the large memory footprints and accelerating LLM decoding, particularly in resource-constrained circumstance such as edge devices. To maintain good accuracy with lower bit-widths, various quantization algorithms [10, 11, 30] have been developed. These algorithms propose the use of various custom data types for storing compressed weight, such as x-bit integers [17], NormalFloat [7], and even arbitrary fixed bit-width types defined by lookup tables [29]. They also introduce additional data storage, such as scaling factors and zero points, as quantization parameters to help recover the

original weight magnitude.

Take asymmetric W4A16 quantization as an example, which is the quantization format adopted in GPTQ [10]. To quantize a half-precision weight tensor w in model, $\min(w)$ and $\max(w)$ are mapped to the lower and upper bound of destination data range, namely -8 and $+7$ in INT4 data type, respectively. The remaining values in tensor w are mapped linearly. Specifically, the quantization procedure can be formulated as:

$$scale = \frac{\max(w) - \min(w)}{2^4 - 1}$$

$$zero = C_{F2I}\left[\frac{0 - \min(w)}{scale}\right]$$

$$Q_{FP16 \to INT4}(w)[i] = C_{F2I}\left[\frac{w[i] - \min(w)}{scale}\right]$$

where *scale* is a half-precision scaling factor and *zero* is an INT4 zero point. They are auxiliary parameters associated with tensor w. The C_{F2I} function represents the type casting from a half-precision real value to integer, which is often implemented with round-to-nearest method.

When inference on a quantized LLM, the quantized model weights need to be dequantized back to the original data type for computation. The dequantization process of above quantization method can be formulated as:

$$DEQ_{INT4 \to FP16}(w_{INT4})[i] = C_{I2F}(w_{INT4}[i] - zero) \times scale$$

It is straightforward to verify that the above equation is capable of recovering the original weight values, except for the precision loss that occurs during the casting between real and integer data types.

To balance between model accuracy, compression rate, and algorithm complexity, researchers have proposed a spectrum of quantization methods with changeable configurations including symmetric/asymmetric quantization (with or without zero point), quantized data types and quantization granularity (the scope of data sharing same quantization parameters). This wide range of diversity poses great challenges to efficient quantized kernel generation.

2.2 Compiling Deep Learning Models

Deep learning compilers have been proposed to accelerate model execution through optimizations on different levels. We conclude existing works to the following two categories:

Graph-level Transformation Deep learning models can be represented as tensor computation graphs, where vertices correspond to operators and edges represent the flow of tensor data between these operators. Previous works [16, 18, 25, 37, 45] explore the transformation space on the computation graph to find optimal kernel mappings. These systems focus primarily on algebraic transformations at the graph level, and attain better performance through kernel fusion or selection of kernels with higher utilization. They leverages lower-level tensor program compilers like TVM [4] for code generation.

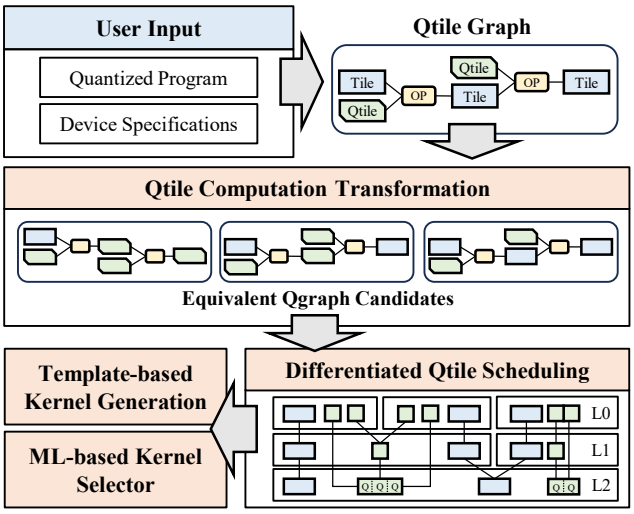


Figure 2: QFactory overview.

Operator-level Memory Scheduling Other compilers [31, 46] focus on efficient operator fusion and kernel generation through a tensor tile abstraction and apply memory scheduling to improve memory efficiency. BitBLAS [39] extends this tile abstraction to support low-precision data formats, enabling the compilation of quantized deep learning models. However, these approaches apply scheduling optimizations solely to model weight tensors and eagerly execute dequantization. As a result, they struggle to effectively handle the unique requirements of quantized models with auxiliary parameters, where more flexible dequantization strategies and finer-grained memory optimizations are necessary for optimal performance.

3 Overview

3.1 QFactory Framework

Figure 2 provides an overview of the QFactory framework. At the entry point, QFactory accepts a user-provided quantized program as input, which are subsequently transformed into a Qtile-graph by replacing certain tensors in the computation graph with the newly designed Qtile abstraction (§3.2). For common quantization algorithms, QFactory provides skeleton templates of Qgraphs, requiring users to define only key settings, such as bit-width, data type, and quantization algorithm. Alternatively, users can directly provide a Qgraph constructed with a combination of Qtiles and standard tensor tiles for greater flexibility.

Thereafter, QFactory performs *Qtile Computation Transformation* (§4) to explore the Qgraph transformation space by propagating Qtiles along operators in Qgraph, generating Qgraph candidates with guaranteed mathematical equivalence. QFactory then analyzes these candidate graphs, and inserts dequantization operations as needed. Afterward, lower-level code is generated for each Qgraph. To optimize overall memory bandwidth utilization, QFactory employs *Differentiated Qtile Scheduling* (§5) to determine the best data placement and loading strategy for Qtiles, maximizing data reuse. Finally, QFactory applies template-based kernel generation (§6.1) to enable instruction-level parallelism and develops a ML-based kernel selector (§6.2) to speedup the selection of optimal kernel configurations.

3.2 Qtile Design

QFactory proposes a novel abstraction for quantization programs, called Qtile, which serves as a critical bridge between quantized program and efficient kernel implementation. Essentially, Qtile is a tensor extended with annotations of quantization attributes. The term "Qtile" is used instead of "Qtensor" because tensors on parallel devices, such as GPUs, are computed in smaller slices, or data tiles. This terminology aligns with the tile-graph abstraction introduced in Welder [31].

Qtile extends tensors with two types of annotations: mapping function and group pattern, representing the quantization algorithm and the scope within which quantization parameters are shared, respectively.

Mapping Function with Quantization Parameters The mapping function encodes all detailed information required for dequantization, including quantization algorithm and its auxiliary parameters. It specifies the storage format of compressed weights (e.g., 4-bit integers for W4A16 quantization), the compression methods (e.g., simple type-casting or asymmetric quantization), and the storage format of any additional quantization parameters. Rather than viewing these quantization parameters as separate data tiles, Qtile incorporates them directly into its extended annotations. This allows QFactory to utilize the relationship between weight tensor and its associated quantization parameters, enabling data reuse opportunities. Theoretically, Qtile can represent arbitrary quantization formats by defining the mapping function as a lookup table. This flexibility allows Qtile to handle unconventional quantization methods, ensuring compatibility with diverse application schemes.

Quantization Group Pattern Quantization is often performed with specific granularity. Tensor elements are divided into groups where scaling or other operations are applied independently. This fine-grained approach enables better approximation and preserves tensor's values. Within each quantization group, the quantization parameters are shared, making group pattern critical for enabling data reuse. Qtile includes the group pattern information in its annotation and provides four commonly used group pattern attributes, ranging from coarse-grained to fine-grained: tensor (t), channel (c), block (b), and individual (i).

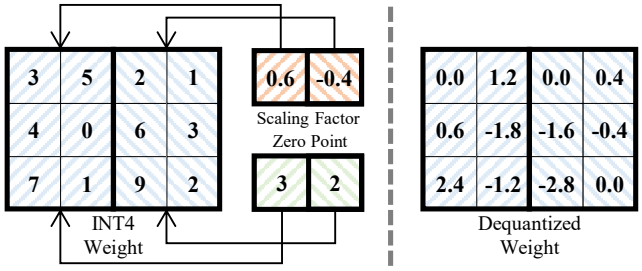


Figure 3: Qtile example of asymmetric INT4 quantization with a group pattern of 3×2 blocks.

Qtile Example We illustrate a Qtile for asymmetric 4-bit quantization with a block group pattern in Figure 3. In this example, the quantized weight tensor is divided into blocks of 3×2, with each block associated with a pair of quantization parameters: a 4-bit integer zero point and a 16-bit half-precision scaling factor. To dequantize a value in the weight tensor, the computation $(W - zero) \times scale$ is performed. For instance, the element in the second row and third column of the weight matrix is dequantized as $(6-2) \times (-0.4) = -1.6$.

Through Qtile, QFactory provides users with a unified interface to define existing quantization algorithms or design new ones. By embedding quantization information directly into the tensor, Qtile balances versatility and efficiency, facilitating wider compilation optimizations to improve kernel performance.

4 Qtile Computation Transformation

With the Qtile abstraction, QFactory replaces certain tensors in the computation graph with quantized tensors, thereby forming a Qtile-graph (Qgraph). As a result, operators within the graph may now receive both Qtiles and normal tensor tiles as input. In existing approaches, such operators are executed eagerly, with dequantization operations performed immediately whenever a quantized tensor is encountered. This strategy leads to a rigid, static translation of Qgraph, limiting the potential for further optimizations.

In contrast, QFactory introduces a compilation procedure for Qgraphs, deferring dequantization operations to later stages by performing *Qtile computation transformations*. Through the transformation rules, Qtiles can actively participate in operator computations, generating new Qtiles as output. As a result, Qtiles are propagated throughout the graph, creating equivalent Qgraphs.

Algorithm 1: Qtile Propagation Search

Input: G^Q: Initial Qtile-graph of quantized program.
Output: S: Set of equivalent Qgraphs discovered.

```
1  Function QtilePropagate(G^Q):
2      S ← ∅ ;
3      Q ← List() ;
4      Q.Push_Back(G^Q) ;
5      while Q.Size() > 0 do
6          g^Q ← Q.Pop_Front() ;
7          S.Insert(g^Q) ;
8          for Op ∈ g^Q do
9              ITiles ← Op.GetInputTiles() ;
10             OTile ← Op.GetOutputTile() ;
11             if ! OTile.IsQtile() and ITiles.hasQtile()
                and CanCompute (Op, ITiles) then
12                 NTile ← ComputeQtile(Op, ITiles) ;
13                 g'^Q ← Replace(g^Q, OTile, NTile) ;
14                 if g'^Q ∉ S and g'^Q ∉ Q then
15                     Q.Push_Back(g'^Q) ;
16     return S ;
```

4.1 Qtile Propagation Search

QFactory proposes *Qtile Propagation Search* for graph-level optimization, generating all possible Qgraph candidates, as outlined in Algorithm 1. Given a Qtile-graph, QFactory enumerates all operators within the graph (Line 8) and determines whether Qtile computation transformation can be applied to each operator. If the input tiles (which may include both Qtiles and normal tiles) of the operator satisfy the transformation rules, they are processed by the *ComputeQTile* function to produce an output Qtile (Line 12). This output Qtile then replaces the original output tile in the graph, resulting in a new Qgraph. The propagation process is repeated on the newly generated Qgraph. QFactory employs a breadth-first search approach to iteratively explore all possible Qgraphs. Finally, a set containing all alternative Qgraphs is returned, which will be passed to subsequent optimization stages.

The key component of the algorithm is the *ComputeQtile* function, which takes the operator and its input tiles as arguments and produces a Qtile expression that is equal to the original output tile. Since Qtiles are quantized tensors with additional annotations, these annotations must be considered in computation. The Qtile computation involves two types of transformation: group pattern transformation and mapping function transformation, corresponding to the two types of annotation in the Qtile design.

4.2 Group Pattern Transformation

The computation on Qtiles leads to transformation in the group pattern. For instance, when an element-wise binary operator (e.g. add, mul) is applied to a normal tile and a quantized Qtile, the group pattern from the Qtile operand is broadcast to the other operand, causing the output Qtile to adopt the same group pattern as the input Qtile.

There are also cases where operators are applied to two Qtile inputs. In such circumstance, QFactory must check whether the group patterns of the input Qtiles can be transformed into a common one. For example, when two block-grouped Qtiles with different block strides (W_1, H_1) and (W_2, H_2) are added, QFactory calculates the greatest common divisor (gcd) of the block strides to determine a common group pattern to which both Qtiles can be converted. In this case, a finer-grained group pattern $(\gcd(W_1, W_2), \gcd(H_1, H_2))$ is identified. Then, QFactory modifies the group pattern of these Qtiles to the common pattern and broadcasts the associated quantization parameters correspondingly. This ensures that different Qtiles can be computed with same group pattern.

4.3 Mapping Function Transformation

Since the mapping function is closely tied to the dequantized value of Qtile tensors, the mapping function of Qtiles also evolves as Qtiles propagate through various operators.

Take the linear mapping function for example, where each element in Qtile is quantized with scaling factor and zero-point bias. The dequantized value in a linearly-quantized Qtile can be represented as $(X_z^s)_{i,j} = s \times X_{i,j} + z$, where $X_{i,j}$ is the compressed tensor value in low precision format, and s and z represent the scaling factor and bias, respectively, which are shared across the quantization group.

Table 1 demonstrates the output Qtile expression resulting from the element-wise addition operator applied to different types of Qtiles. The operands may include non-quantized normal tiles (X) or Qtiles with scaling factors (X^s), bias offsets (X_z), or both (X_z^s). The Qtile computation transformation extends conventional tensor computations to a broader domain of quantized tensor operations.

From Table 1, quantized parameters can be redistributed between different operands. For instance, adding A_{z_1} to a non-quantized tile B can be transformed to $A + B_{z_1}$, with the bias offset z_1 shifted from the quantization parameters of tile A to those of tile B. Furthermore, in the case of plus A_{z_1} with $B_{z_2}^{s_2}$, the result can be transformed to the sum of one Qtile ($B_{z_1+z_2}^{s_2}$) and one normal tensor tile (A). This transformation minimizes the number of dequantization operations required when generating code, offering potential performance improvements.

Next, we present the Qtile transformation table for the matrix multiplication operator in Table 2. The transformation rules for Matmul operator produces more complicated result

Table 1: Mapping function transformation for the element-wise tensor addition operator. Marker $\times$ indicates that no valid transformation exists for the given Qtile inputs.

+	A	A_{z_1}	$A^{s_1}_{z_1}$
B	$A+B$	$A+B_{z_1}$	$A^{s_1}+B_{z_1}$
B^{s_2}	$\times$	$A^{s_1}+B^{s_2}_{z_1}$	$A^{s_1}+B^{s_2}_{z_1}$
B_{z_2}	$A_{z_2}+B$	$A_{z_1+z_2}+B$	$A^{s_1}+B_{z_1+z_2}$
$B^{s_2}_{z_2}$	$A_{z_2}+B^{s_2}$	$A+B^{s_2}_{z_1+z_2}$	$A^{s_1}_{z_1+z_2}+B^{s_2}$

Table 2: Mapping function transformation for the matrix multiplication operator.
J_X denotes a matrix filled with all ones, with the same shape as matrix X. K is the reduction dimension size of the matrix multiplication.

MatMul	A	A_{z_1}
B	AB	$AB+(J_AB)^{z_1}$
B^{s_2}	$(AB)^{s_2}$	$(AB)^{s_2}+(J_AB)^{s_2z_1}$
B_{z_2}	$AB+(AJ_B)^{z_2}$	$AB+(J_AB)^{z_1}+(AJ_B)^{z_2}+(J_{AB})^{z_1z_2K}$
$B^{s_2}_{z_2}$	$(AB)^{s_2}+(AJ_B)^{s_2z_2}$	$(AB)^{s_2}+(J_AB)^{s_2z_1}+(AJ_B)^{s_2z_2}+(J_{AB})^{s_2z_1z_2K}$

expressions due to the incorporation of reduction along the K dimension. Matrix J_X, which represents a matrix filled with all ones and has the same shape as X, is introduced in the result.

To illustrate the potential benefits brought by these transformations, consider the multiplication of Qtiles A_{z_1} and B_{z_2} as an example. The output is the summation of four Qtiles, as shown in the table. The first component, AB, is a normal tile that does not require dequantization. The second and third components involve matrix multiplications with J matrices. While these components are Qtiles requiring dequantization, the all-one property of J matrices reduces the Qtile shape. Since all rows in J_A matrix are identical, the product J_AB inherits this property. Consequently, the dequantization can be performed solely on the first row of $(J_AB)^{z_1}$, and the resulting values can be broadcast to the remaining rows. As a result, the dequantization overhead is reduced because of the reduction property of J matrices. The final component $(J_AJ_B)^{z_1z_2} = (J_{AB})^{z_1z_2K}$ is a matrix with all identical values, which can be efficiently represented and computed as a single scalar. In summary, the number of computationally expensive dequantization operations is significantly reduced, leading to improved performance.

Additionally, Qtile computation transformation allows for shifting between different quantization formats. In the matrix multiplication of A and $B^{s_2}_{z_2}$, the resulting Qtile expression consists solely of symmetrically quantized Qtiles (without zero points). Since distinct mapping functions may incur varying dequantization overheads, Qtile transformations provide an expanded optimization space by replacing heavy dequantization operations with more lightweight alternatives.

5 Differentiated Qtile Scheduling

After the graph-level Qtile transformation, the generated Qgraphs are executed by inserting dequantization operations. When performing dequantization, Qtiles need to be loaded from the lowest GPU memory to the highest registers. Previous DNN compilers solely focus on the memory access of model weight tensors, overlooking the overhead brought by the associated quantization parameters introduced in quantized models. When it comes to Qtiles, the scheduling needs further consideration to ensure full reuse of quantization parameters.

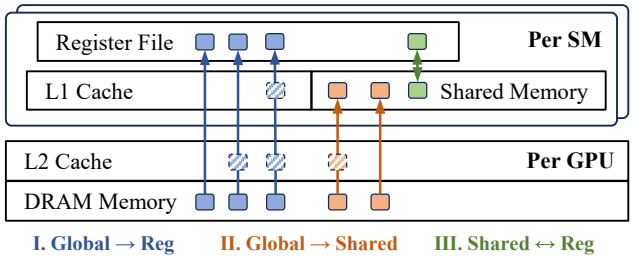


Figure 4: GPU memory hierarchy.

5.1 Diverse Data paths in Memory Hierarchy

Modern GPUs are equipped with hierarchical and multi-path data loading mechanisms in memory system. As illustrated in Figure 4, GPUs typically feature at least four distinct levels of memory hierarchy, ranging from the lowest DRAM memory to the highest register file. Memory resources are allocated across various scopes of processing units, with each layer offering a different visibility. The GPU memory and L2 cache are globally visible and shared across the entire GPU, whereas the upper two layers are specific to each Streaming Multiprocessor (SM). The user-configurable L1 cache, referred to as shared memory, is visible to all threads within a thread block, while registers are private to individual threads.

To perform computations on GPUs, data must be transferred from DRAM to registers, passing through intermediate memory layers. The thread-block-level shared memory allows for explicit control of data at L1 level, creating two categories of data paths, namely global-shared path and shared-register path, as depicted in Figure 4. Other intermediate cache layers remain transparent to programmer. However, CUDA provides a mechanism for controlling the cache eviction policy of certain data through PTX-level attributes called cache operators [27]. Specifically, .cg cache operator allows load instructions to bypass L1 cache, while .cs operator sets an evict-first policy for data in both L1 and L2 caches. This evict-first pol-

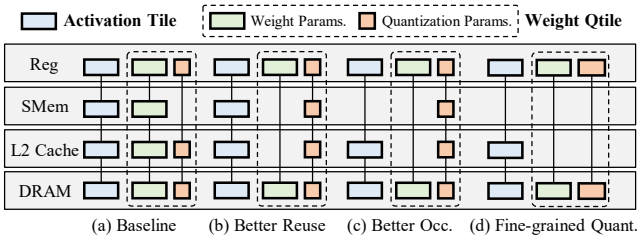


Figure 5: Qtile scheduling examples.

icy prioritizes the eviction of data from cache, minimizing memory pollution in cache, forming a data path bypassing the transparent cache layers. As a consequence, a wider range of distinct data loading paths from global memory to register are explored, as shown in blue lines in Figure 4.

Recent generations of NVIDIA GPUs (since Ampere) have introduced new asynchronous data movement instructions, enabling deeper software pipelining. The Hopper-series GPUs further introduces an independent hardware component specialized for data transfer and address calculation called Tensor Memory Accelerator (TMA). These advanced features are compatible with QFactory and can be passed as device specifications by users, enabling full utilization of these new hardware capabilities.

5.2 Scheduling Qtiles on Memory Hierarchy

With these various data loading paths, QFactory differentiates the mapping of tensor tiles to different data paths for maximizing overall memory utilization, generating strategies tailored to specific scenarios.

To illustrate, we demonstrate the data loading of a non-quantized activation tile and a quantized weight Qtile, which consists of low-precision weight parameters and their associated auxiliary quantization parameters. Figure 5 presents possible Qtile loading schedules discovered by QFactory.

Schedule (a) in Figure 5 represents the static strategy employed by previous compilers. In this approach, activation tensor tiles and weight tensor tiles are cached in shared memory before fetched into registers, enabling data reuse for both tiles. However, this strategy fails to optimize the quantization parameters in Qtiles, as it rigidly executes the dequantization operation, overlooking the overhead introduced by quantization parameters.

Beyond this conventional scheduling, QFactory proposes schedule (b) for better data reuse. By being aware of the group structure within Qtiles, QFactory enables the reuse of quantization parameters with shared memory. Moreover, QFactory applies the evict-first cache operator and bypasses intermediate cache levels for weight tiles. Since the decoding phase involves matrix-vector multiplication, weight values loaded are never reused across computation units. Bypassing intermediate cache levels frees up cache resources, increasing the cache hit rate for other kernel inputs.

In scenarios where GPU occupancy is critical for achieving optimal memory bandwidth utilization, QFactory trades off cache hit rate for improved occupancy, further eliminating caching of activation tiles in shared memory, as shown in schedule (c). This tradeoff is particularly effective on modern GPUs like H100, which provides massive number of LD/ST units for memory access. Allocating excessive shared memory to each thread block can restrict maximum occupancy, leading to underutilization of memory units and a degradation in overall memory bandwidth.

Finally, if an extremely fine-grained quantization algorithm is used or in the circumstance of lower bit-width, one thread may be assigned the workload of computing a whole weight group. In such case, QFactory switches to schedule (d), disabling shared memory usage for quantization parameters and directly reuse them at the register level.

6 Implementation

In this section, we introduce other necessary details of implementing QFactory, including the code generation method and kernel auto-tuning.

6.1 Template-based Kernel Generation

Inspired by CUTLASS [33], QFactory constructs CUDA code templates to implement various data loading paths on GPUs for moving data between memory hierarchy layers. This approach not only facilitates efficient code generation but also promotes automated tuning for performance optimization.

Given that type conversion instructions often have limited computational throughput compared to standard floating-point instructions [26], QFactory incorporates efficient type casting methods from previous works [12, 39]. Specifically, QFactory implement template functions for fast converting different formats of low-bit integers to half-precision real values in a unified manner.

To address the inefficiency caused by unaligned matrix sizes, which is a common issue in quantized kernels due to the increased alignment requirements of lower bit-width data types, QFactory eliminates unnecessary conditional jump instructions through templates. This is achieved by passing matrix shapes and other tuning parameters as compile-time constants, allowing most program constants to be deduced at compile time.

Finally, with the help of CUDA compiler, QFactory enables loop unrolling and instruction reordering to leverage instruction level parallelism (ILP) for overlapping memory overheads. While our approach is orthogonal to techniques such as double buffering and software pipelining, we observe that ILP is already efficient to deliver near-optimal performance in most LLM decoding scenarios.

6.2 ML-based Kernel Selector

As QFactory expands the search space for kernel optimization, efficiently determining the best kernel candidates and other hyperparameters in kernel tiling becomes a critical challenge.

To address this, QFactory employs a machine learning-based approach to quickly identify the optimal configuration of tunable parameters. Rather than directly training a model to output the best configuration, QFactory trains a lightweight multi-layer perceptron (MLP) model to predict the efficiency of given kernel configurations. We choose the achieved memory bandwidth utilization rate as regression goal instead of latency as it provides a limited value range. Kernels with different quantization settings do not share prediction models, but the kernels with the same quantization config for different matrix shapes share the same prediction model. We concatenate input matrix shapes and kernel hyperparameters (e.g., tiling block sizes, threadblock size) into a vector as input data and use the ratio of the kernel's achieved bandwidth to the device's maximum bandwidth as label. Training data is collected through offline profiling, where the real execution latencies of a small subset of kernels with randomly generated configurations are measured.

During the online compilation of a new quantized program, QFactory uses the trained model to estimate the latencies of all possible kernel configurations. QFactory then selects a few kernel configurations with the lowest estimated latencies for code generation and benchmarking. Therefore, a significant portion of inefficient kernel implementations are excluded, significantly accelerating the tuning process.

7 Evaluation

In this section, we present a comprehensive evaluation of QFactory across three different generations of NVIDIA data center GPUs: V100 (PCIe, 32GB), A100 (PCIe, 40GB), and H100 (PCIe, 80GB). We use CUDA 12.4 for evaluation on A100 and H100 GPUs, and CUDA 12.1 for V100 GPUs.

Evaluated Workloads We conduct experiments of two types of workloads to assess the overall performance of QFactory. We first evaluate the performance of the generated quantized kernels across different matrix shapes. These matrix shapes are categorized into two groups: one consisting of square matrices of varying scales, and the other comprising matrix shapes derived from actual LLM models. Next, we integrate QFactory into the popular LLM serving framework vLLM [21] to test the end-to-end model inference speedup. The evaluated LLM models include Llama-2 [35] and Qwen-2.5 [32] series with different scales, and can serve as a representative benchmark for a boarder range of LLMs. All workloads include different quantization bit-widths ranging from 8-bit to 2-bit.

Baselines For kernel-level benchmark, we compare QFactory to BitBLAS [39], which is the state-of-the-art deep learning compiler for low-precision linear kernels. We also compare QFactory to Marlin [12], which is a manually optimized kernel library for 4-bit quantized kernels on NVIDIA Ampere-series GPUs. For end-to-end evaluation, we compare QFactory to BitBLAS, Marlin and llama.cpp [13]. llama.cpp is a popular LLM inference framework with support of multiple quantization bit-widths and formats. For model inference with BitBLAS and Marlin, we leverage their integration with vLLM for fair comparision.

Methodology For kernel analysis, latency is measured with 200 warmup runs and 100 measurement runs. L2 cache is flushed between runs, and timing is recorded using CUDA events. No synchronization is invoked during the 300 runs to exclude CPU launch overhead, which can significantly affect measurements for small kernels. For end-to-end inference, we use the benchmarking scripts of vLLM for fairness. Generation speed is reported in tokens per second.

7.1 Kernel Overall Performance

NVIDIA H100 GPU Figure 6 demonstrates the performance of quantized kernels compiled by QFactory compared to BitBLAS and Marlin on NVIDIA H100 GPU. The evaluated quantization method is asymmetric weight-only quantization, with a group pattern of 128×1 blocks. Since Marlin is originally supported only on Ampere-series GPUs, we made necessary source code modifications for running it on H100.

The figure shows the relative speedup of different quantization bit-widths, including 8-bit (W8), 4-bit (W4), and 2-bit (W2), against the half-precision (W16) cuBLAS kernel as the baseline. The x-axis represents different weight matrix shapes. The shapes from M0 to M2 are square matrices of size 4096, 8192, and 16384. The matrix shapes from M3 to M8 are derived from Llama2-13B and Qwen2.5-72B models. We also demonstrates the geometric mean speedup of all test cases in the figure. The batch size of input activation is set to 1, indicating the common decoding scenario where weight-only quantization could gain most benefits.

As shown in the figure, QFactory consistently outperforms BitBLAS across most quantization configurations and matrix shapes. For quantization bit-width of 8, 4 and 2, QFactory achieves average speedups of $1.17\times$, $1.52\times$, and $1.66\times$, respectively. The performance gap becomes more pronounced at lower bit-width settings due to the changes in data volume of weight parameters. Specifically, as the bit-width decreases, the data volume of weight parameters loaded from memory is reduced, while the data volume of quantization parameters remains unchanged. As BitBLAS is unaware of this change, its kernel is hindered by the overhead associated with loading and processing quantization parameters, leading to significant performance degradation at lower bit-widths.

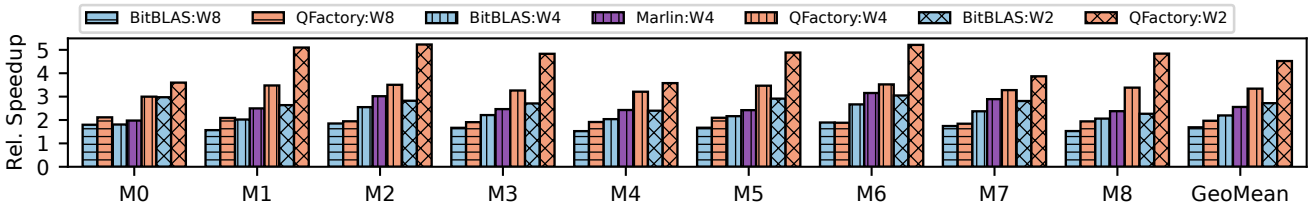


Figure 6: Relative kernel performance compared to FP16 cuBLAS kernel on NVIDIA H100 GPU.

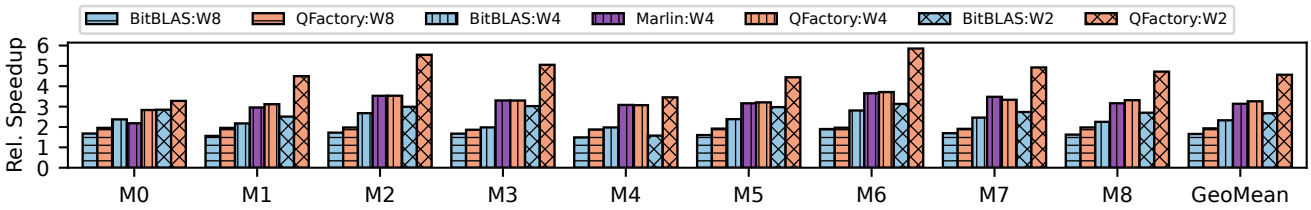


Figure 7: Relative kernel performance compared to FP16 cuBLAS kernel on NVIDIA A100 GPU.

It is noteworthy that QFactory also outperforms the heavily optimized Marlin kernel in 4-bit quantization on H100 GPU by 1.30×. This suggests that simply applying optimizations designed for Ampere-series GPUs results in suboptimal performance on Hopper-series GPUs. To achieve optimal performance across different GPU generations, it is essential to develop general-purpose compilers.

NVIDIA A100 and V100 GPUs We also evaluate kernel performance on NVIDIA A100 and V100 GPUs. Figure 7 and Figure 8 show the relative kernel performance on these GPUs. On A100, QFactory achieves an average speedup over BitBLAS of 1.17× for 8-bit, 1.40× for 4-bit, and 1.71× for 2-bit quantization. The Marlin library performs well on A100, significantly outperforming BitBLAS. Nevertheless, compared to the heavily optimized Marlin library, QFactory still delivers a comparable performance with 1.04× speedup.

On V100, QFactory achieves an average speedup of 0.99× for 8-bit, 1.17× for 4-bit, and 1.41× for 2-bit quantization compared to BitBLAS. Marlin is not supported on V100 due to its reliance on specific hardware characteristics, such as asynchronous data copy instructions, which are unavailable on this older GPU. The relative speedup against cuBLAS shown in Figure 8 appears significantly higher than on other GPUs, likely because the cuBLAS library on V100 has not been fully optimized for certain matrix shapes commonly encountered in recent LLM inference workloads.

7.2 Memory Bandwidth Analysis

Quantized linear kernels are memory-bound at low batch sizes, so achieved memory bandwidth is a strong proxy for kernel efficiency. The bandwidth metric also allows us to better evaluate GPU bandwidth utilization by comparing it with hardware specifications. Below, we conduct an in-depth analysis of the kernel's memory bandwidth utilization by scaling various matrix sizes and different quantization methods.

Scaling Matrix Sizes We evaluate the achieved kernel memory bandwidth utilization of QFactory, BitBLAS, and Marlin in 4-bit quantization on the NVIDIA H100 GPU. To provide a point of comparison, we also include the FP16 cuBLAS kernel. The evaluation uses square matrices of varying sizes. To align with different storage precision between 4-bit weight and 16-bit weight (in cuBLAS), we plot the achieved memory bandwidth as a function of the matrix storage size rather than its shape.

Figure 9 presents the memory bandwidth achieved by the simple-casting W4A16 and asymmetric quantized W4A16 kernels across different matrix sizes. In all cases, the achieved bandwidth increases as the size of the weight matrix grows. This trend can be attributed to the improved utilization of GPU's hardware load units with larger matrices.

The Marlin kernel achieves strong performance, comparable to cuBLAS, with little to no difference observed between simple and asymmetric workloads. This is due to Marlin's well-designed software pipeline, which effectively hides memory loading latencies of both weight parameters and quantization parameters. However, Marlin fails to fully utilize the memory bandwidth of H100 hardware due to its static execution paradigm.

BitBLAS also demonstrates comparable performance in the simple-casting W4A16 kernel but suffers from a significant performance degradation in the asymmetric kernel. This is primarily due to its inability to handle the overhead introduced by additional quantization parameters.

In contrast, QFactory consistently achieves higher bandwidth, particularly for medium-sized weight matrices. Al-

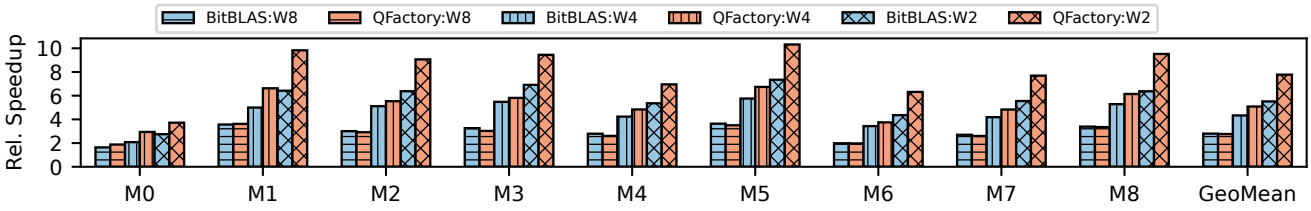


Figure 8: Relative kernel performance compared to FP16 cuBLAS kernel on NVIDIA V100 GPU.

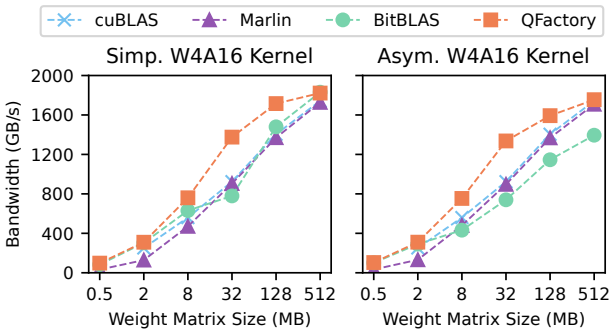


Figure 9: Achieved memory bandwidth on NVIDIA H100 GPU across varying weight matrix sizes for 4-bit kernels. Simp. and Asym. in the figure title stands for Simple-Casting and Asymmetric, respectively.

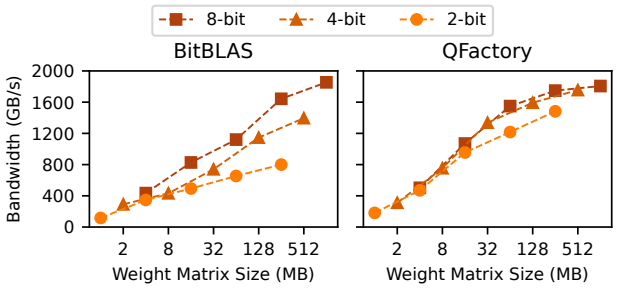


Figure 10: Achieved memory bandwidth on NVIDIA H100 GPU for Asymmetric WxA16 quantized kernels across varying bit-widths.

though the asymmetric kernel achieves slightly lower bandwidth compared to its simple-casting counterpart, QFactory still outperforms other baselines, highlighting its efficiency in managing memory load overhead introduced by quantization parameters.

The speedup advantage of QFactory diminishes with larger matrix sizes as all baseline methods approach the hardware's peak memory bandwidth (approximately 90% of 2TB/s on H100 GPUs). However, extremely large matrices are rarely encountered in practical LLM serving scenarios. For instance, the largest weight matrix in Llama2-70B (found in the gate/up projection layer) measures only 28,672 × 8,192 (112MB in FP16 format). The memory footprint becomes even smaller when employing weight quantization to lower bit-widths.

Consequently, real-world inference workloads primarily operate on medium-sized matrices, where QFactory demonstrates substantially superior performance compared to baseline approaches.

Scaling Bit-Widths We further analyze kernel memory bandwidth by scaling the bit-widths of weight matrices. Figure 10 illustrates the achieved memory bandwidth of QFactory and BitBLAS under 8-bit, 4-bit, and 2-bit asymmetric quantization settings on H100.

As shown in Figure 10, the memory bandwidth utilization for BitBLAS decreases significantly as the bit-width is reduced from 8-bit to 2-bit. This decline is primarily attributed to the increasing dominance of additional quantization parameters, including scaling factors and zero points, which introduce more overhead and reduce effective memory bandwidth utilization in lower-bit-width settings.

In contrast, QFactory demonstrates significantly better scalability across different bit-widths. Although there is a minor reduction in bandwidth when moving from 4-bit to 2-bit quantization, QFactory consistently outperforms BitBLAS across all quantization levels. This advantage becomes particularly pronounced as the matrix size increases, highlighting QFactory's ability to fully utilize H100's ample memory bandwidth by delivering higher occupancy and more efficient resource usage.

7.3 End-to-end Inference

To evaluate the effectiveness of QFactory in accelerating model serving, we conduct end-to-end performance tests using real models and measure the generation speed in tokens per second during the decoding stage. The tests are performed on both 4-bit and 2-bit quantization levels. All models are quantized with GPTQ algorithm [10], which is widely adopted for quantized model deployment. The quantization format follows group size 128 with symmetric quantization, a common configuration in HuggingFace [41] model repositories.

For Marlin, BitBLAS, and QFactory, we use the vLLM framework integration to ensure consistency and fairness. For llama.cpp, which uses a custom GGUF format for quantiza-

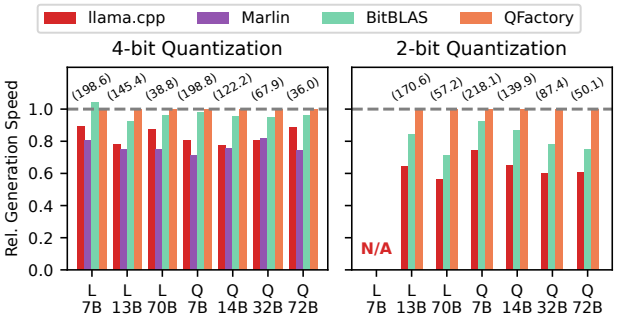


Figure 11: End-to-end model inference performance on NVIDIA H100 GPU. All generation speeds are normalized to QFactory. "L" denotes Llama models, while "Q" denotes Qwen models. The number in parentheses indicates the absolute generation speed of QFactory in tokens per second (tokens/s).

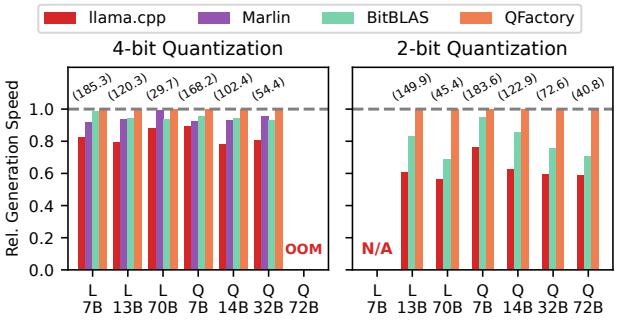


Figure 12: End-to-end model inference performance on NVIDIA A100 GPU.

tion and tensor storage, we apply the Q4_0 format for all linear layers in 4-bit quantization and Q2_K format for 2-bit quantization to ensure comparable computation workloads with GPTQ quantization. The inference tests are conducted with a batch size of 1, a prompt length of 16 tokens, and a generation length of 128 tokens. We use the decoding stage generation speed (tokens/second) as the primary metric. As our optimizations do not involve mathematically non-equivalent transformations, we have verified that QFactory could produces identical model output tokens.

NVIDIA H100 GPU Figure 11 presents the end-to-end inference results on NVIDIA H100 GPU across seven models, including models from Llama-2 and Qwen-2.5 series with different model size ranging from 7B to 72B. All generation speeds are normalized to QFactory for clarity, with absolute generation speeds (in tokens/s) displayed in parentheses for reference.

On 4-bit quantization, QFactory demonstrates robust performance, achieving a geometric average speedup compared to llama.cpp, Marlin, and BitBLAS by 1.21×, 1.32×, and 1.03×, respectively. The speedup is particularly significant for larger models, such as Llama-2-70B and Qwen-2.5-72B, as the relative impact of overheads, such as kernel launches and non-linear-layer operations, is more effectively amortized in larger models.

For 2-bit quantization, Marlin is not supported. Additionally, Llama-2-7B is excluded because its intermediate hidden size (11008) is not divisible by 128 (quantization group size) × 4 (alignment requirement for packing 2-bit data into 1 byte). QFactory exhibits superior performance on 2-bit quantization, with an average speedup of 1.58× and 1.23× compared to llama.cpp and BitBLAS, respectively. Notably, llama.cpp performs poorly in 2-bit quantization, showing no significant latency reduction compared to 4-bit quantization. This is primarily due to the lack of optimizations in its 2-bit kernel implementation.

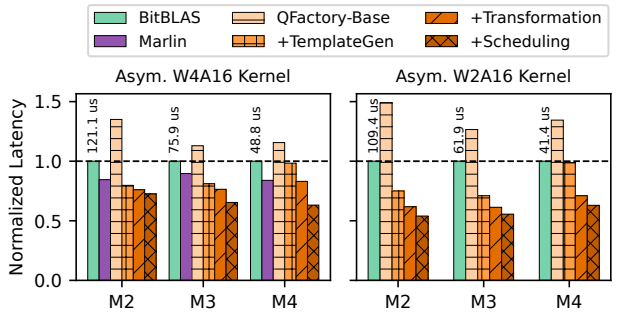


Figure 13: Kernel performance breakdown. All kernel latencies are normalized to BitBLAS. The number above bar shows the absolute latency of BitBLAS kernel in us.

NVIDIA A100 GPU Figure 12 shows the end-to-end inference performance on NVIDIA A100 GPU. Qwen-72B runs out of memory (OOM) on A100 due to its limited 40GB memory capacity. Across the remaining models, QFactory demonstrates consistent speedups, highlighting its adaptability to different GPU architectures.

Specifically, on 4-bit quantization, QFactory achieves average speedups of 1.21×, 1.06×, and 1.05× compared to llama.cpp, Marlin, and BitBLAS, respectively. On 2-bit quantization, QFactory achieves speedups of 1.61× and 1.26× over llama.cpp and BitBLAS, respectively. On A100, Marlin shows significantly better performance than on H100, as it has been fully optimized for this GPU architecture. Nevertheless, QFactory remains competitive, delivering comparable performance to Marlin while maintaining its robust efficiency across all tested models and quantization levels.

Table 3: Average kernel speedup over cuBLAS baseline on varying batch size.

Batch Size	1	2	4
BitBLAS	2.19	1.46	0.87
Marlin	2.56	2.54	**2.53**
QFactory	**3.44**	**3.18**	2.52

7.4 Optimization Breakdown

Figure 13 illustrates the kernel performance as optimization steps are progressively applied during the compilation of quantized kernels. We select three representative matrix shapes from Figure 6 for ablation study. For clarity, all kernel latencies are normalized to BitBLAS, with the absolute latency of BitBLAS indicated above bar.

The optimization process starts with the QFactory-Base implementation. The naive kernel shows degraded performance for 2-bit kernels, primarily due to the increased overhead caused by severe data alignment mismatches inherent in low bit-width computations.

The template-based kernel generation (+TemplateGen) introduced in §6.1 delivers substantial performance improvements. By leveraging instruction-level parallelism to overlap memory access and computation, and aligning memory layouts with hardware cacheline size, QFactory achieves marked reductions in latency. Specifically, QFactory surpasses all baselines in M2 and M3, while achieving comparable performance to BitBLAS on M4.

The Qtile computation transformation (+Transformation, proposed in §4) and Qtile scheduling (+Scheduling, introduced in §5) further unlock additional performance gains by exploring a broader range of transformation spaces. Together, these techniques enable QFactory to achieve better overall performance than all baselines across the evaluated matrix configurations.

7.5 Scaling Batch Size

Table 3 compares the geometric average speedup of asymmetric W4A16 kernel over the FP16 cuBLAS baseline across nine selected kernels same with Figure 6 on H100 GPU.

All baselines achieve highest speedup at batch size 1, as weight-only quantization provides the greatest benefit in memory-bound scenarios. From Table 3, QFactory consistently outperforms BitBLAS across all batch sizes but is surpassed by Marlin at larger batch sizes. This is because Marlin leverages manually optimized MMA instructions to take advantage of Tensor Core capabilities in high-batch-size scenarios. However, this optimization is orthogonal to QFactory and can be integrated with additional engineering efforts.

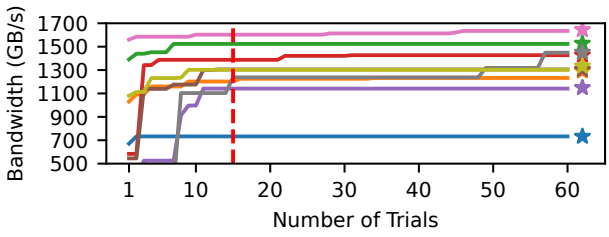


Figure 14: Prediction performance of kernel selector.

7.6 Kernel Selector

In the offline profiling stage of auto-tuning, QFactory randomly chooses 50 configurations for each kernel and collect data through profiling. The trained model is than utilized for prediction. We use the same nine matrices from §7.1 for evaluating the effectiveness of kernel selector. QFactory enumerates candidate kernels in the order of shorter predicted execution time and selects best kernel configurations by benchmarking. Figure 14 illustrates the achieved memory bandwidth as a function of the number of trials. The star markers in the figure represent the optimal kernel performance of each matrix shape in the whole search space. It is shown that the optimal kernel configuration can be identified within 15 trials in most matrix shape cases.

Since kernel latency is measured in microseconds, the tuning overhead primarily comes from compilation. Compared to evaluating the full search space, which could contain thousands of candidate kernels, testing just 15 candidates achieves a practical balance between tuning time and performance.

Users can increase the number of trials or conduct exhaustive offline profiling to obtain better kernel performance. In our experiments, full parameter enumeration takes a few hours for a single LLM model. As the results are reusable, even exhaustive tuning is feasible for production deployment.

8 Related Work

Quantization of LLMs Quantization methods for LLMs can be broadly categorized into Quantization-Aware Training (QAT) and Post-Training Quantization (PTQ), both aiming to reduce the bit-width of model weights while maintaining performance.

QAT integrates quantization into the training process, requiring retraining or fine-tuning of the model. While this incurs significant computational and memory overhead, it enables much lower bit-widths, with some methods achieving extreme cases such as 1-bit quantization. Examples of QAT methods include LLM-QAT [24], QLoRA [7], BitNet [38], BitDistiller [9], and OneBit [44].

In contrast, PTQ avoids the need for retraining by directly quantizing pre-trained weights, often using a calibra-

tion dataset to guide the quantization process. Due to its efficiency, 4-bit PTQ methods including GPTQ [10], AWQ [23] and SpQR [8] have gained widespread adoption, particularly in resource-constrained scenarios. Recent works, including QuIP [3], SqueezeLLM [19], and OmniQuant [30], have extended PTQ to explore even lower bit-widths.

System for Deploying Quantized LLMs Systems such as bitsandbytes [5], TensorRT [28], llama.cpp [13], vLLM [21], and Marlin [12] provide fast kernels for popular quantization formats. However, these systems are limited in application scope, supporting only a few specific cases, and often exhibit less portable performance across different hardware devices. Consequently, researchers frequently resort to manually implementing custom kernels for advanced quantization formats, such as those used in AWQ [23], FP6-LLM [42], and SqueezeLLM [19]. LUT-GEMM [29], FLUTE [15], and T-MAC [40] focus on optimizing lookup table based quantization, which is particularly useful in low-bit-width scenarios. BitBLAS [39], which builds on normal-precision deep learning compilers including TVM [4], Roller [46], and Welder [31], provides support for low-precision quantization. Nonetheless, it does not adequately address the complexities of advanced quantization formats or group-wise quantization, limiting its effectiveness in more sophisticated quantization workflows.

9 Limitation

As batch size increases, weight-only quantization fails to consistently deliver theoretical speedups because the kernel shifts from memory-bound to compute-bound, diminishing the benefits of reduced memory footprint. To address this limitation, joint activation-weight quantization introduces additional compression for activations and employs low-precision computation, enabling sustained speedups even at large batch sizes.

While QFactory currently supports only weight-only quantization, its core methodology can be extended to activation-weight quantization for further kernel acceleration. We leave this direction as future work.

10 Conclusion

We present QFactory, an efficient compilation framework for accelerating quantized LLM serving. QFactory addresses the inefficiency of existing systems by introducing a novel Qtile abstraction, enabling the exploration of a broader optimization space through Qtile computation transformations and differentiated memory scheduling. Extensive evaluations across a range of GPUs demonstrate that QFactory outperforms state-of-the-art compilers and manually optimized libraries. As a scalable solution for deploying quantized LLMs, QFactory not only enhances model efficiency but also facilitates the exploration of new quantization algorithms by alleviating the significant human effort required for custom kernel implementation.

Acknowledgments

We would like to thank the anonymous reviewers and our shepherd Irfan Ahmad for their insightful comments. This work is supported by the National Key R&D Program of China under Grant 2023YFB3002002, NSFC of Distinguished Young Scholar (62225206), Beijing Natural Science Foundation (L242017), and Tsinghua University Initiative Scientific Research Program. Jidong Zhai (zhaijidong@tsinghua.edu.cn) is the corresponding author of this paper.

References

[1] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin, editors, *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020.

[2] Sébastien Bubeck, Varun Chandrasekaran, Ronen Eldan, Johannes Gehrke, Eric Horvitz, Ece Kamar, Peter Lee, Yin Tat Lee, Yuanzhi Li, Scott M. Lundberg, Harsha Nori, Hamid Palangi, Marco Túlio Ribeiro, and Yi Zhang. Sparks of artificial general intelligence: Early experiments with GPT-4. *CoRR*, abs/2303.12712, 2023.

[3] Jerry Chee, Yaohui Cai, Volodymyr Kuleshov, and Christopher De Sa. Quip: 2-bit quantization of large language models with guarantees. In Alice Oh, Tristan Naumann, Amir Globerson, Kate Saenko, Moritz Hardt, and Sergey Levine, editors, *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*, 2023.

[4] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Q. Yan, Haichen Shen, Meghan Cowan,

Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: an automated end-to-end optimizing compiler for deep learning. In Andrea C. Arpaci-Dusseau and Geoff Voelker, editors, *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*, pages 578–594. USENIX Association, 2018.

[5] Tim Dettmers. bitsandbytes: 8-bit optimizers and quantization routines, 2022.

[6] Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. Llm.int8(): 8-bit matrix multiplication for transformers at scale. *CoRR*, abs/2208.07339, 2022.

[7] Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. Qlora: Efficient finetuning of quantized llms. In Alice Oh, Tristan Naumann, Amir Globerson, Kate Saenko, Moritz Hardt, and Sergey Levine, editors, *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*, 2023.

[8] Tim Dettmers, Ruslan Svirschevski, Vage Egiazarian, Denis Kuznedelev, Elias Frantar, Saleh Ashkboos, Alexander Borzunov, Torsten Hoefler, and Dan Alistarh. Spqr: A sparse-quantized representation for near-lossless LLM weight compression. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net, 2024.

[9] Dayou Du, Yijia Zhang, Shijie Cao, Jiaqi Guo, Ting Cao, Xiaowen Chu, and Ningyi Xu. Bitdistiller: Unleashing the potential of sub-4-bit llms via self-distillation. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar, editors, *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2024, Bangkok, Thailand, August 11-16, 2024*, pages 102–116. Association for Computational Linguistics, 2024.

[10] Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. GPTQ: accurate post-training quantization for generative pre-trained transformers. *CoRR*, abs/2210.17323, 2022.

[11] Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. OPTQ: accurate quantization for generative pre-trained transformers. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net, 2023.

[12] Elias Frantar, Roberto L Castro, Jiale Chen, Torsten Hoefler, and Dan Alistarh. Marlin: Mixed-precision auto-regressive parallel inference on large language models. *arXiv preprint arXiv:2408.11743*, 2024.

[13] Georgi Gerganov. llama.cpp: Llm inference in c/c++. https://github.com/ggerganov/llama.cpp, 2023.

[14] Amir Gholami, Sehoon Kim, Zhen Dong, Zhewei Yao, Michael W. Mahoney, and Kurt Keutzer. A survey of quantization methods for efficient neural network inference. *CoRR*, abs/2103.13630, 2021.

[15] Han Guo, William Brandon, Radostin Cholakov, Jonathan Ragan-Kelley, Eric P. Xing, and Yoon Kim. Fast matrix multiplications for lookup table-quantized llms. In Yaser Al-Onaizan, Mohit Bansal, and Yun-Nung Chen, editors, *Findings of the Association for Computational Linguistics: EMNLP 2024, Miami, Florida, USA, November 12-16, 2024*, pages 12419–12433. Association for Computational Linguistics, 2024.

[16] Muyan Hu, Ashwin Venkatram, Shreyashri Biswas, Balamurugan Marimuthu, Bohan Hou, Gabriele Oliaro, Haojie Wang, Liyan Zheng, Xupeng Miao, Jidong Zhai, and Zhihao Jia. Optimal kernel orchestration for tensor programs with korch. In Rajiv Gupta, Nael B. Abu-Ghazaleh, Madan Musuvathi, and Dan Tsafrir, editors, *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3, ASPLOS 2024, La Jolla, CA, USA, 27 April 2024- 1 May 2024*, pages 755–769. ACM, 2024.

[17] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew G. Howard, Hartwig Adam, and Dmitry Kalenichenko. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *2018 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2018, Salt Lake City, UT, USA, June 18-22, 2018*, pages 2704–2713. Computer Vision Foundation / IEEE Computer Society, 2018.

[18] Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. TASO: optimizing deep learning computation with automatic generation of graph substitutions. In Tim Brecht and Carey Williamson, editors, *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019*, pages 47–62. ACM, 2019.

[19] Sehoon Kim, Coleman Hooper, Amir Gholami, Zhen Dong, Xiuyu Li, Sheng Shen, Michael W. Mahoney, and Kurt Keutzer. Squeezellm: Dense-and-sparse quantization. In *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*. OpenReview.net, 2024.

[20] Sehoon Kim, Coleman Hooper, Thanakul Wattanawong, Minwoo Kang, Ruohan Yan, Hasan Genc, Grace Dinh, Qijing Huang, Kurt Keutzer, Michael W. Mahoney, Yakun Sophia Shao, and Amir Gholami. Full stack optimization of transformer inference: a survey. *CoRR*, abs/2302.14017, 2023.

[21] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*, 2023.

[22] Shiyao Li, Xuefei Ning, Luning Wang, Tengxuan Liu, Xiangsheng Shi, Shengen Yan, Guohao Dai, Huazhong Yang, and Yu Wang. Evaluating quantized large language models. In *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*. OpenReview.net, 2024.

[23] Ji Lin, Jiaming Tang, Haotian Tang, Shang Yang, Wei-Ming Chen, Wei-Chen Wang, Guangxuan Xiao, Xingyu Dang, Chuang Gan, and Song Han. AWQ: activation-aware weight quantization for on-device LLM compression and acceleration. In Phillip B. Gibbons, Gennady Pekhimenko, and Christopher De Sa, editors, *Proceedings of the Seventh Annual Conference on Machine Learning and Systems, MLSys 2024, Santa Clara, CA, USA, May 13-16, 2024*. mlsys.org, 2024.

[24] Zechun Liu, Barlas Oguz, Changsheng Zhao, Ernie Chang, Pierre Stock, Yashar Mehdad, Yangyang Shi, Raghuraman Krishnamoorthi, and Vikas Chandra. LLM-QAT: data-free quantization aware training for large language models. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar, editors, *Findings of the Association for Computational Linguistics, ACL 2024, Bangkok, Thailand and virtual meeting, August 11-16, 2024*, pages 467–484. Association for Computational Linguistics, 2024.

[25] Wei Niu, Jiexiong Guan, Yanzhi Wang, Gagan Agrawal, and Bin Ren. Dnnfusion: accelerating deep neural networks execution with advanced operator fusion. In Stephen N. Freund and Eran Yahav, editors, *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, pages 883–898. ACM, 2021.

[26] NVIDIA. *CUDA C++ Programming Guide*, 2024.

[27] NVIDIA. *Parallel Thread Execution ISA Version 8.5*, 2024.

[28] NVIDIA Corporation. *NVIDIA TensorRT: High Performance Deep Learning Inference Platform*, 2023.

[29] Gunho Park, Baeseong Park, Minsub Kim, Sungjae Lee, Jeonghoon Kim, Beomseok Kwon, Se Jung Kwon, Byeongwook Kim, Youngjoo Lee, and Dongsoo Lee. LUT-GEMM: quantized matrix multiplication based on luts for efficient inference in large-scale generative language models. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net, 2024.

[30] Wenqi Shao, Mengzhao Chen, Zhaoyang Zhang, Peng Xu, Lirui Zhao, Zhiqian Li, Kaipeng Zhang, Peng Gao, Yu Qiao, and Ping Luo. Omniquant: Omnidirectionally calibrated quantization for large language models. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net, 2024.

[31] Yining Shi, Zhi Yang, Jilong Xue, Lingxiao Ma, Yuqing Xia, Ziming Miao, Yuxiao Guo, Fan Yang, and Lidong Zhou. Welder: Scheduling deep learning memory access via tile-graph. In Roxana Geambasu and Ed Nightingale, editors, *17th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2023, Boston, MA, USA, July 10-12, 2023*, pages 701–718. USENIX Association, 2023.

[32] Qwen Team. Qwen2.5: A party of foundation models, September 2024.

[33] Vijay Thakkar, Pradeep Ramani, Cris Cecka, Aniket Shivam, Honghao Lu, Ethan Yan, Jack Kosaian, Mark Hoemmen, Haicheng Wu, Andrew Kerr, Matt Nicely, Duane Merrill, Dustyn Blasig, Fengqi Qiao, Piotr Majcher, Paul Springer, Markus Hohnerbach, Jin Wang, and Manish Gupta. CUTLASS, January 2023.

[34] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurélien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. Llama: Open and efficient foundation language models. *CoRR*, abs/2302.13971, 2023.

[35] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton-Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton,

Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurélien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. Llama 2: Open foundation and fine-tuned chat models. *CoRR*, abs/2307.09288, 2023.

[36] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pages 5998–6008, 2017.

[37] Haojie Wang, Jidong Zhai, Mingyu Gao, Zixuan Ma, Shizhi Tang, Liyan Zheng, Yuanzhi Li, Kaiyuan Rong, Yuanyong Chen, and Zhihao Jia. PET: optimizing tensor programs with partially equivalent transformations and automated corrections. In Angela Demke Brown and Jay R. Lorch, editors, *15th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2021, July 14-16, 2021*, pages 37–54. USENIX Association, 2021.

[38] Hongyu Wang, Shuming Ma, Li Dong, Shaohan Huang, Huaijie Wang, Lingxiao Ma, Fan Yang, Ruiping Wang, Yi Wu, and Furu Wei. Bitnet: Scaling 1-bit transformers for large language models. *CoRR*, abs/2310.11453, 2023.

[39] Lei Wang, Lingxiao Ma, Shijie Cao, Quanlu Zhang, Jilong Xue, Yining Shi, Ningxin Zheng, Ziming Miao, Fan Yang, Ting Cao, Yuqing Yang, and Mao Yang. Ladder: Enabling efficient low-precision deep learning computing through hardware-aware tensor transformation. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 307–323, Santa Clara, CA, July 2024. USENIX Association.

[40] Jianyu Wei, Shijie Cao, Ting Cao, Lingxiao Ma, Lei Wang, Yanyong Zhang, and Mao Yang. T-MAC: CPU renaissance via table lookup for low-bit LLM deployment on edge. *CoRR*, abs/2407.00088, 2024.

[41] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, and Jamie Brew. Huggingface's transformers: State-of-the-art natural language processing. *CoRR*, abs/1910.03771, 2019.

[42] Haojun Xia, Zhen Zheng, Xiaoxia Wu, Shiyang Chen, Zhewei Yao, Stephen Youn, Arash Bakhtiari, Michael Wyatt, Donglin Zhuang, Zhongzhu Zhou, Olatunji Ruwase, Yuxiong He, and Shuaiwen Leon Song. Quant-llm: Accelerating the serving of large language models via fp6-centric algorithm-system co-design on modern gpus. In Saurabh Bagchi and Yiying Zhang, editors, *Proceedings of the 2024 USENIX Annual Technical Conference, USENIX ATC 2024, Santa Clara, CA, USA, July 10-12, 2024*, pages 699–713. USENIX Association, 2024.

[43] Guangxuan Xiao, Ji Lin, Mickaël Seznec, Hao Wu, Julien Demouth, and Song Han. Smoothquant: Accurate and efficient post-training quantization for large language models. In Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett, editors, *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA*, volume 202 of *Proceedings of Machine Learning Research*, pages 38087–38099. PMLR, 2023.

[44] Yuzhuang Xu, Xu Han, Zonghan Yang, Shuo Wang, Qingfu Zhu, Zhiyuan Liu, Weidong Liu, and Wanxiang Che. Onebit: Towards extremely low-bit large language models. *CoRR*, abs/2402.11295, 2024.

[45] Liyan Zheng, Haojie Wang, Jidong Zhai, Muyan Hu, Zixuan Ma, Tuowei Wang, Shuhong Huang, Xupeng Miao, Shizhi Tang, Kezhao Huang, and Zhihao Jia. EINNET: optimizing tensor programs with derivation-based transformations. In Roxana Geambasu and Ed Nightingale, editors, *17th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2023, Boston, MA, USA, July 10-12, 2023*, pages 739–755. USENIX Association, 2023.

[46] Hongyu Zhu, Ruofan Wu, Yijia Diao, Shanbin Ke, Haoyu Li, Chen Zhang, Jilong Xue, Lingxiao Ma, Yuqing Xia, Wei Cui, Fan Yang, Mao Yang, Lidong Zhou, Asaf Cidon, and Gennady Pekhimenko. ROLLER: fast and efficient tensor compilation for deep learning. In Marcos K. Aguilera and Hakim Weatherspoon, editors, *16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2022, Carlsbad, CA, USA, July 11-13, 2022*, pages 233–248. USENIX Association, 2022.

[47] Xunyu Zhu, Jian Li, Yong Liu, Can Ma, and Weiping Wang. A survey on model compression for large language models. *CoRR*, abs/2308.07633, 2023.

PluS: Highly Efficient and Expandable ML Compiler with Pluggable Graph Schedules

Ruofan Wu[§*1], Zhen Zheng[†], Feng Zhang[§], Chuanjie Liu[†], Zaifeng Pan[§*2], Jidong Zhai[‡], Xiaoyong Du[§]
[§]*Renmin University of China* [†]*Microsoft* [‡]*Tsinghua University*

Abstract

Machine learning (ML) compilers are effective solutions for deploying diverse Deep Neural Network (DNN) workloads on various hardware platforms automatically. However, there is a notable lag in existing ML compilers when it comes to supporting emerging optimization techniques like recent attention optimizations. These compilers lack the requisite flexibility to support expert-driven subgraph optimizations timely, resulting in suboptimal performance compared to manually optimized libraries. Conversely, template-based compilers lack the ability to abstractly express subgraphs, thereby reducing their adaptability to subtle changes in model architectures.

In this paper, we present PluS, an end-to-end ML compiler that facilitates the deployment of expert-optimized subgraph implementations while still preserving compiler flexibility. We rethink the encapsulation of ML compiler and decouple the burdensome embedded graph transformation process. PluS provides a lightweight loop-centric subgraph abstraction for experts to manage a flexible pattern warehouse, and employs a pattern identification approach for subgraph generation. As a result, PluS can deploy efficient subgraph implementations with minimal manual efforts, making it outperform the state-of-the-art rule-based embedded compilers (up to 4.04× speedup) on popular ML models.

1 Introduction

Deep neural networks (DNNs) have achieved remarkable success in various ML tasks [16, 25, 29, 33, 34, 41, 50]. In recent years, key DNN architectures have exhibited the trend of convergence. For instance, notably, the attention mechanism [43] has become a staple in state-of-the-art DNN designs. Even so, they continue to evolve with the introduction of novel local changes over time, e.g., RMSNorm normalizing function [48] and SwiGLU activation function [37]. Besides the above trends of model evolving, new tensor graph optimization techniques emerge rapidly, e.g., FlashAttention [13, 14] and fused Matmul-LayerNorm-Matmul [40]. The partially convergent model architectures and the rapidly evolved graph optimizations lead to a new demand for ML compilers.

ML compilers [10, 22, 30, 36, 57] play a crucial role in generating efficient kernel implementations for deploying ML models on various hardware platforms automatically. Typically, ML compilers translate ML models into computational graphs of operators and leverage graph transformation techniques like operator fusion to enhance performance. However, many ML compilers [22, 30, 36, 57] rely on predefined rules hard-written in the compilers for graph transformation. This approach cannot meet the growing demands of quickly supporting emerging novel and high-performance optimization techniques, as adjusting the compilers often requires significant effort. Consequently, ML compilers often lag behind the state-of-the-art performance due to the need for heavy modifications to predefined rules and code bases.

Simultaneously, template-based methods have been proposed to optimize computational graphs by matching predefined templates with specific operator compositions. Compared with the compilers with hard-written predefined rules, these methods allow users to support emerging optimizations more easily, without the need to modify the compiler implementation itself. However, this approach often struggles to support diverse model structures. For instance, the current AITemplate [45] codebase fails to support models like T5 due to the absence of fused `T5LayerNorm` composition, even though the `T5LayerNorm` consists of very basic operators. To support this model, users need to define the `T5LayerNorm` graph pattern and code generation (codegen) template.

Given the limitations of existing works, ML compilers face significant demands and challenges in swiftly supporting emerging graph optimization technologies, especially as ML model architectures exhibit a convergent yet evolving trend.

In this work, we introduce PluS, a highly efficient and

*Work was done when Ruofan and Zaifeng interned at Microsoft, advised by Zhen.
[1]Currently at the University of Michigan.
[2]Currently at the University of California, San Diego.

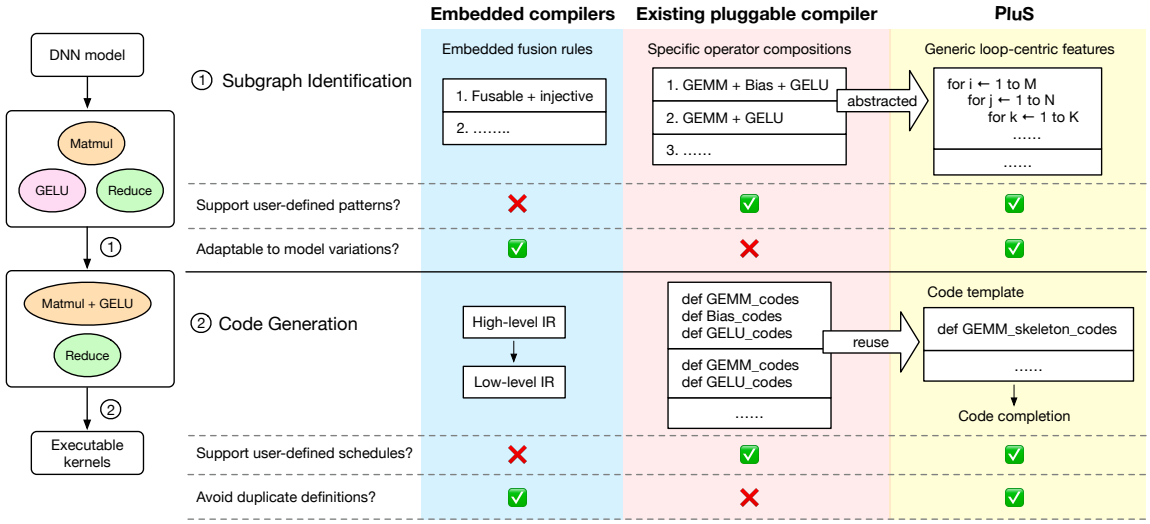


Figure 1: Workflow of ML compilers and a comparison between embedded compilers, existing pluggable compilers, and PluS.

expandable ML optimizing compiler with pluggable graph schedules. PluS decouples the graph transformation and codegen schedule definition from the compiler implementation itself based on the following insights. First, rather than relying on cumbersome graph transformation rules hard-written in the compiler itself, PluS offers a pluggable graph transformation module, which allows domain experts to maintain a warehouse of subgraph patterns. This approach facilitates the seamless deployment of emerging high-performance optimization techniques. Second, PluS recognizes that the optimized schedules for a subgraph are usually determined by the loop structure of key operators, such as `MatMul` and `Reduce`. For instance, substituting an `Add` operator with a `Sub` operator does not alter the loop structure of a graph, making it unnecessary to regenerate the entire schedule for such modifications. Therefore, instead of matching the exact graph of operators, PluS employs a loop-centric pattern matching method for graph transformation, wherein a pattern represents operators or subgraphs with similar underlying loop-centric features.

Building upon these insights, PluS proposes *+Graph*, a lightweight subgraph abstraction based on the loop-centric feature (named *+Loop*) (Sec. 4.1). PluS employs a novel graph transformation approach by generically matching a set of predefined subgraph patterns (with *+Graph*) from the expert-driven pattern warehouse (Sec. 4.2). Subsequently, PluS generates code for the subgraph pattern using the predefined code template (Sec. 4.3).

We have implemented PluS on top of PyTorch and conducted evaluations using five mainstream ML models on NVIDIA A100 GPU and RTX 4090 GPU. In comparison with TorchInductor [8] and TensorRT [22], PluS demonstrates an average end-to-end performance improvement of 4.04× and 1.77×, respectively. These performance gains are attributed to PluS's ability to integrate high-performance subgraph optimization techniques from various vendors and select optimal schedules for different shapes. When compared to the template-based ML compiler AITemplate [45], PluS excels in supporting newly developed DNN architectures while still maintaining a slight performance advantage (approximately 7.8%) on those AITemplate-supported models.

In summary, we make the following contributions:

► We pioneer the concept of pluggable graph schedules, enabling the effortless integration of expert-driven subgraph optimizations within an ML compiler. This innovation facilitates the decoupling of subgraph code generation iteration from the burdensome compiler infrastructure.

► We introduce a novel subgraph abstraction based on the underlying loop-centric features, providing a robust representation of subgraph patterns. Leveraging this abstraction, we devise a novel pattern-matching strategy for graph optimization.

► We develop PluS, an ML compiler supporting the pluggable graph schedule definition, and establish a subgraph pattern warehouse leveraging optimization schedules from state-of-the-art vendors. Extensive evaluation demonstrates the efficacy, portability, and flexibility of PluS.

2 Background and Motivation

2.1 Embedded and Pluggable Compilers

Given a DNN model, an ML compiler typically lowers it into a computation graph composed of a set of fine-grained tensor operators. The common workflow of ML compilers usually involves two key modules: subgraph identification and code generation (codegen), as shown in the left half of Fig. 1. The subgraph identification is to identify subgraphs within the computation graph that can be further optimized. For example, it identifies groups of operators that can be fused into a single subgraph, allowing for the generation of higher-performance code. Codegen is responsible for generating

codegen schedules for the identified subgraphs, which define how to map data computation onto hardware [56].

We categorize the optimizing compilers into two types: embedded and pluggable, based on whether the subgraph identification process supports user-defined functions or not. Embedded compilers implement the subgraph identification within the compiler so that users cannot customize this behavior. In contrast, pluggable compilers empower users to customize the process without modifying the compiler itself. Fig. 1 shows the characteristics of the embedded compilers and the existing pluggable compilers.

Embedded mode. Many of the ML compilers adopt embedded mode, such as XLA [36], TorchInductor [8], BladeDISC [56], DNNFusion [31] and TensorRT [22]. These compilers utilize predefined fusion rules to generate subgraphs. Users cannot modify the fusion rules without changing the compiler implementation. For example, TVM [10] has a set of embedded fusion rules not customizable by users. It categorizes operators into complex-out-fusable, injective, and reduction, and proposes three fusion rules: injective can fuse after complex-out-fusable, injective can fuse before reduction, and injective can fuse with injective. After generating the subgraphs, these compilers lower the high-level graph representations into target-specific code using a low-level intermediate representation (IR). For example, BladeDISC [56] employs the MLIR dialects [26], which encodes computations as loop constructs that map closely to the underlying hardware.

However, embedded compilers are not friendly to adopting the emerging subgraph optimizations. This stems from the complete reliance on compiler predefined rules for graph transformation, making it challenging to harness the innovations of the new optimizations fast. To catch up the new optimizations, it usually requires to modify the cumbersome graph transformation module in the compiler, demanding substantial effort and time.

We utilize the subgraph and the corresponding optimized kernels depicted in Fig. 2 to elucidate the limitations of embedded graph transformation. This subgraph comprises a matrix multiplication (Matmul) followed by a softmax function (Softmax), and the Softmax is decomposed into several basic operators by the ML compiler. Existing manually-written libraries (CUTLASS [40]) can optimize this pattern into 3 kernels as shown in Fig. 2. However, embedded-mode ML compilers face challenges in leveraging these optimizations. First, it requires modifying the fusion and codegen rules to support such subgraphs, which is usually labor-intensive and cannot be timely. Second, given that the high-level operators (Softmax) are usually decomposed into smaller and basic operators in the modern ML compilers, it is hard for the existing compilers to match the intricate subgraph adequately based on simplistic rules.

Pluggable mode. Some tensor compilers allow users to customize the subgraph identification. Users do not need to rewrite the compiler itself to add new fusion or codegen rules. For example, AITemplate [45] employs operator-level subgraph matching to optimize ML models. Users can define desired subgraphs during model definition, and AITemplate identifies specific operator-composed subgraphs for code generation. For instance, AITemplate currently supports 18 types of matrix multiplication-based subgraphs, such as `gemm_rcr_bias_gelu`. This particular subgraph represents a General Matrix Multiply (GEMM) with inputs in row major and column major, followed by Bias and GELU operations, with the output in row major.

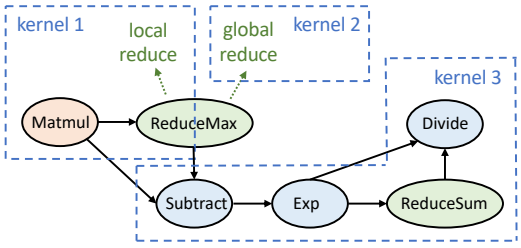


Figure 2: A subgraph formed by basic operators consisting of matrix multiplication followed by the softmax function, and the corresponding kernel implementation in CUTLASS.

However, the existing pluggable mode ML compiler relies on operator-level pattern matching to identify subgraphs and lacks proficiency in handling subtle changes in model architectures. Users are required to provide corresponding implementations for the specific operator composition and specify it during the model definition. For instance, in the case of AITemplate, it currently supports `gemm_rcr_bias_gelu` but not `gemm_rcr_gelu`. This implies that if a user opts to exclude the Bias operation, a new backend template and frontend definition must be added to AITemplate. Furthermore, AITemplate lacks the design for compiling models with dynamic shapes, limiting its flexibility in real-world scenarios. These shortcomings significantly reduce adaptability of pluggable ML compiler and hinders its ability to generalize for broader use cases.

2.2 Opportunities and Insights

We recognize the pressing need for ML compilers to alleviate manual burdens and overcome the inflexibility of embedded graph transformations. In this work, we introduce the concept of pluggable graph transformation to decouple the intricate graph transformation process from the ML compiler implementation. By providing a pluggable and flexible approach to subgraph identification and optimization, we empower users to define graph transformation patterns flexibly.

Given the limitations of existing pluggable compilers that rely on operator-level pattern matching for subgraph optimization, our insight is to use a general pattern with fundamental characteristics to map different operator compositions into the same underlying codegen schedule. We address several key issues with PluS in Sec. 4.1, Sec. 4.2 and Sec. 4.3 respectively:

► How is the general pattern defined, and how does it express the fundamental characteristics of a subgraph?

► How does PluS match or identify subgraph patterns?

► How do users provide codegen schedules for subgraph patterns, and how does PluS generate code accordingly?

First, operators and subgraphs can be expressed through only the loop skeleton of several key operators, which dominate the codegen schedules [56, 57]. PluS introduces a lightweight subgraph abstraction, +*Graph*, to define subgraph patterns with the loop-centric characteristics of key operators while ignoring trivial operators.

Unlike low-level loop abstractions designed for code generation like MLIR dialects [26], which are complex and generally inaccessible to ML optimization experts, +*Graph* is specifically designed for subgraph pattern matching, offering users the concise and necessary representation of a subgraph's essential structure.

Second, PluS employs a novel subgraph pattern identification method based on +*Graph*, rather than relying on fusion rules or matching specific operator compositions. PluS starts identification from the key operators and iteratively matches the subgraphs with user-predefined subgraph patterns.

Compared to the operator-by-operator matching approach employed by AITemplate [45], PluS's subgraph identification algorithm applies a generic matching on subgraph loop skeletons, which allows subgraphs with the same underlying loop-centric features but different operator compositions to be matched to the same +*Graph*.

Third, PluS offers an interface called +*Code* for users to provide the codegen template for any +*Graph*. The code templates define the schedules for the loop skeleton of the key operators, and PluS generates code for other trivial operations in the subgraph according to the data and computation information maintained by the compiler.

With this mechanism, PluS can reuse the codegen schedules for subgraphs with identical +*Graph*, eliminating the need to repeatedly define code implementations for every operator composition like in AITemplate (Fig. 1).

3 System Overview

Building upon the motivation and insights elaborated in Sec. 2, we introduce PluS to address the challenge of flexible subgraph optimization integration with a pluggable graph transformation solution based on loop-centric pattern matching. The core insight is to map different subgraphs with the same loop skeleton to a unique identifier and share the same codegen schedules for these subgraphs. PluS first designs +*Graph* (Sec. 4.1) to serve as an identifier for subgraph pattern matching, which encapsulates the loop-centric characteristics of subgraphs. PluS develops a pattern warehouse to decouple the subgraph identification and codegen process from compiler internals. Users are empowered to manage these processes by providing mappings of +*Graph*s to their corresponding

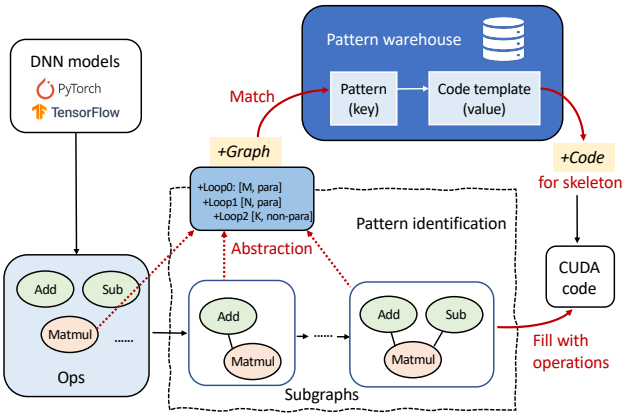


Figure 3: System Overview of PluS.

code templates using PluS's interface +*Code* (Sec. 4.3). The code template defines the codegen schedules for the skeleton operations in the subgraphs.

Figure 3 shows the system overview of PluS. To begin the compilation process, PluS ingests a DNN model from the framework (e.g., PyTorch). It then automatically translates the model into a computational graph comprised of operators. Each operator is essentially expressed as a +*Loop* (Sec. 4.1), which serves as the building block of +*Graph*. Subsequently, PluS employs a pattern identification methodology to identify subgraphs that match the pre-defined subgraph patterns in the pattern warehouse, as detailed in Sec. 4.2. Specifically, PluS adopts a greedy extension policy to expand the subgraph scope, obtaining the +*Graph*s of subgraphs and iteratively querying the pattern warehouse to determine whether the +*Graph* matches the pre-defined patterns. Finally, PluS generates code for the identified subgraphs using the code templates in the pattern warehouse and specific information within each subgraph, filling in the trivial operations, as detailed in Sec. 4.3.

4 Design Methodology

4.1 Subgraph Identifier: +Graph

In this section, we introduce +*Graph*, a subgraph representation designed to serve as an identifier in the subgraph identification process. +*Graph* abstracts DNN computation to its loop structure and key operations. This allows subgraphs with different operator compositions but similar loop-centric features to be mapped to the same +*Graph*, and share the same codegen schedule. This enhances PluS's adaptability to varying model structures. The +*Graph* consists of multiple levels of +*Loop*s. Typically, computations on a tensor can be expressed as nested loops. +*Loop* is an abstraction of such loops, with the nested loops of tensor computations corresponding to the nested +*Loop*s.

As a subgraph identifier for pattern identification, +*Graph* must possess two key features: it encapsulates the underlying characteristics of loops and computations, and it maintains

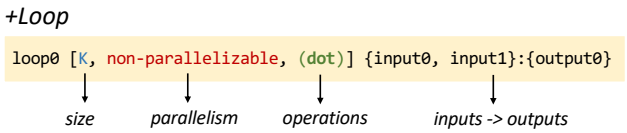


Figure 4: The +*Loop* representation.

uniqueness—each subgraph has a distinct +*Graph*. In the following content, we first introduce the key properties of +*Loop*, which succinctly express essential information of a loop for the pattern abstraction. We then present a comprehensive list of transformation primitives of +*Loop*. When multiple operators are grouped into a single subgraph, the +*Loop*s are interconnected and undergo transformations based on these primitives, forming the unique +*Graph* of the subgraph.

Properties. As shown in Fig. 4, +*Loop* is characterized by three main properties: *size*, *parallelism*, and *operation*.

Size describes the size of a +*Loop*, encompassing two types: *integer* and *symbolic*. When the size is known at the compile time, it will be a constant integer, otherwise it will be represented symbolically (i.e., dynamic shape).

Loop-level Parallelism assumes paramount significance for mapping +*Loop* computations onto hardware execution units. It can be classified into two categories: *parallelizable* and *non-parallelizable* +*Loop*. A *parallelizable* +*Loop* indicates that there are no data dependencies between consecutive loop iterations, allowing for parallel execution. Conversely, in the *non-parallelizable* +*Loop*, statements in an iteration of the +*Loop* depend on statements in another iteration of the +*Loop*, which necessitates serial execution or synchronization efforts. For instance, the innermost +*Loop* of a reduction operator (ReduceOp) is non-parallelizable, while each level of the +*Loop* in an elementwise operator is parallelizable.

Operation is an optional property that records computational operations within a +*Loop*. When users provide +*Graph* in the pattern warehouse, they need to specify only the crucial operations that determine codegen schedules in the subgraph pattern, such as dot product, reduce-max, etc., rather than specifying all trivial operations. PluS will identify subgraph patterns by leveraging the information within the *Operation* property. If an operation is defined in a +*Graph*, only subgraphs whose corresponding +*Loop* contains this operation will match the pattern (detailed in Sec. 4.2). Other trivial operations not specified by users will be ignored in the subgraph's +*Loop*. For instance, users can specify the permutation (transpose) operation in +*Graph* to determine whether the permutation operator can be included in a subgraph.

In addition to the three properties mentioned above, +*Loop*s in a subgraph also carry *input and output tensors*. However, this information does not affect pattern matching as this does not affect the underlining codegen schedule design. This means that subgraphs with different inputs and outputs might still map to the same +*Graph*, depending on the key operations within the subgraph. We will discuss the input and output features in the code generation section in Sec. 4.3.

Transformation primitives. As an identifier of a subgraph, the +*Graph* representation should be deterministic and unique for a given subgraph. Our insight is to merge the loops in a graph as much as possible while maintaining correctness to simplify the representation of the subgraph, making pattern identification easier. To achieve this, we design three primitives to guide the transformation of +*Loop*s across different operators when forming a subgraph. After the +*Loop*s of the subgraph are transformed, a fourth primitive collapses nested +*Loop*s for final standardization.

Assuming operator B follows operator A, i.e., B's input corresponds to A's output, we define the tensor connecting these two operators as the *hub tensor*. The +*Loop* associated with this hub tensor in operator A is referred to as *prev_loop*. The corresponding loop of operator B is denoted as *cur_loop*.

Primitive 1: Merge without Altering Properties. This primitive involves merging prev_loop and cur_loop when they share identical sizes and are both parallelizable.

For example, consider merging MatmulOp [M, N, K] and AddOp [M, N] into a +*Graph*, where axes M and N can be seamlessly merged without altering their properties. Before the merge, the computation of MatmulOp is illustrated in Fig. 5(1)(a). After the merge, the computation of MatmulOp and AddOp is depicted in Fig. 5(1)(b). The final +*Loop* structure of the subgraph is determined as shown in Fig. 5(1)(c).

Primitive 2: Merge with Parallelism Modification. In cases where prev_loop and cur_loop share identical sizes, and prev_loop is parallelizable while cur_loop is non-parallelizable, when merging the two loops, the *Parallelism* property is adjusted to non-parallelizable.

For instance, consider the fusion of AddOp [M, N] and ReduceOp [M, N], where the reduction axis is N. Since the N axis of AddOp is parallelizable while the N axis of ReduceOp is non-parallelizable, these +*Loop*s can be merged into a non-parallelizable +*Loop*. The computation of AddOp and this fused subgraph, and the +*Loop* structure of this subgraph are shown in Fig. 5(2).

Primitive 3: Transition to a New Loop. If the prev_loop is non-parallelizable, cur_loop cannot be merged and instead forms a distinct new +*Loop*, irrespective of the size and parallelism of cur_loop.

For instance, consider the fusion of ReduceOp [M, N] and AddOp [M, N], shown in Fig. 5(3)(a) and (b). Given that the N axis of ReduceOp is non-parallelizable, it results in the creation of a distinct new +*Loop* for the N axis of AddOp in Fig. 5(3)(c). This subgraph is represented as Fig. 5(3)(d).

Primitive 4: Nested loops collapsing. After the +*Loop*s in different operator are transformed, PluS collapses consecutive nested +*Loop*s that belong to the same categories within the subgraph to achieve the simplest representation under equivalent conditions. For instance, a BatchMatmulOp [B, L, M, N, D] can be simplified to BatchMatmulOp [B x L, M, N, D] because axes B and L both represent batch dimensions in this

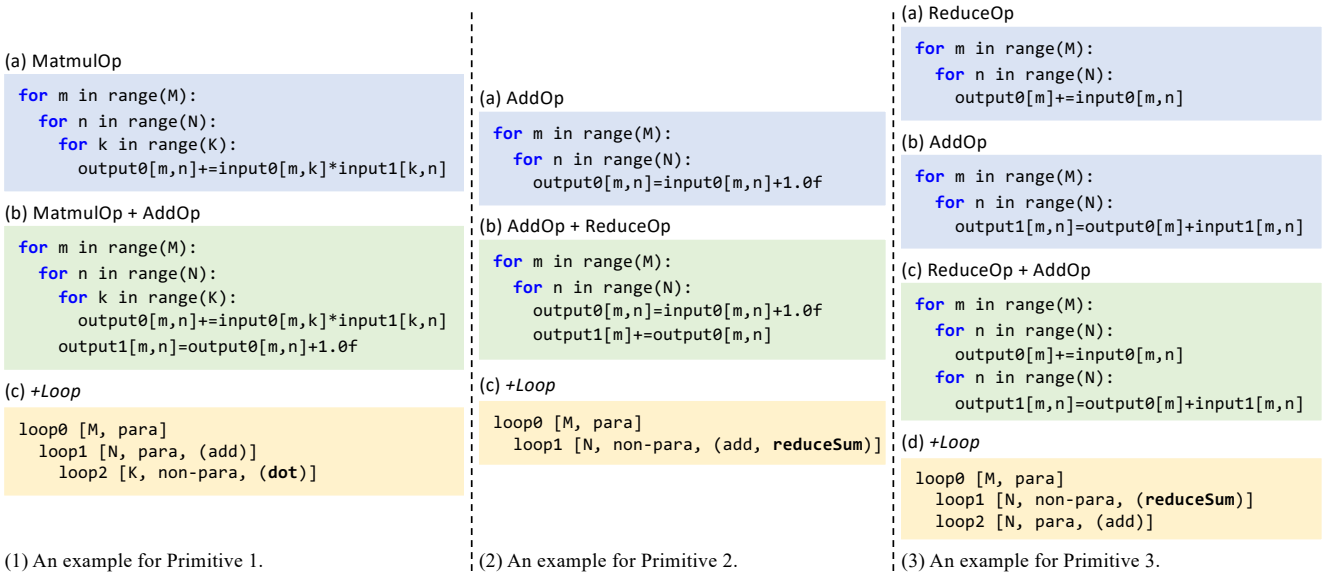


Figure 5: Examples for transformation primitives of +*Loop*.

scenario and can be collapsed.

Specifically, PluS determines the collapsibility of a +*Loop* and its next +*Loop* sequentially, adhering to three conditions for collapsing: First, the next +*Loop* must be the only +*Loop* nested inside the current +*Loop*. Second, the next +*Loop* must possess the same parallelism as the current +*Loop*. Third, the input tensors involved in both the next +*Loop* and the current +*Loop* must be exactly the same. For instance, consider the BatchMatmulOp [B x L, M, N, D]. Here, axes B and L are both parallelizable and present in all input tensors, whereas the parallelizable M axis exists only in the first input tensor, and the parallelizable N axis exists only in the second input tensor. The D axis is non-parallelizable.

4.2 Subgraph Identification

Based on the subgraph identifier +*Graph*, PluS employs a pattern identification method to generate subgraphs. This process is guided by two key insights. First, PluS begins subgraph identification with a skeleton operator. The concept of a *skeleton operator*, as proposed in [57] and [56], denotes a crucial operator that determines the data layout and codegen schedule within the computational graph, such as MatmulOp and ReduceOp. Most advanced optimized subgraphs are now built around these skeleton operators.

Second, PluS uses a greedy extension strategy to expand the subgraph scope when identifying subgraph patterns. This approach aims to match the largest possible range of subgraph patterns, including the maximum number of operators. Through subgraph extension, PluS iteratively matches the subgraph with pre-defined +*Graph*s in the pattern warehouse. The greedy extension strategy provides users with the greatest flexibility in code planning to decide the best implementation of the subgraphs. Unlike conventional ML compilers that map each subgraph into a single kernel, PluS allows users to freely determine the number of kernels in the code template of a +*Graph*. By expanding the range of subgraphs from the skeleton operator, PluS can identify most of the current mainstream subgraph optimizations.

In this section, we first present the steps of PluS's pattern identification method, which continually expands the subgraph and queries the pattern warehouse to match +*Graph*s. We then propose the match conditions to determine whether to continue or stop the extension.

Algorithm 1: Main steps of subgraph identification.

```
Input  : unfused_ops initialized by the computational graph of operators
Output : computational graph of fused subgraphs
1 while unfused_ops.IsEmpty() ≠ True do
2     skeleton_op = FindSkeletonOp(unfused_ops);
3     if skeleton_op then
4         MatchFromSkeletonOp(skeleton_op);
5     else
6         root_op = FindRootOp(unfused_ops);
7         MatchFromSkeletonOp(root_op);
```

Subgraph identification process. The main steps of PluS's subgraph identification are outlined as Algo. 1. The variable *unfused_ops* denotes the operators that have not yet been fused into a subgraph and is initialized by the computational graph. When the list of unfused operators is not empty (line 1), PluS attempts to identify a skeleton operator from unfused operators (line 2) and initiates pattern matching from that point (line 4). In PluS, we define the skeleton operator as the operator possessing at least one non-parallelizable +*Loop*, including MatmulOp and ReduceOp. If there is no skeleton operator in the unfused operators, PluS will start pattern matching from the root operator (lines 7-8), defined as the operator without a producer operator in the list in this paper.

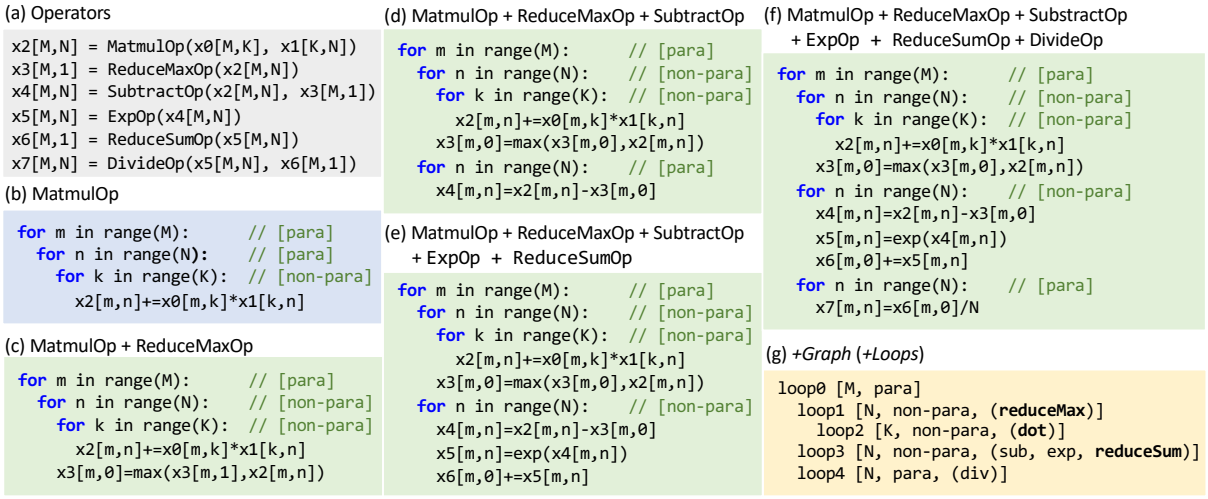


Figure 6: Case study for the subgraph identification workflow of PluS.

Algorithm 2: The steps of MatchFromSkeletonOp.

```
   Input : subgraph initialized by skeleton operator
   Input : Boolean flags: allow_prologue, allow_epilogue
1  if allow_prologue then
2      while prologue = FindPrologueOp(subgraph) do
3          if Matching(subgraph, prologue) then
4              FusePrologueOp(subgraph, prologue);
5  if allow_epilogue then
6      while epilogue = FindEpilogueOp(subgraph) do
7          if Matching(subgraph, epilogue) then
8              FuseEpilogueOp(subgraph, epilogue);
9              while producer = GetProducerOp(epilogue) do
10                 if Matching(subgraph, producer) then
11                     FusePrologueOp(subgraph, producer);
```

The steps of *MatchFromSkeletonOp* are outlined in Algo.2. First, PluS initializes the current *subgraph* with a skeleton operator. Each skeleton operator has two boolean properties predefined by experts in the pattern warehouse: *allow_prologue* and *allow_epilogue*, which determine the fusion direction of the skeleton operator. Specifically, if the skeleton operator allows prologue fusion (line 1), PluS continuously attempts to find a prologue operator of the current subgraph (line 2), which produces one of the input tensors of the current subgraph. PluS then assesses whether the subgraph meets the match conditions if the prologue operator is fused (line 3). If the conditions are met, PluS fuses this prologue operator into the subgraph (line 4). The criteria for determining whether the subgraph meets the match conditions will be detailed later.

Additionally, if the skeleton operator allows epilogue fusion (line 5), PluS continuously attempts to find an epilogue operator of the current subgraph that utilizes the output tensor of the current subgraph (line 6). PluS then evaluates whether the subgraph meets the match conditions if the epilogue operator is fused (line 7). If the conditions are met, PluS incorporates this epilogue operator into the subgraph (line 8). Moreover, unlike prologue operators, after an epilogue operator is fused, PluS endeavors to find the producer operator of this operator that is not in the subgraph (line 9) and assesses whether it can be fused (lines 10-11). For instance, after fusing a MatmulOp and a binary AddOp, PluS will also fuse the producer operator of another input tensor in the AddOp, such as the DivideScalarOp.

PluS applies the following prioritization to avoid ambiguity in subgraph matching. First, if an operator can serve either as an epilogue to a skeleton operator or as a prologue, PluS prioritizes epilogue fusion. This is because prologue fusion sometimes introduce redundant computations due to repeated input loading across multiple thread blocks, whereas epilogue operations do not, as the output is computed and written only once, such as the data processing in MatMul. Second, if an operator A produces two outputs, and each output could be fused with different downstream operators (B and C), but not both together, PluS does not enforce a fusion priority between them. PluS applies a greedy matching strategy: if B is encountered first during traversal, PluS fuses A with B, saves A's intermediate output, and computes C separately using that output. From a performance standpoint, fusing A with either B or C yields comparable efficiency, as the key benefit—avoiding redundant memory accesses and kernel launches—is achieved in both cases.

PluS fuses operators only to generate *+Graph*s as identifiers to match the pre-defined patterns and corresponding codegen schedules in the pattern warehouse. This approach differs from conventional compilers, which use hard-written fusion rules to generate code of subgraphs.

Case study for subgraph identification. We use Fig. 6 to illustrate how the *+Graph* of a subgraph is transformed during subgraph identification based on the transformation primitives. It is assumed that at each step the subgraph satisfies the match conditions. To exemplify, we consider the subgraph shown in Fig. 6, which consists of a subgraph comprising a GEMM operation followed by a Softmax operation. This subgraph

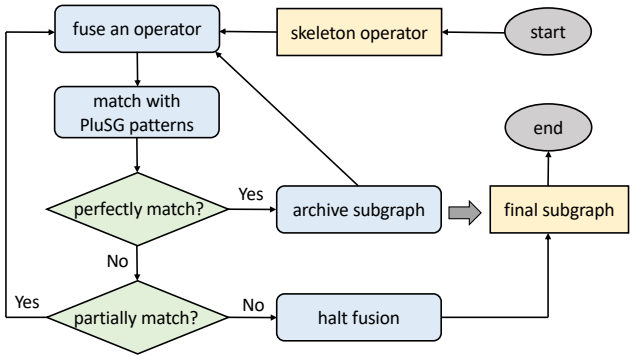


Figure 7: Workflow of match conditions.

encompasses six basic operators, as shown in Fig. 6(a).

Initially, PluS identifies a skeleton operator, MatmulOp [M, N, K], within the *unfused_ops*. The MatmulOp permits epilogue fusion while disallowing prologue fusion. The +*Loop* of MatmulOp [M, N, K] is shown in Fig. 6(b), where the M axis and N axis are parallelizable +*Loop*s, while the K axis is a non-parallelizable +*Loop*.

Subsequently, PluS identifies an epilogue operator, ReduceMaxOp [M, N], and attempts to incorporate it into the subgraph. During the fusion of MatmulOp and ReduceMaxOp, the properties of the M axis remain unaltered, following Primitive 1. However, the N axis is modified to be non-parallelizable in accordance with Primitive 2. The results are shown in Fig. 6(c).

Following that, PluS identifies an epilogue operator, SubtractOp [M, N]. The M axis can be merged according to Primitive 1 since the M axis in the subgraph and SubtractOp are both parallelizable. However, the N axis in the subgraph is non-parallelizable, while in SubtractOp, it is parallelizable. Hence, after fusing SubtractOp based on Primitive 3, the subgraph requires a new N axis, depicted in Fig. 6(d).

Similarly, as depicted in Fig. 6(e), PluS fuses injective ExpOp [M, N] where axes M and N are merged following Primitive 1, and ReduceSumOp [M, N] where M axis is merged following Primitive 1 and N axis is merged following Primitive 2. Additionally, a new N axis emerges within the subgraph during the fusion of DivideOp [M, N], according to Primitive 3. Finally, the computation of the resulting subgraph is depicted in Fig. 6(f), and can be represented by Fig. 6(g).

Match conditions. During the pattern matching process, PluS generates subgraphs represented by +*Graph* and evaluates them against match conditions with the patterns stored in the pattern warehouse to determine whether to extend further. Fig. 7 shows the workflow of this assessment process in PluS. Initially, it verifies whether each pattern in the warehouse perfectly aligns with the +*Graph* of the subgraph. If there is a complete match, PluS archives the subgraph as the most recently successfully matched subgraph corresponding to that skeleton operator and proceeds with the extension. This aligns with the principle of PluS, which is to expand subgraphs within the broadest scope possible.

In cases where there's no perfect match, PluS investigates whether the current subgraph represents a partial or intermediate state of a +*Graph* pattern in the warehouse. This involves determining if the current subgraph could potentially evolve into a specific +*Graph* pattern in the future. PluS accomplishes this by recursively traversing the multi-level +*Loop* tree, considering the following criteria for each +*Loop*: First, if the +*Loop* in the warehouse pattern is non-parallelizable, the corresponding +*Loop* in the current subgraph can be either parallelizable or non-parallelizable, as parallelizable +*Loop*s might transform into non-parallelizable ones in subsequent extension as per Primitive 2. Second, if the +*Loop* in the warehouse pattern is parallelizable, the corresponding +*Loop* in the current subgraph must also be parallelizable. Third, the number of +*Loop*s nested within one +*Loop* in the warehouse pattern must be greater than or equal to the number of +*Loop*s nested within the corresponding +*Loop* in the current subgraph, as subsequent extension operations might introduce new +*Loop*s. Fourth, if the warehouse pattern includes an operation, such as dot product and max-reduction, the subgraph must match it exactly.

Finally, two outcomes emerge: First, if the current subgraph doesn't align with any +*Graph* pattern in the pattern warehouse in its partial or intermediate state, the pattern matching process for the current skeleton operator is halted. The last archived subgraph, i.e., the most recently matched subgraph corresponding to a warehouse pattern, becomes the final subgraph for the current skeleton operator. Second, if the current subgraph represents the partial or intermediate state of a specific pattern in the pattern warehouse, the pattern matching process continues.

Case study for match conditions. Taking the subgraph in Fig. 6(d) as an example, which includes MatmulOp, ReduceMaxOp, and SubtractOp, its +*Graph* (+*Graph0*) can be represented as follows.

```
loop0 [M, parallelizable]
  loop1 [N, non-parallelizable, (reduceMax)]
    loop2 [K, non-parallelizable, (dot)]
  loop3 [N, parallelizable, (sub)]
```

While this subgraph does not perfectly match the +*Graph* pattern in Fig. 6(g) (+*Graph1*), it can be identified as a partial state of that pattern. First, loop0 in +*Graph1* is parallelizable, and loop0 in +*Graph0* is also parallelizable. Second, loop3 in +*Graph1* is non-parallelizable, whereas loop3 in +*Graph0* can be parallelizable. Third, the number of +*Loops* nested within +*Graph1*-loop0 (loop1, loop3, loop4) is greater than those nested within +*Graph0*-loop0 (loop1, loop3). Finally, the skeleton operations involving in +*Graph0*, specifically `reduceMax` and `dot`, are the same as those in +*Graph1*.

4.3 Code Generation

In this section, we describe how PluS generates code for subgraphs. Subgraph optimization experts first provide mappings between a +*Graph* and the code template in the pattern

```
loop0 [M, para] {x0,x1}:{x7}
  loop1 [N, non-para, (reduceMax)] {x2}:{x3}
    loop2 [K, non-para, (dot)] {x0,x1}:{x2}
  loop3 [N, non-para, (sub, exp, reduceSum)] {x2,x3}:{x6}
  loop4 [N, para, (div)] {x6}:{x7}
```

Figure 8: An example for loop bodies.

warehouse. The +*Graph* encapsulates the loop skeleton of the subgraph with key operations, while the code template defines the codegen schedules for this skeleton, leaving the code for trivial operations to fill by compiler. PluS compiler will maintain every detailed information during subgraph identification and generate the remaining code for the subgraph.

PluS compiler. During actual compilation, PluS retains information about all the data dependencies and operations of each +*Loop*, including the input tensors, output tensors, and computations involved in each loop. When the PluS compiler fills the code template of the +*Graph* with specific code, it traverses from the leaf nodes of the +*Loop* tree and generates the corresponding data and operation code, because the +*Loop* body of each leaf node can be implemented as an independent code segment. For instance, consider the subgraph in Fig. 6(g). PluS compiler will generate code for four +*Loop* bodies in turn: loop0-loop1-loop2, loop0-loop1, loop0-loop3 and loop0-loop4. The inputs, outputs, and operations involved in each of these +*Loop*s are illustrated in Fig 8.

+*Code* interface. Each matched subgraph pattern has a expert-provided code template in the pattern warehouse. Experts can create these templates by adding +*Code* statements in the code. +*Code* includes three types: data placeholder, compute, and data write-back, as shown in Fig 9. A *data placeholder* can represent any number of input and output tensors in the code. The compiler will complete the code for pointers to the data at this position based on the specific +*Loop* information. *Compute*-type statement tells the compiler where the data computation occurs in the code. If the data has not been loaded before computation, the compiler will automatically generate the code to load the data based on the addressing information in the +*Loop*, followed by the corresponding computation code. *Data write-back* usually occurs after all +*Loop* body computations are completed, indicating to the compiler to generate code that writes the computation results back to global memory. Typically, the compiler needs to address computations and write-backs. PluS has also implemented a compilation path compatible with CUTLASS, which is widely used by many ML compiler backends. Since CUTLASS provides a wrapper for implementing high-performance linear algebra on CUDA, PluS's compiler does not need to handle those details in code generation.

Case study for code generation. Continuing with the subgraph from Fig. 6 as an example, an expert might want to define a codegen schedule for subgraphs with this skeleton. First, they would define the pattern of this subgraph, or +*Graph*, in the pattern warehouse, retaining only the skeleton operations in the pattern. This allows the compiler to reuse the codegen

```
// data placeholder
plus::InputList input_list;
plus::OutputList output_list;
// compute
plus::compute</*DataType*/float>(input_list,
              /*Result*/local_out);
// data write-back
plus::writeback(local_out, output_list);
```

Figure 9: +*Code* statements.

schedule for any subgraph that matches the same structure.

Then, in the code template, the expert defines the code design for the loop0-loop1-loop2 structure, which corresponds to a matrix multiplication pattern. The expert has the flexibility to use a simple cuBLAS interface, the Matmul templates from CUTLASS, or even custom handwritten code. Since MatmulOp is a skeleton operator that allows epilogue fusion, the expert uses the *compute* statement in the code to reserve a placeholder for epilogue operations. Next, the expert implements the schedule logic for loop0-loop1 and loop0-loop2, handling the reduction logic by designing parallel execution and synchronization operations as needed. They can also decide whether to integrate this logic with the previous `dot` operation to achieve the most efficient schedule. Similarly, *compute* placeholders for prologue operations are reserved before reduction operations. Finally, since the loop0-loop4 loop body does not contain skeleton operations, the expert only needs to reserve a placeholder with a *compute* statement for this loop body. After all the loops are completed, a data write-back statement is added to the code.

Dynamic shape. PluS is capable of compiling models with dynamic shapes. It uses symbolic types to represent shapes during subgraph identification. These symbolic dimensions are carried through the compilation process, and the corresponding code templates treat shapes as runtime variables. Experts can then autonomously decide dynamic schedules based on the actual input sizes. PluS enables broad flexibility in handling dynamic shapes by allowing multiple implementations for the same subgraph pattern. For example, if different shapes require different fusion or scheduling strategies, experts can define a subgraph pattern with maximal scope, and encode shape-based routing logic within the code template.

5 Implementation

Recalling the overview of PluS outlined in Sec. 3, PluS has the capability to compile model representations compatible with various mainstream deep learning frameworks, and our current implementation is specifically based on PyTorch [32].

Pattern warehouse. PluS incorporates an expert-defined pattern warehouse that facilitates the dynamic insertion and deletion of patterns at any given time. This warehouse serves as a mapping between +*Graph*s and their corresponding kernel implementations. Users can define subgraph patterns through two methods. The first method allows them to deploy optimized code for a specific subgraph by providing a

torch.nn.Module and the corresponding code. For instance, deploying the FlashAttention optimization for the attention mechanism. In such cases, PluS automatically generates a *+Graph* and stores the provided code as a template in the pattern warehouse. Nevertheless, this approach may lack the desired flexibility and universality.

Meanwhile, PluS offers a second method where users can articulate the *+Graph*, as detailed in Sec. 4.1. Users also provide the associated code template to the pattern warehouse with *+Code* detailed in Sec. 4.3. Subgraphs with the same data layout share identical *+Graph*, minimizing the need for additional deployment efforts. PluS automatically generates codes for various operations.

Workflow. We implement a complete TorchDynamo-based backend. Initially, PluS takes a *torch.fx.GraphModule* as inputs and leverages Hidet [17]'s API to parse the graph module, resulting in a computational graph composed of low-level operators represented by the *+Graph* outlined in Sec. 4.1. Subsequently, PluS generates subgraphs through a pattern matching method explained in Sec. 4.2. Finally, PluS utilizes a code generation module detailed in Sec. 4.3 to produce codes for each subgraph. The code generation module maps the *+Graph* to codes using the templates stored in the pattern warehouse. In our current implementation, we seamlessly integrate kernels from CUTLASS, ByteTransformer, FlashAttention, and other sources.

Applicability discussion. PluS can support all DNN models, as all DNN computations are fundamentally loop-based computations involving multi-dimensional tensors. The subgraph abstraction introduced by PluS is built on this loop-based nature, which allows various types of operators to be supported through this abstraction.

6 Evaluation

6.1 Experimental Setup

Methodology. We compare PluS against mainstream DNN compilers and framework including TorchInductor [8] v2.4.0, TensorRT [22] v10.5.0 (in ONNX Runtime [15] v1.18.0), and AITemplate [45] v0.3.dev0. In our evaluation, we initially evaluate the end-to-end inference latency (Sec. 6.2). The effectiveness of PluS's *+Graph*-matching fusion strategies is demonstrated by presenting the layer count before and after fusion. Subsequently, we conduct several case studies to highlight the advantages of PluS in two aspects (Sec. 6.3). First, we report the subgraph inference latency to showcase PluS's ability to leverage efficient expert-optimized subgraph implementations. Second, we compare the size of code required by experts to add a new subgraph implementation against AITemplate, demonstrating that PluS can accommodate new subgraph implementations with minimal effort.

Platforms. We conduct experiments on two GPUs with different architectures: an NVIDIA A100 PCIe 80GB and an NVIDIA GeForce RTX 4090. The servers are equipped with AMD EPYC 7V13 CPUs and Intel Xeon Gold 5318Y CPUs, running Ubuntu 22.04 with CUDA 11.8 installed.

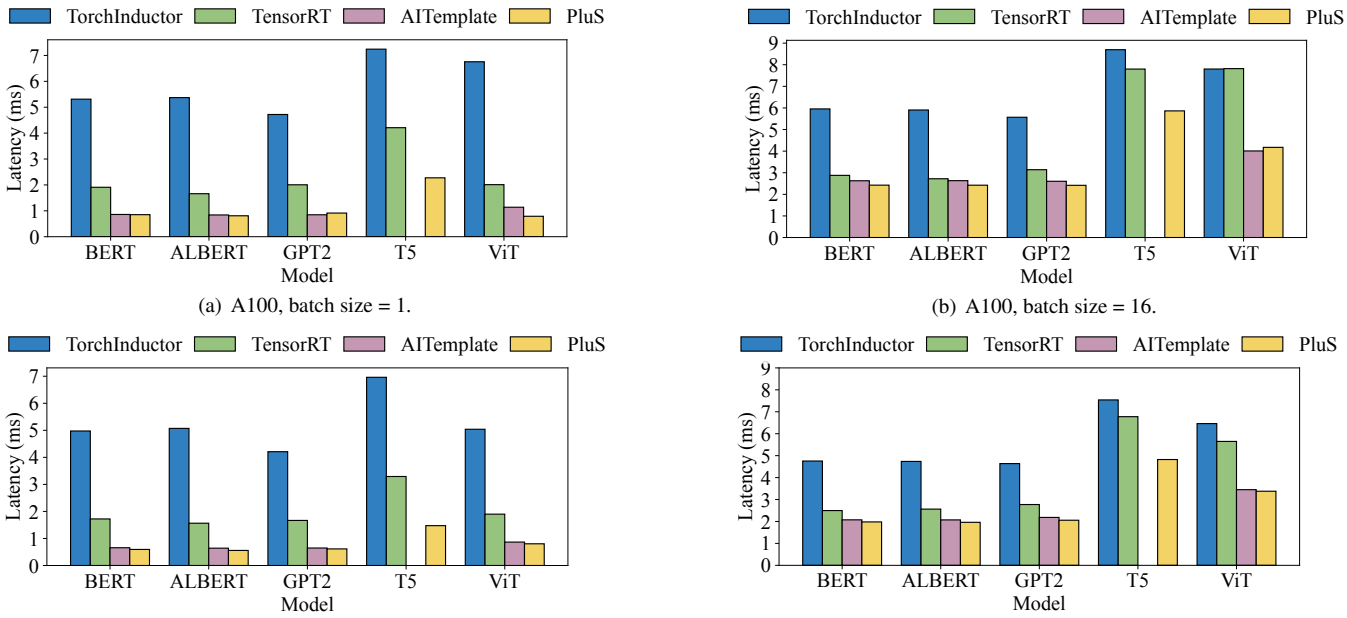
Workloads. We use five typical DNN models for evaluation, including BERT-base [16], ALBERT [25], GPT-2 [33], T5 [34] and ViT [18], encompassing popular transformer architectures in both natural language processing and computer vision. For the experiments, we set the batch sizes to 1 and 16, with a default sequence length of 128 for language models and an input image size of 224×224 for vision models.

6.2 End-to-End Evaluation

Performance. We first evaluate the end-to-end performance of PluS by comparing it against TorchInductor, TensorRT, and AITemplate. For TorchInductor, we employ the TorchInductor compiler provided by TorchDynamo to compile the PyTorch models, but dynamic shape support is disabled due to its poor performance. To evaluate TensorRT, we utilize the TensorRT provider in the Onnx Runtime with dynamic shape enabled, defining a batch size range of 1 to 32 and a sequence length range of 64 to 256 for the TensorRT engine. Furthermore, AITemplate does not support dynamic shape, whereas PluS supports the compilation of PyTorch models with dynamic shape. For each experiment, we measure the average end-to-end latency over 10 executions, repeated three times after 10 warm-up executions. One exception is that we observe unusually long GPU-to-CPU memory transfer time when running the T5 model in TensorRT, resulting in the end-to-end runtime exceeding 300 ms. Therefore, we perform additional profiling of the T5 model in TensorRT using Nsight Systems [1] and use the pure kernel execution time from the profiling data as the end-to-end runtime for the T5 model.

Fig. 10 shows the end-to-end inference latency on all workloads for batch sizes 1 and 16. PluS, when compared to state-of-the-art deep learning compilers with embedded graph transformation, outperforms TorchInductor by an average of 4.04× and TensorRT by 1.77× on A100 GPU. On the RTX 4090 GPU, it achieves an average speedup of 4.59× over TorchInductor and 2.01× over TensorRT. In Sec. 6.3, we delve into a subgraph performance analysis to showcase PluS's advantages in supporting highly optimized subgraph implementations.

Additionally, as a pluggable graph transformation solution, PluS demonstrates comparable end-to-end performance with AITemplate, even exhibiting an average 7.8% performance improvement on the A100 and 7.2% on the RTX 4090. This achievement stems from PluS's ability to leverage all subgraph optimization techniques available in AITemplate, alongside additional techniques from various vendors, such as ByteTransformer [47] and FlashInfer [46]. Notably, the performance of the T5 model in AITemplate is not reported, as we cannot define the `T5LayerNorm` module using the AITemplate interface. Meanwhile, AITemplate is designed solely for static shapes, limiting its flexibility in deployment. A case study in Sec. 6.3 further illustrates the portability of PluS in



(a) A100, batch size = 1.

(b) A100, batch size = 16.

(c) RTX 4090, batch size = 1.

(d) RTX 4090, batch size = 16.

Figure 10: End-to-end inference latency on different GPU platforms.

Table 1: Fusion rate evaluation. #Ops represents the number of fundamental operators comprising the DNN model, and #Layers denotes the number of layers after graph transformation within DNN compilers.

Model	#Ops	#Layers			
		TorchInductor	TensorRT	AITemplate	**PluS**
BERT	635	195	107	88	**87**
ALBERT	637	195	107	89	**88**
GPT2	630	171	126	89	**87**
T5	1460	364	247	-	**220**
ViT	655	201	105	90	**87**

comparison to AITemplate.

Fusion rate. In this section, we report the number of fundamental operators and the number of layers after subgraph identification (i.e., fusion) in PluS, TorchInductor, TensorRT, and AITemplate across all evaluated models. We define the fusion rate as the ratio of the original operator count to the fused layer count as [31]. As illustrated in Tab. 1, PluS demonstrates an average 2.08× and 1.25× fusion rate improvement compared to TorchInductor and TensorRT, respectively. This is due to PluS's capability to generate fused subgraphs with a larger scope, facilitated by its flexible pattern warehouse, in contrast to TorchInductor and TensorRT's rule-based embedded graph transformation module. Moreover, PluS exhibits subgraph fusion behavior similar to that of AITemplate, as both are ML compilers equipped with pluggable graph transformation modules, sharing advanced capabilities for deploying state-of-the-art subgraph optimizations. However, PluS can identify subgraphs with broader operator variations and leverage the reuse of codegen schedules, enabling it to support a wider range of models than AITemplate, such as T5.

Runtime overhead. We analyze the compilation time and memory overhead of PluS. PluS incurs a compilation overhead of 18 to 25 seconds when the model is cached, which is dominated by NVCC compilation. It takes 1 to 2 minutes to compile the model for the first time. The memory overhead of PluS includes the memory consumption of pattern warehouse and the intermediate representations, all of which reside on the CPU. After excluding memory consumed by other Python modules such as torch andtransformers, PluS introduces a memory overhead of 130 to 190 MB.

6.3 Case Studies

Subgraph optimization. In this section, we utilize the benchmark BERT on A100 GPU to dissect why PluS outperforms other frameworks. Tab.2 delineates the configurations of subgraphs that consume over 95% of end-to-end execution time, detailing the operators constituting each subgraph along with shape information. The final column specifies the implementation vendor PluS utilizes for each subgraph. PluS integrates highly optimized kernel implementations from CUTLASS [40], ByteTransformer [47], and FlashAttention [13, 14]. We dynamically select the optimal solution for different configurations during the pattern definition phase. It is noteworthy that we adopt the implementation from FlashAttention for multi-head attention in batch size 16 (S1) and the implementation from ByteTransformer for multi-head attention in batch size 1 (S6). Both implementations demonstrate optimal performance within their respective configurations during our evaluation.

We utilize Nsight Systems [1] to measure the latency of subgraphs. Comparisons of subgraph latency between PluS, TensorRT, and AITemplate are presented in Fig. 11. The results demonstrate that PluS achieves an average speedup of

Table 2: Dominant subgraph configurations in BERT.

Note	Operators	Configuration	PluS vendor
S0	Matmul+Bias	M=2048,N=2304,K=768	CUTLASS
S1	Split+Permute+Matmul+ Div+Add+Softmax+ Matmul+Permute	M0=2048,N0=128,K0=64 M1=2048,N1=64,K1=128	FlashAttention
S2	Matmul+Bias+ Add+LayerNorm	M=2048,N=768,K=768	ByteTransformer
S3	Matmul+Bias+Gelu	M=2048,N=3072,K=768	CUTLASS
S4	Matmul+Bias+ Add+LayerNorm	M=2048,N=768,K=3072	ByteTransformer
S5	Matmul+Bias	M=128,N=2304,K=768	CUTLASS
S6	Split+Permute+Matmul+ Div+Add+Softmax+ Matmul+Permute	M0=128,N0=128,K0=64 M1=128,N1=64,K1=128	ByteTransformer
S7	Matmul+Bias+ Add+LayerNorm	M=128,N=768,K=768	ByteTransformer
S8	Matmul+Bias+Gelu	M=128,N=3072,K=768	CUTLASS
S9	Matmul+Bias+ Add+LayerNorm	M=128,N=768,K=3072	ByteTransformer

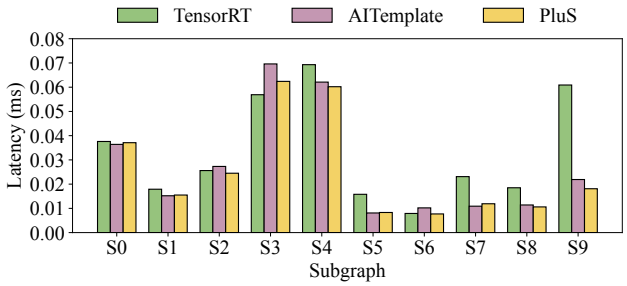


Figure 11: Subgraph latency.

1.53× over TensorRT and 1.07× over AITemplate. Moreover, PluS retains the ability to reduce their latency by incorporating superior optimizations subsequently.

Ablation study. We then compare PluS with a configuration that disables optimized kernel sources (i.e., CUTLASS, ByteTransformer, and FlashAttention), falling back to generic implementations. PyTorch eager mode is used as the default baseline. Experimental results on the ten subgraphs show that PluS achieves an average speedup of 4.79× and 13.27× over the baseline with batch size of 16 and 1, respectively.

We further analyze why flexibly supporting a variety of subgraph patterns is necessary. Take the subgraphs S1 and S6 as examples. If these subgraphs are executed in PyTorch's eager mode—where each operator is executed separately—the total runtime for S1 and S6 is 0.221 ms and 0.209 ms, respectively. This demonstrates the poor performance that results from a lack of operator fusion. Next, when using TorchInductor, it excutes three kernels for this subgraph, fusing the element-wise operators with MatmulOp and implementing SoftmaxOp as a whole. Under this mode, the runtime for S1 and S6 becomes 0.0183 ms and 0.0078 ms. Furthermore, TensorRT, AITemplate and our PluS go a step further by implementing S1 and S6 as a single subgraph, generating one large kernel to execute the entire subgraph [13, 14]. They achieve better performance due to the tighter integration, with execution times of 0.015 ms and 0.0077 ms in PluS. The results demonstrate that different graph optimization techniques can lead to significant changes in model performance.

Portability. In this section, we use the T5 model as an example to showcase the portability of PluS compared to AITemplate, which does not support T5. Assuming that both frameworks can already support models like BERT, ALBERT, and GPT2, it becomes evident that PluS exhibits superior adaptability to minor alterations in DNN models, thereby reducing deployment complexities when compared to other pluggable ML compilers.

First, we observe that all Linear layers utilized in the T5 model lack bias operations, leading to the emergence of numerous new subgraphs not present in models like BERT, ALBERT, and GPT2. Intuitively, this subgraph should share the same code generation schedule as the subgraph with bias operations found in BERT, ALBERT, and GPT2, given their shared loop structure and data layout. However, to accommodate this fused subgraph in AITemplate, a new template must be added. This entails a developer writing both frontend and backend code for the Matmul+Relu module, comprising over 250 lines of code (LoC), and registering it in the pipeline. As similar subgraphs continue to arise, programmers must repetitively perform this task. In contrast, leveraging the +Graph and code templates previously added by experts in the pattern warehouse, PluS can seamlessly generate code for these subgraphs without additional efforts.

```
x1[M,N] = Pow(x0[M,N])
x2[M,1] = ReduceMean(x1[M,N])
x3[M,1] = AddScalar(x2[M,1])
x4[M,1] = Rsqrt(x3[M,1])
x5[M,N] = Multiply(x1[M,N], x4[M,1])
x6[M,N] = Multiply(x5[M,N], c[1,N])
```

Another case arises with T5 introducing a novel layer normalization method, T5LayerNorm [49], leading to a distinct subgraph composition as depicted above. To support this subgraph in AITemplate, developers must invest additional effort in writing an end-to-end pipeline, including frontend and backend components, due to the lack of inheritable similar subgraphs defined previously. This process involves 1701 LoC to accommodate all components related to layer normalization, including 453 LoC for frontend registration and 1248 LoC for backend implementation. Similarly, in PluS, a new subgraph pattern must be incorporated into the pattern warehouse due to the absence of subgraphs sharing the same subsequent +Loop structure. In total, PluS requires 18 LoC to define the +Graph and an additional 129 lines to provide

the necessary code template. In contrast, when new variants of this algorithm emerge, such as adopting a zero-centered gamma, PluS supports them seamlessly without requiring any additional effort.

```
loop0 [M, parallelizable]
  loop1 [N, non-parallelizable]
  loop2 [N, parallelizable]
```

Sensitivity analysis. PluS is designed as a functional compiler, where subgraph matching and operator fusion does not change the computation semantics. The code generation process in PluS strictly follows the data dependencies between operations, ensuring correct implementation of the backend kernels. Therefore, accuracy degradation is not expected under correct pattern definitions.

To validate the correctness and robustness of PluS, we compare the accuracy of subgraph outputs between the optimized and unoptimized execution paths using randomly initialized inputs from a standard normal distribution. We use PyTorch eager mode as the baseline. Experimental results show that the maximum absolute difference between the outputs is 1.9e-3, and the average absolute difference is 3.57e-5.

7 Related Works

Graph-level optimization. Graph-level optimization techniques are crucial for enhancing model execution in ML compilers [8, 10, 19, 22, 23, 30, 31, 36, 39, 45, 51, 52, 56, 57]. ML compilers can be categorized into two classes based on the approach to graph transformation. Embedded mode ML compilers [8,22,23,30,31,36,39,51,52,56,57], such as TensorFlow XLA [36] and TorchInductor [8], employ a rule-based strategy for graph enhancement. In particular, FractalTensor [28] employs nested loops to express DNN operator, breaking the optimization boundaries of traditional operator functions. However, they often overlook opportunities for expert-driven subgraph optimizations due to the complexities of their embedded graph transformation modules. In contrast, pluggable mode ML compilers like AITemplate [45] utilize fixed operator patterns for graph-level optimizations, sacrificing adaptability to structure-similar subgraphs. PluS offers a novel approach supporting flexible integration of pluggable graph schedules with an adaptable pattern-matching strategy based on loop-centric features. Additionally, FlexAttention [20] supports the implementation of different attentions variants as FlashAttention optimizations; however, its scope is limited to attention-specific adaptations. While TVM [10] and TensorFlow Grappler [19] allow users to catch and modify the entire computational graph with customized passes, they require developers to define complicated graph transformation rules from scratch. In contrast, PluS allows users to define graph optimization easily, providing a mapping of loop-centric subgraph pattern with associated code templates. It is also possible for us to implement PluS on top of TVM and TensorFlow Grappler, and integrate FlexAttention in the future.

Kernel-level optimization. Many deep learning frameworks [4, 5, 32] rely on ad-hoc kernels libraries [2, 3, 7, 13, 14, 27, 40, 47] to efficiently execute specific operators or subgraphs. NVIDIA's libraries like cuBLAS [2], CUTLASS [40], and cuSPARSE [3] provide high-performance implementations for operators and subgraphs, and frameworks like ByteTransformer [47], FasterTransformer [4], xFormers [27] and FlashInfer [46] offer optimized GPU kernels for transformer-based models. FlashAttention [13, 14] is a novel attention kernel implementation with fewer memory accesses. Meanwhile, ML compilers [6,9,11,12,17,21,24,38,42,44,52–55,58] focus on different operator-level scheduling strategies to improve efficiency. TVM [10], for instance, leverages Halide [35]'s scheduling principles to map tensor expressions to low-level code. AutoTVM [11] and Ansor [54] employ heuristic algorithms to automatically search optimal configurations. Tiramisu [9], AKG [52], and Tensor Comprehensions [42] automate scheduling space exploration using polyhedral models. PluS is a compiler that supports pluggable graph schedules, making it compatible with these manually-optimized and auto-scheduled approaches, and capable of accommodating various tensor program scheduling within its pattern warehouse.

8 Conclusion

ML compilers encounter new challenges in the face of partially convergent model architectures and the rapid evolution of graph optimization techniques. PluS addresses this challenge by offering pluggable graph scheduling based on the *+Graph* abstraction, enabling rapid deployment of high-performance subgraph implementations. Unlike existing ML compilers burdened by cumbersome embedded graph transformation rules, or limited by matching fixed operator composition, PluS introduces a pioneering loop-centric pattern-matching solution. Consequently, PluS outperforms state-of-the-art embedded ML compilers, achieving up to a 4.04× speedup, owing to its superior expandability for graph optimizations. Moreover, it demonstrates improved adaptability to model evolution while still preserving minor performance benefits (about 7.8%) compared to template-based solutions.

Acknowledgments

We sincerely thank all the anonymous reviewers and our shepherd, Prof. Saurabh Bagchi, for their insightful comments and feedback. This work is supported by the Project of Key R&D Program of Shandong Province (2024CXGC010113), National Natural Science Foundation of China (No. 62461146205 and 62322213), and Beijing Nova Program (No. 20230484397 and 20220484137). Ruofan Wu, Feng Zhang, Zaifeng Pan, and Xiaoyong Du are with the Key Laboratory of Data Engineering and Knowledge Engineering (MOE), and School of Information, Renmin University of China. Feng Zhang is the corresponding author of this paper (fengzhang@ruc.edu.cn).

References

[1] Nvidia nsight systems. https://developer.nvidia.com/nsight-systems.

[2] Basic linear algebra on nvidia gpus. https://developer.nvidia.com/cublas, 2024.

[3] Gpu library apis for sparse computation. https://developer.nvidia.com/cusparse, 2024.

[4] Nvidia fastertransformer. https://github.com/NVIDIA/FasterTransformer, 2024.

[5] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: a system for large-scale machine learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, page 265–283, USA, 2016. USENIX Association.

[6] Peter Ahrens, Fredrik Kjolstad, and Saman Amarasinghe. Autoscheduling for sparse tensor algebra with an asymptotic cost model. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2022, page 269–285, New York, NY, USA, 2022. Association for Computing Machinery.

[7] Reza Yazdani Aminabadi, Samyam Rajbhandari, Ammar Ahmad Awan, Cheng Li, Du Li, Elton Zheng, Olatunji Ruwase, Shaden Smith, Minjia Zhang, Jeff Rasley, and Yuxiong He. Deepspeed-inference: enabling efficient inference of transformer models at unprecedented scale. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '22. IEEE Press, 2022.

[8] Jason Ansel, Edward Yang, Horace He, Natalia Gimelshein, Animesh Jain, Michael Voznesensky, Bin Bao, Peter Bell, David Berard, Evgeni Burovski, Geeta Chauhan, Anjali Chourdia, Will Constable, Alban Desmaison, Zachary DeVito, Elias Ellison, Will Feng, Jiong Gong, Michael Gschwind, Brian Hirsh, Sherlock Huang, Kshiteej Kalambarkar, Laurent Kirsch, Michael Lazos, Mario Lezcano, Yanbo Liang, Jason Liang, Yinghai Lu, C. K. Luk, Bert Maher, Yunjie Pan, Christian Puhrsch, Matthias Reso, Mark Saroufim, Marcos Yukio Siraichi, Helen Suk, Shunting Zhang, Michael Suo, Phil Tillet, Xu Zhao, Eikan Wang, Keren Zhou, Richard Zou, Xiaodong Wang, Ajit Mathews, William Wen, Gregory Chanan, Peng Wu, and Soumith Chintala. Pytorch 2: Faster machine learning through dynamic python bytecode transformation and graph compilation. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS '24, page 929–947, New York, NY, USA, 2024. Association for Computing Machinery.

[9] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman Amarasinghe. Tiramisu: a polyhedral compiler for expressing fast and portable code. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization*, CGO 2019, page 193–205. IEEE Press, 2019.

[10] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: An automated End-to-End optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 578–594, Carlsbad, CA, October 2018. USENIX Association.

[11] Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Learning to optimize tensor programs. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018.

[12] Stephen Chou, Fredrik Kjolstad, and Saman Amarasinghe. Automatic generation of efficient sparse tensor format conversion routines. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 823–838, New York, NY, USA, 2020. Association for Computing Machinery.

[13] Tri Dao. FlashAttention-2: Faster attention with better parallelism and work partitioning. 2023.

[14] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. FlashAttention: Fast and memory-efficient exact attention with IO-awareness. In *Advances in Neural Information Processing Systems*, 2022.

[15] ONNX Runtime developers. ONNX Runtime, November 2018.

[16] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. In Jill Burstein, Christy Doran, and Thamar Solorio, editors,

Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers), pages 4171–4186. Association for Computational Linguistics, 2019.

[17] Yaoyao Ding, Cody Hao Yu, Bojian Zheng, Yizhi Liu, Yida Wang, and Gennady Pekhimenko. Hidet: Task-mapping programming paradigm for deep learning tensor programs. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS 2023, page 370–384, New York, NY, USA, 2023. Association for Computing Machinery.

[18] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929*, 2020.

[19] Google. TensorFlow graph optimization with Grappler. https://www.tensorflow.org/guide/graph_optimization, 2024.

[20] Horace He, Driss Guessous, Yanbo Liang, and Joy Dong. FlexAttention: The Flexibility of PyTorch with the Performance of FlashAttention, 2024.

[21] Guyue Huang, Yang Bai, Liu Liu, Yuke Wang, Bei Yu, Yufei Ding, and Yuan Xie. Alcop: Automatic load-compute pipelining in deep learning compiler for ai-gpus. *Proceedings of Machine Learning and Systems*, 5, 2023.

[22] NVIDIA Inc. NVIDIA TensorRT., 2022.

[23] Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. Taso: optimizing deep learning computation with automatic generation of graph substitutions. SOSP '19, page 47–62, New York, NY, USA, 2019. Association for Computing Machinery.

[24] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. The tensor algebra compiler. *Proc. ACM Program. Lang.*, 1(OOPSLA):77:1–77:29, October 2017.

[25] Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, and Radu Soricut. Albert: A lite bert for self-supervised learning of language representations. In *International Conference on Learning Representations*, 2020.

[26] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. Mlir: A compiler infrastructure for the end of moore's law, 2020.

[27] Benjamin Lefaudeux, Francisco Massa, Diana Liskovich, Wenhan Xiong, Vittorio Caggiano, Sean Naren, Min Xu, Jieru Hu, Marta Tintore, Susan Zhang, Patrick Labatut, Daniel Haziza, Luca Wehrstedt, Jeremy Reizenstein, and Grigory Sizov. xformers: A modular and hackable transformer modelling library. https://github.com/facebookresearch/xformers, 2022.

[28] Siran Liu, Chengxiang Qi, Ying Cao, Chao Yang, Weifang Hu, Xuanhua Shi, Fan Yang, and Mao Yang. Uncovering nested data parallelism and data reuse in dnn computation with fractaltensor. In *SOSP*, November 2024.

[29] Yaoyang Liu, Zhen Zheng, Feng Zhang, Jincheng Feng, Yiyang Fu, Jidong Zhai, Bingsheng He, Xiao Zhang, and Xiaoyong Du. A comprehensive taxonomy of prompt engineering techniques for large language models. *Frontiers of Computer Science*, 2025.

[30] Lingxiao Ma, Zhiqiang Xie, Zhi Yang, Jilong Xue, Youshan Miao, Wei Cui, Wenxiang Hu, Fan Yang, Lintao Zhang, and Lidong Zhou. Rammer: Enabling holistic deep learning compiler optimizations with rTasks. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 881–897. USENIX Association, November 2020.

[31] Wei Niu, Jiexiong Guan, Yanzhi Wang, Gagan Agrawal, and Bin Ren. Dnnfusion: accelerating deep neural networks execution with advanced operator fusion. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, page 883–898, New York, NY, USA, 2021. Association for Computing Machinery.

[32] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019.

[33] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. 2019.

[34] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *J. Mach. Learn. Res.*, 21(1), jan 2020.

[35] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *SIGPLAN Not.*, 48(6):519–530, jun 2013.

[36] Amit Sabne. Xla : Compiling machine learning for peak performance, 2020.

[37] Noam Shazeer. Glu variants improve transformer. *arXiv preprint arXiv:2002.05202*, 2020.

[38] Haichen Shen, Jared Roesch, Zhi Chen, Wei Chen, Yong Wu, Mu Li, Vin Sharma, Zachary Tatlock, and Yida Wang. Nimble: Efficiently compiling dynamic neural networks for model inference. *Proceedings of Machine Learning and Systems*, 3:208–222, 2021.

[39] Yining Shi, Zhi Yang, Jilong Xue, Lingxiao Ma, Yuqing Xia, Ziming Miao, Yuxiao Guo, Fan Yang, and Lidong Zhou. Welder: Scheduling deep learning memory access via tile-graph. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 701–718, 2023.

[40] Vijay Thakkar, Pradeep Ramani, Cris Cecka, Aniket Shivam, Honghao Lu, Ethan Yan, Jack Kosaian, Mark Hoemmen, Haicheng Wu, Andrew Kerr, Matt Nicely, Duane Merrill, Dustyn Blasig, Fengqi Qiao, Piotr Majcher, Paul Springer, Markus Hohnerbach, Jin Wang, and Manish Gupta. CUTLASS, January 2023.

[41] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023.

[42] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. *CoRR*, abs/1802.04730, 2018.

[43] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[44] Jiarong Xing, Leyuan Wang, Shang Zhang, Jack Chen, Ang Chen, and Yibo Zhu. Bolt: Bridging the gap between auto-tuners and hardware-native performance. *Proceedings of Machine Learning and Systems*, 4:204–216, 2022.

[45] Bing Xu, Ying Zhang, Hao Lu, Yang Chen, Terry Chen, Mike Iovine, Mu-Chu Lee, and Zhijing Li. AITemplate, October 2022.

[46] Zihao Ye, Lequn Chen, Ruihang Lai, Wuwei Lin, Yineng Zhang, Stephanie Wang, Tianqi Chen, Baris Kasikci, Vinod Grover, Arvind Krishnamurthy, and Luis Ceze. Flashinfer: Efficient and customizable attention engine for llm inference serving. *arXiv preprint arXiv:2501.01005*, 2025.

[47] Y. Zhai, C. Jiang, L. Wang, X. Jia, S. Zhang, Z. Chen, X. Liu, and Y. Zhu. Bytetransformer: A high-performance transformer boosted for variable-length inputs. In *2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 344–355, Los Alamitos, CA, USA, may 2023. IEEE Computer Society.

[48] Biao Zhang and Rico Sennrich. Root mean square layer normalization. *Advances in Neural Information Processing Systems*, 32, 2019.

[49] Biao Zhang and Rico Sennrich. Root Mean Square Layer Normalization. In *Advances in Neural Information Processing Systems 32*, Vancouver, Canada, 2019.

[50] Feng Zhang, Chenyang Zhang, Jiawei Guan, Qiangjun Zhou, Kuangyu Chen, Xiao Zhang, Bingsheng He, Jidong Zhai, and Xiaoyong Du. Breaking the edge: Enabling efficient neural network inference on integrated edge devices. *IEEE Transactions on Cloud Computing*, 2025.

[51] Jie Zhao, Xiong Gao, Ruijie Xia, Zhaochuang Zhang, Deshi Chen, Lei Chen, Renwei Zhang, Zhen Geng, Bin Cheng, and Xuefeng Jin. Apollo: Automatic partition-based operator fusion through layer by layer optimization. *Proceedings of Machine Learning and Systems*, 4:1–19, 2022.

[52] Jie Zhao, Bojie Li, Wang Nie, Zhen Geng, Renwei Zhang, Xiong Gao, Bin Cheng, Chen Wu, Yun Cheng, Zheng Li, Peng Di, Kun Zhang, and Xuefeng Jin. Akg: automatic kernel generation for neural processing units using polyhedral transformations. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, page 1233–1248, New York, NY, USA, 2021. Association for Computing Machinery.

[53] Bojian Zheng, Ziheng Jiang, Cody Hao Yu, Haichen Shen, Joshua Fromm, Yizhi Liu, Yida Wang, Luis Ceze, Tianqi Chen, and Gennady Pekhimenko. Dietcode: Automatic optimization for dynamic tensor programs. *Proceedings of Machine Learning and Systems*, 4:848–863, 2022.

[54] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, Joseph E. Gonzalez, and Ion Stoica. Ansor: generating high-performance tensor programs for deep learning. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, OSDI'20, USA, 2020. USENIX Association.

[55] Size Zheng, Yun Liang, Shuo Wang, Renze Chen, and Kaiwen Sheng. Flextensor: An automatic schedule exploration and optimization framework for tensor computation on heterogeneous system. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 859–873, 2020.

[56] Zhen Zheng, Zaifeng Pan, Dalin Wang, Kai Zhu, Wenyi Zhao, Tianyou Guo, Xiafei Qiu, Minmin Sun, Junjie Bai, Feng Zhang, Xiaoyong Du, Jidong Zhai, and Wei Lin. Bladedisc: Optimizing dynamic shape machine learning workloads via compiler approach. *Proc. ACM Manag. Data*, 1(3), nov 2023.

[57] Zhen Zheng, Xuanda Yang, Pengzhan Zhao, Guoping Long, Kai Zhu, Feiwen Zhu, Wenyi Zhao, Xiaoyong Liu, Jun Yang, Jidong Zhai, Shuaiwen Leon Song, and Wei Lin. AStitch: enabling a new multi-dimensional optimization space for memory-intensive ML training and inference on modern SIMT architectures. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, page 359–373, New York, NY, USA, 2022. Association for Computing Machinery.

[58] Hongyu Zhu, Ruofan Wu, Yijia Diao, Shanbin Ke, Haoyu Li, Chen Zhang, Jilong Xue, Lingxiao Ma, Yuqing Xia, Wei Cui, Fan Yang, Mao Yang, Lidong Zhou, Asaf Cidon, and Gennady Pekhimenko. ROLLER: Fast and efficient tensor compilation for deep learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 233–248, Carlsbad, CA, July 2022. USENIX Association.

Obscura: Concealing Recomputation Overhead in Training of Large Language Models with Bubble-filling Pipeline Transformation

Yuzhou Huang[1], *Yapeng Jiang*[1], *Zicong Hong*[2], *Wuhui Chen*[1]*,
Bin Wang[3], *Weixi Zhu*[3], *Yue Yu*[4], *Zibin Zheng*[1]

[1]*Sun Yat-sen University,* [2]*Hong Kong University of Science and Technology,*
[3]*Huawei Technologies,* [4]*Peng Cheng Laboratory*

Abstract

Pipeline parallelism has become a widely adopted strategy for training large language models (LLMs) by distributing computational workloads across multiple nodes. However, it faces a significant challenge in the form of memory bottlenecks at early stages. While recomputation can mitigate this issue, it incurs additional computational overhead.

To address this limitation, we propose Obscura, a computationally efficient pipeline training system designed to optimize recomputation overhead under the given memory constraints. Leveraging the observation that bubbles following backward passes can conceal recomputation overhead in pipeline parallelism, Obscura introduces a novel pipeline transformation to enhance overhead concealment. Furthermore, we integrate swapping techniques into the pipeline and model the execution time as an optimization problem to identify an optimal recomputation strategy. A partition adjustment algorithm is also implemented to balance computation across stages under the transformation. Evaluations on Llama-2 and GPT-3 models of various sizes demonstrate that Obscura achieves throughput improvements of up to 1.33× compared to widely used recomputation baselines.

1 Introduction

A new AI paradigm leveraging large language models (LLMs) has emerged, significantly improving performance across various domains [3, 4, 6, 21, 23, 25]. However, the lack of extensive GPU resources poses significant challenges for small and medium enterprises (SMEs) in adopting this trend to meet their specific needs. Fine-tuning a Llama-2 13B [23] model, for example, demands approximately 400GB of GPU memory to store model states, activations, and other operations. This far exceeds the 80 GB memory of a single A100 GPU, necessitating at least six A100 GPUs to complete the task.

To accommodate larger models, distributed training has become a standard approach, leveraging multiple devices for distributed computation and storage through various parallelism strategies [8, 13, 20]. Among these strategies, pipeline parallelism [5, 8] stands out due to its lower communication overhead, which enables efficient scaling across multiple nodes and supports the training of larger models.

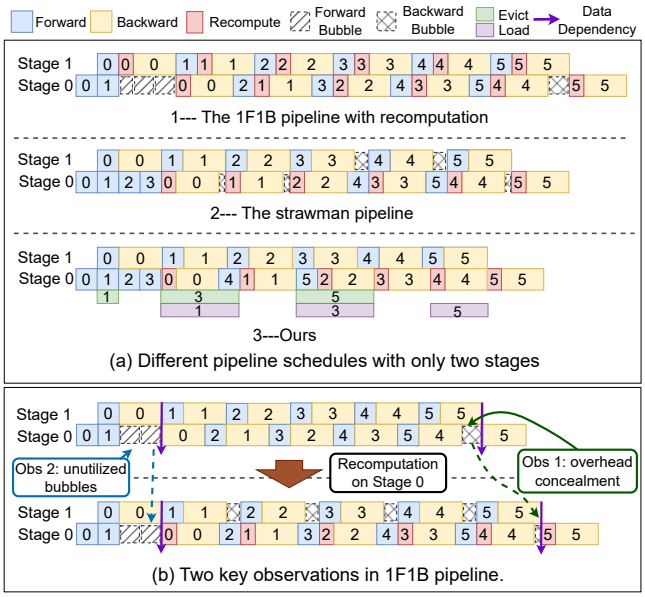


Figure 1: The timeline of three pipeline schedules.

The 1F1B (one-forward, one-backward) pipeline [5] is the most commonly used approach in pipeline parallelism, but it encounters a memory bottleneck in the early stages [10, 29]. Specifically, 1F1B pipeline alternates the scheduling of forward and backward pass across a large number of micro-batches. To maximize pipeline utilization, earlier stages execute more forward passes than later stages before initiating the first backward pass. This results in greater accumulation of micro-batch activations, leading to significantly higher memory usage in the early stages. For example, when training Llama 2-13B model using an 8-way 1F1B pipeline, memory usage in stage 0 surpasses that in stage 7 by 35 GB.

Recomputation [12], an effective technique for reducing

*Corresponding author: chenwuh@mail.sysu.edu.cn

memory usage caused by activations, involves discarding activations during the forward pass and recomputing them during the backward pass. In cases of out-of-memory (OOM) errors, as illustrated in Figure 1(a1), the conventional approach is to recompute all micro-batches across stages, discarding most activations to avoid OOM. However, this approach introduces substantial computational overhead and extends the pipeline duration due to the recomputation of each micro-batch. To alleviate this, several studies [11, 27] have proposed selectively recomputing cost-effective operators—those with high memory usage but low recomputation overhead—to minimize computational overhead. Nonetheless, under severe memory constraints, many non-cost-effective operators still need to be recomputed, which keeps the overhead substantially high.

In our exploration of integrating recomputation into pipeline parallelism, we apply recomputation only to stages experiencing memory bottlenecks, termed *On-Demand Recomputation*, achieving better performance than *All-Stage Recomputation*. Through an analysis of On-Demand Recomputation, we make the following key observations: **Backward Bubbles Can Conceal Recomputation Overhead:** *Backward Bubbles*, defined as bubbles between the first and last backward passes, are effectively utilized. As shown in Figure 1(b), in stage 0, backward bubbles are primarily filled by the recompute forward pass and the backward pass of micro-batch 4, which prevents delays in subsequent micro-batches. Moreover, **Forward Bubbles are Unutilized:** *Forward Bubbles*, defined as bubbles between the first forward pass and the first backward pass, are non-utilized. As they occur before the first backward pass, making them unavailable for recomputation.

Building on these findings, we propose a novel pipeline transformation: converting forward bubbles into backward bubbles to better conceal recomputation overhead. To achieve this, we introduce a strawman pipeline based on 1F1B pipeline through the following adjustments: First, stages that exceed memory constraints are identified as "adjusted stages," where recomputation is applied. Next, in adjusted stages, forward passes that occur after the first backward pass are migrated to execute before the first backward pass. This transformation converts forward bubbles into backward bubbles, enabling them utilization to conceal recomputation overhead. Therefore, the resulting pipeline, illustrated in Figure 1(a2), leverages the converted backward bubbles effectively conceal the recomputation overhead of micro-batches 0 and 1.

Although the strawman pipeline better conceals recomputation overhead to accelerate training, it still encounters three performance-related challenges: **1) How to Fully Utilize Backward Bubbles:** Due to tight data dependencies—where micro-batches are executed sequentially from the first stage to the last—each backward bubble can only conceal the recomputation of its preceding backward pass. This limitation leads to the underutilization of available bubbles. **2) How to Reduce Extra Memory Usage:** Compared to the 1F1B pipeline, adjusted stages in the strawman pipeline perform additional forward passes before the first backward pass. This results in the accumulation of more activations, increasing the likelihood of exceeding memory constraints. **3) How to Address the Computation Imbalance:** On-demand recomputation creates workload imbalances between adjusted and non-adjusted stages. While bubbles conceal some of the recomputation, their limited number and the increasing overhead with larger micro-batches leave some recomputation unconcealed, creating idle bubbles in non-adjusted stages.

To address these challenges, we propose Obscura, a computationally efficient pipeline training system for LLMs that leverages pipeline transformation to better conceal recomputation overhead, as illustrated in Figure 1(a3). Obscura enhances the strawman pipeline by introducing three key components to further improve performance:

Dependency Relaxation. To relax data dependencies, Obscura introduces a left-shifting method for the remaining unadjusted forward passes, strategically inserting them alternately between backward passes. This transformation forms the Obscura pipeline, ensuring that backward bubbles are fully utilized to effectively conceal recomputation overhead.

Swapping-Aware Recomputation. Obscura leverages activation swapping in adjusted stages to reduce activation footprints before the first backward pass. Furthermore, it models pipeline execution as an optimization problem to derive an optimal recomputation strategy that minimizes both communication and recomputation overhead.

Partition Adjustment. To address computation imbalance, Obscura introduces a stage partitioning adjustment algorithm that redistributes layers from adjusted stages to non-adjusted stages. For instance, adjusted stages are assigned fewer layers, while non-adjusted stages handle more, thereby balancing workloads and improving overall performance.

The primary contributions of Obscura are as follows:

- We identify a novel opportunity to conceal recomputation overhead by utilizing pipeline bubbles and propose a strawman pipeline to achieve overhead concealment.
- We introduce Obscura, which incorporates three key components—Dependency Relaxation, Swapping-Aware Recomputation, and Partition Adjustment—to improve pipeline training performance.
- We implement Obscura on DeepSpeed, demonstrating a throughput improvement of up to 1.32× compared to widely used recomputation strategies on NVIDIA GPUs.

2 Background and Related Work

2.1 Pipeline Parallelism

Pipeline parallelism [5, 8, 15] partitions model layers into distinct stages distributed across multiple GPUs. The input batch is divided into smaller micro-batches, which are processed sequentially in a pipelined manner. Communication between

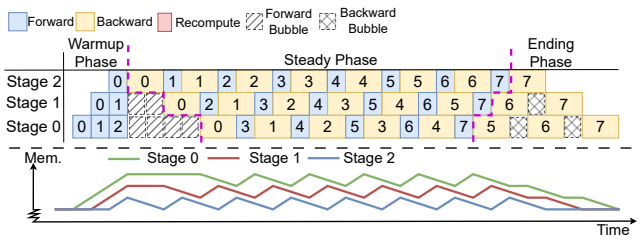


Figure 2: Scheduling of the 1F1B pipeline.

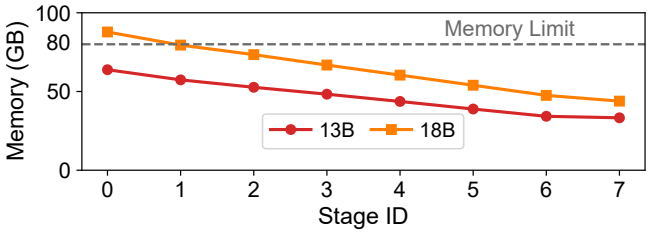


Figure 3: Memory usage across stages during the training of 13B and 18B Llama-2 models in the 1F1B pipeline. Results for the 18B model are estimated.

stages is required before and after each forward or backward pass to transfer activations and gradients.

GPipe [8] processes the forward passes of all micro-batches sequentially before initiating the backward passes. This scheduling approach requires storing activations for all micro-batches, resulting in significant GPU memory usage.

In contrast, the 1F1B pipeline [5] reduces memory overhead by initiating backward passes earlier, effectively avoid memory usage of activations accumulation from all micro-batches. Its schedule, shown in Figure 2, comprises three phases: warmup, steady, and ending. The warmup phase spans from the first forward pass to the start of the first backward pass, with memory usage rising linearly as only forward passes are executed. The steady phase, the core of 1F1B, alternates backward and forward passes, reusing memory freed by backward passes to store activations from forward passes, ensuring stable memory usage. The ending phase mirrors the warmup phase in reverse. Additionally, we define two types of bubbles: forward bubbles (diagonal blocks), occurring between the last forward and first backward passes, and backward bubbles (crossed-line blocks), occurring between the first and last backward passes.

To reduce pipeline bubbles, several scheduling strategies [1,14–16,26] based on the 1F1B schedule have been proposed, though they often increase memory pressure or affect convergence. For example, Chimera [14] reduces bubbles through bidirectional scheduling but at the cost of higher memory consumption. PipeDream [15] eliminates bubbles by overlapping the forward pass of the second micro-batch with the backward pass of the first, but it faces challenges with convergence.

Although the 1F1B pipeline offers better memory efficiency than GPipe, it faces a memory bottleneck in the early stages [10,29]. Specifically, stage s out of p stages must store activations for up to $p-s$ micro-batches. As a result, earlier stages ($s \to 0$) experience higher memory usage. As shown in Figure 3, for the 13B model, the memory usage in stage 0 is significantly higher than in stage 7, with a gap of up to 24 GB. For the 18B model, the memory usage in stage 0 exceeds the GPU memory limit (OOM), causing training failures.

2.2 Recomputation

A straightforward approach to reducing the memory usage of activations is the recomputation technique, which is incorporated into various parallel strategies in modern frameworks like DeepSpeed [18] and Megatron [20].

Formally, a pipeline stage consists of a set of operators, each producing and storing activations during the forward pass. The recomputation technique reduces memory usage by discarding a portion of these activations during the forward pass and recomputing them during the backward pass. Although this approach effectively lowers memory consumption, it introduces additional computation overhead, as the discarded activations must be generated by re-executing the corresponding operators in the backward pass.

Various studies [7,9,11,22,27] have investigated strategies to minimize recomputation overhead within specific memory constraints. These works leverage the varying computation and space complexities of operators to identify an optimal recomputation strategy, consisting of cost-effective operators with high memory usage but low recomputation overhead. However, relying solely on recomputation for memory savings is highly model-dependent and inevitably introduces substantial overhead, particularly under strict memory limits.

To address this limitation, several works [2,17,24,28] combine offloading with recomputation, utilizing tensor liveness analysis and dynamic memory states to decide whether to swap data out or discard it. While these techniques reduce recomputation overhead through swapping, they are constrained by PCIe bandwidth, often introducing significant communication overhead. Additionally, they may interfere with the communication required for pipeline parallelism [17] and, in some cases, incur extra CPU computation latency [19].

BPipe [10] introduces a pipeline-specific approach that leverages high-bandwidth inter-GPU connections to transfer activations from earlier stages to the spare memory of later stages, alleviating memory bottlenecks. Additionally, it incorporates selective recomputation [11] to further reduce memory usage. However, the limited spare memory available in later stages constrains the effectiveness of BPipe's swapping technique, necessitating a greater reliance on recomputation under stringent memory constraints. Due to its lack of optimized integration between inter-GPU swapping and recomputation, BPipe still incurs significant recomputation overhead.

In summary, recomputation overhead remains a significant challenge. Our work, Obscura, identifies an opportunity to

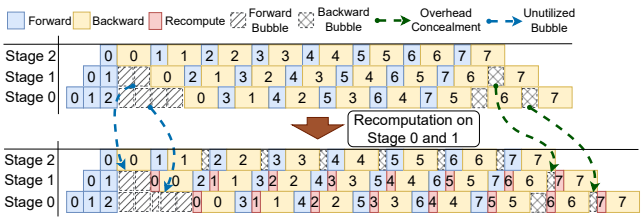


Figure 4: Observations in On-Demand Recomputation.

hide recomputation overhead within pipeline bubbles and introduces a novel pipeline schedule designed to better conceal this overhead. This approach also enables a more effective integration of recomputation and swapping, achieving efficient memory savings. To our knowledge, we are the first to explore reducing overhead by strategically concealing recomputation.

3 Motivation

3.1 Observations in 1F1B Pipeline Schedule

Existing works typically implement recomputation across all stages (referred to as *All-Stage Recomputation*) when the pipeline encounters OOM errors. However, as discussed in §2.1, the memory bottleneck is primarily concentrated in the early stages, where only a subset exceeds the memory limit while others remain underutilized. Therefore, recomputation should be selectively applied to the early stages that surpass the memory limit. We refer to this approach as *On-Demand Recomputation* and derive the following observations.

Observation 1: On-Demand Recomputation Achieves Better Performance than All-Stage Recomputation. We evaluate the execution time of various on-demand recomputation pipelines with four stages. The *Full* recomputation strategy discards nearly all activations and performs an extra forward pass during the backward pass. As shown in Figure 5(a), *Recomp-N* denotes recomputation applied to stages with $s \leq N$, *Recomp-All* represents recomputation applied to all stages, and *Recomp-Non* indicates no recomputation. By comparing stage 0 across all configurations, we observe that *Recomp-0* and *Recomp-1* increase execution time by 19% and 25%, respectively, compared to *Recomp-Non*, outperforming *Recomp-All*, which increases execution time by 33%.

To investigate the benefits of on-demand recomputation, we dive into the 1F1B pipeline schedule in Figure 4 with recomputation selectively applied to stages 0 and 1 (*Recomp-1*), leading to another critical observation.

Observation 2: Backward Bubbles Can Conceal Recomputation Overhead. As illustrated in Figure 4, after applying *Recomp-1*, the backward bubbles in stage 1 are largely filled by the time extension caused by the recomputation of micro-batch 6, preventing it from impacting subsequent executions. In contrast, micro-batches 0–5, which lack sufficient backward bubbles to absorb the recomputation overhead, cause the

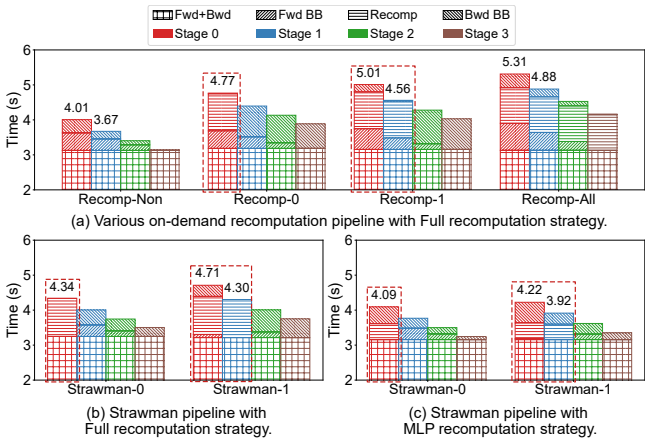


Figure 5: Computation and bubble times for a single iteration. *Fwd+Bwd* denotes the total forward and backward time; *Fwd BB* and *Bwd BB* represent forward and backward bubble times, respectively; and 'Recomp' indicates the recomputation time.

overall execution to shift rightward, extending the pipeline. A similar pattern is observed for the recomputation of both micro-batch 6 and micro-batches 0–5 in stage 0.

To validate this finding, we analyze the changes in computation and bubble times shown in Figure 5(a). Comparing *Recomp-Non* with *Recomp-0* in stage 0 reveals that backward bubbles are replaced by recomputation task. A similar phenomenon is observed in stages 0 and 1 of *Recomp-1*. In contrast, the backward bubbles in *Recomp-All* remain unaffected because the last stage lacks backward bubbles and all stages incur the same recomputation time, making the last stage the slowest. This delay then propagates through inter-stage dependencies, preventing earlier stages from utilizing their backward bubbles.

Although backward bubbles can partially conceal recomputation overhead, their limited availability allows them to hide only a small portion of it. To explore opportunity for increasing backward bubbles, we present the third observation.

Observation 3: Forward Bubbles Are Unutilized. As illustrated in Figure 4, forward bubbles remain unused in recomputation stages. This is because forward bubbles are generated before the first backward pass, whereas recomputation inherently occurs during the backward pass. Furthermore, the duration of forward bubbles is significantly longer than that of backward bubbles. In a p-stage pipeline, stage s contains $2(p-s-1)$ forward bubbles, which is double the number of backward bubbles $(p-s-1)$.

In Figure 5(a), compared to *Recomp-Non*, the forward bubbles in stage 0 of *Recomp-0* remain unchanged, with the bubble time being longer than the backward bubble time (0.50s vs. 0.37s, not a double relation due to synchronized communication between stages). Similar trends are observed in *Recomp-1* and *Recomp-All*, where the forward bubbles are slightly extended. This occurs because the recomputation in stage s delays the initiation of the first backward pass in stage

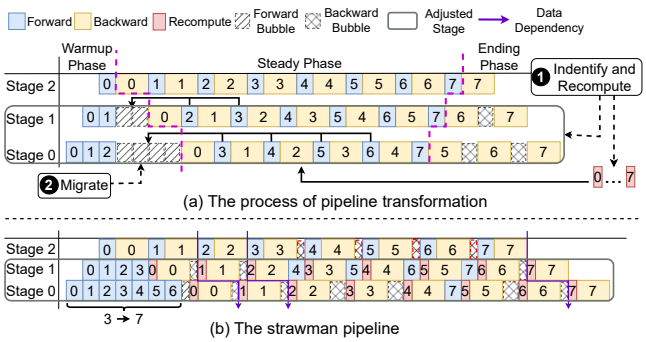


Figure 6: Pipeline transformation process (top) and the resulting strawman pipeline (bottom).

$s-1$, thereby increasing the duration of forward bubbles.

3.2 A Strawman Pipeline: Optimization Opportunity of Recomputation Overhead Concealing

Building on our three key observations, we propose converting forward bubbles into backward bubbles to more effectively conceal recomputation overhead. To implement this concept, we introduce a strawman pipeline, derived from the 1F1B pipeline using a pipeline transformation, as illustrated in Figure 6(a). This transformation consists of two key steps:

Step 1: Identify and Recompute. In the 1F1B pipeline, stages that exceed the memory limit are identified as "adjusted stages" and are subjected to recomputation. The remaining, referred to as "non-adjusted stages," are left unchanged.

Step 2: Migrate. For adjusted stages, the forward passes in the steady phase are migrated leftward into forward bubbles as much as possible, causing the forward bubbles to shift rightward and convert into backward bubbles.

Figure 6(b) illustrates the proposed strawman pipeline, whose performance is evaluated in Figure 5(b). The term *Strawman-N* corresponds to the same meaning as *Recomp-N*. Compared to *Recomp-0* and *Recomp-1*, the recomputation overhead in *Strawman-0* and *Strawman-1* is reduced to 8% and 17%, respectively. This is achieved by eliminating forward bubbles and replacing them with recomputation tasks.

In practice, instead of applying the *Full* recomputation strategy to the adjusted stage, a selective strategy that discards only partial activations to precisely meet memory constraints proves more effective. This approach results in reduced recomputation overhead and improved overhead concealment. However, it introduces three key challenges:

Challenge 1: How to Fully Utilize Backward Bubbles. The tight data dependency between adjusted and unadjusted stages—where micro-batches are executed sequentially in the forward pass and in reverse during the backward pass—restricts the ability of backward bubbles to conceal recomputation. Backward bubbles can only hide preceding recomputation, not subsequent ones. For example, as depicted in Figure 6(b), the recomputation for micro-batch 1 cannot utilize the preceding bubble because it must wait for the backward pass in stage 2 to finish. Furthermore, when a selective recomputation strategy is employed, the recomputation time per micro-batch becomes shorter than the bubble time, leading to underutilized backward bubbles. Using the *MLP* recomputation strategy, which discards only *MLP* activations, Figure 5(c) demonstrates that, compared to *Strawman-0* in Figure 5(b), a significant portion of the previously fully utilized backward bubbles is now left unused.

Table 1: Memory usage across stages in different pipelines.

	Recomp-Non	Recomp-0	Recomp-1	Strawman-0	Strawman-1
Stage 0	48.01	34.40	34.40	**46.18**	**46.18**
Stage 1	42.62	42.62	32.25	42.62	**39.58**
Stage 2	36.72	36.72	36.72	36.72	36.72
Stage 3	30.92	30.92	30.92	30.92	30.92

Challenge 2: How to Reduce Extra Memory Usage. Compared to the 1F1B pipeline, the adjusted stages in the strawman pipeline store activations for more forward passes during the warmup phase. As a result, the strawman pipeline exhibits higher memory usage, which can lead OOM errors. Table 1 presents the memory usage across stages under the *MLP* recomputation strategy. The results clearly show that the memory usage of the adjusted stages in the strawman pipeline is significantly higher than that of the 1F1B pipeline.

Challenge 3: How to Address the Computation Imbalance. On-demand recomputation introduces workload imbalances between adjusted and non-adjusted stages. While forward and backward bubbles can partially conceal recomputation, their limited availability and the growing overhead with larger micro-batches leave some recomputation unconcealed. This leads to the formation of additional backward bubbles in the non-adjusted stages, as clearly demonstrated in Figure 6(b). Specifically, as illustrated in Figures 5(b) and 5(c), backward bubbles caused by computation imbalance are evident in the non-adjusted stages.

4 Overview of Obscura

To address above challenges, we propose Obscura, a training system with an architecture illustrated in Figure 7. Obscura is composed of two primary modules: the Obscura Planner and the Obscura Runtime. The Obscura Planner transforms the original 1F1B pipeline into the Obscura pipeline and generates a deployment configuration for use by the Obscura Runtime. The Obscura Runtime manages distributed deployment and model training during execution.

The Obscura Planner comprises four submodules: Pipeline Transformation, which converts forward bubbles into backward bubbles to better conceal recomputation overhead; Dependency Relaxation, which relaxes tight data dependencies between stages to fully utilize backward bubbles; Swapping-

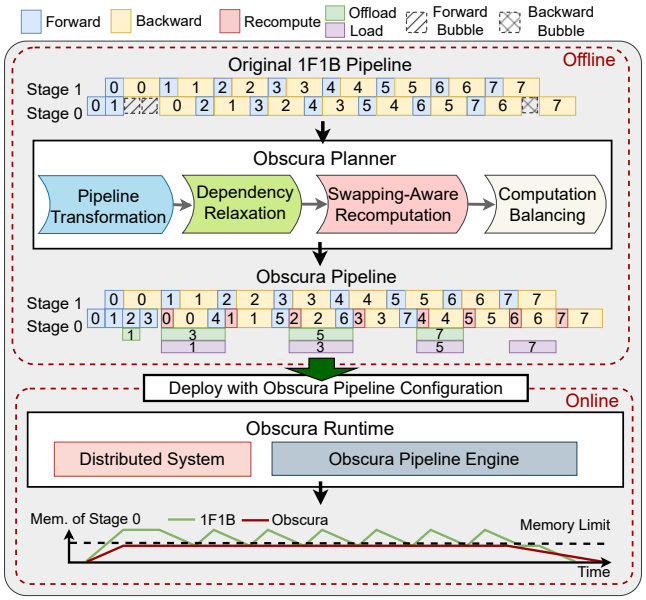


Figure 7: System Architecture of Obscura.

Aware Recomputation, which combines swapping and recomputation to efficiently adhere memory constraints; and Partition Adjustment, which refines the stage partition strategy to eliminate bubbles in non-adjusted stages. The Obscura Runtime consists of the Distributed System, which enables distributed training, and the Obscura Pipeline Engine, which implements the core runtime of the Obscura pipeline.

Notably, the Obscura Planner operates offline to generate the Obscura configuration using only a few training iterations. Specifically, the pipeline transformation submodule runs for several iterations to determine the adjusted stages and collect runtime profiling data, including the computation and memory costs of operators. The remaining submodules use this profiling data to solve small-scale optimization problems and execute lightweight algorithms, introducing negligible overhead. In addition, the Obscura Runtime operates online and manages system behavior during training.

5 Dependency Relaxation

To fully utilize backward bubbles, we first perform an in-depth analysis of computation and bubbles in adjusted stages. Based on this analysis, we propose dependency relaxation, which re-purposes unused backward bubbles for other computations. Additionally, we refine the migration of forward passes to minimize the additional activations footprints caused by unnecessary forward pass migrations.

5.1 Analysis of Computation and Bubbles

We illustrate the simplified steady phase of adjacent adjusted and non-adjusted stages in Figure 8. In Figure 8(a), the left side shows the underutilization of backward bubbles after migration, while the right side depicts the remaining forward passes. Due to tight data dependencies between the two stages, backward passes cannot shift leftward to utilize left-side bubbles. Specifically, the backward pass of micro-batch x in stage s must wait for the completion of the backward pass of the same micro-batch in stage $s+1$, preventing the use of preceding bubbles for recomputation. Forward passes encounter a similar trouble: they cannot shift rightward to utilize right-side bubbles (if any), as this would delay the forward passes of the same micro-batch in later stages, extending the pipeline.

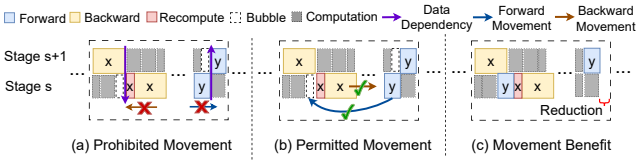


Figure 8: The simplified steady-phase schedule for adjacent adjusted and non-adjusted stages.

However, the opposite movements to exploit bubbles are allowed. As illustrated in Figure 8(b), the forward pass of micro-batch y in stage s can shift leftward to utilize the left-side bubbles without violating its data dependency. Similarly, the backward pass of micro-batch x can shift rightward. This observation provides a key insight: instead of relying solely on adjacent recomputation tasks to utilize unused backward bubbles, we can shift the remaining forward passes leftward to exploit these bubbles. Subsequently, the backward passes can be shifted rightward to take advantage of the bubbles created by the adjusted forward passes.

Figure 8(c) illustrates the outcome of this adjustment. The bubbles preceding the backward pass of micro-batch x in stage s are utilized, resulting in a reduction in execution time. This demonstrates that the previously unused backward bubbles are effectively utilized to conceal recomputation overhead.

5.2 Dependency Relaxing among Stages

Based on the analysis, we propose dependency relaxation to improve backward bubble utilization, as shown in Figure 9. The key adjustment involves shifting the remaining forward passes leftward and the backward passes rightward during the steady phase. To prevent excessive activation accumulation, the left-shifted forward passes are interleaved with the backward passes, similar to the 1F1B approach. For example, the forward passes of micro-batches 4–7 are shifted leftward and orchestrated in an interleaved manner with the corresponding backward passes, which are shifted rightward accordingly.

This adjustment relaxes the tight data dependencies between stages, ensuring that the execution of one stage no longer directly impacts another. As a result, underutilized backward bubbles are effectively leveraged, reducing overall iteration time, as shown in the middle of Figure 9.

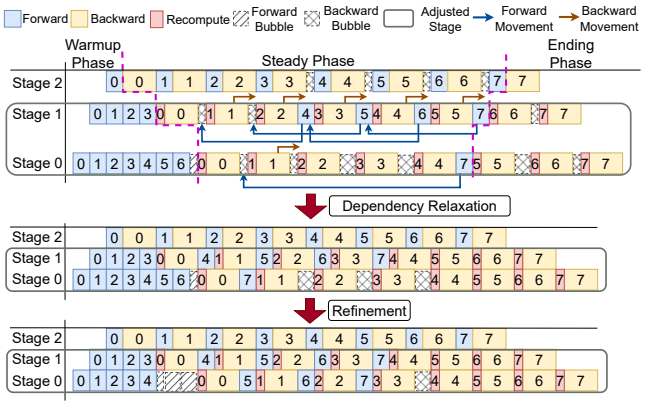


Figure 9: Two-step optimization process applied to the strawman pipeline to achieve Obscura pipeline.

5.3 Migration Refinement

We refine the pipeline by reducing the forward passes migrated into forward bubbles, resulting in the Obscura pipeline depicted at the bottom of Figure 9. This refinement decreases the activations footprints during the warmup phase without affecting the concealment of recomputation overhead. It is motivated by the persistence of backward bubbles in stage 0, as illustrated in the middle of Figure 9.

In a p-way pipeline, stage s can have up to $\min(m-1, 3(p-s-1))$ backward bubbles, where m denotes the total number of micro-batches. This suggests that earlier adjusted stages can theoretically conceal more recomputation overhead. However, due to data dependencies in the final backward pass of adjusted stages, the total time extension across all adjusted stages cannot recede that of the last adjusted stage, k. From this analysis, we conclude that the ability to conceal recomputation overhead is determined by the number of bubbles in stage k. This also accounts for the unused bubbles observed in earlier adjusted stages after dependency relaxation. Therefore, for adjusted stages where $s < k$, it is sufficient to align the number of backward bubbles with those in stage k.

6 Swapping-Aware Recomputation

To reduce the extra memory usage caused by the migration of forward passes, we introduce an activation swapping scheme for the Obscura pipeline, which reduces activation storage during the warmup phase. Furthermore, we analyze the trade-off between recomputation and communication overhead, formulating an optimization problem to determine the optimal recomputation strategy under a given swapping configuration.

6.1 Activation Swapping

Relying solely on increased recomputation relative to the 1F1B pipeline to eliminate extra memory consumption introduces more recomputation overhead, which conflicts with our objective of utilizing bubbles to conceal recomputation overhead. To resolve this, we incorporate swapping techniques into Obscura pipeline to alleviate recomputation pressure.

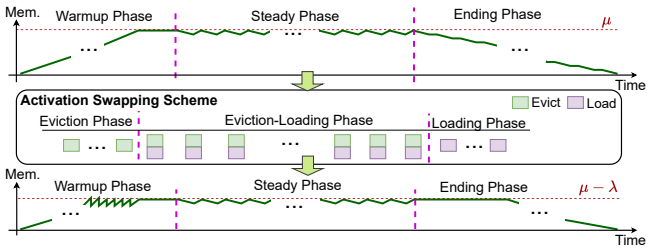


Figure 10: The activation swapping scheme.

Activation Swapping Scheme. We propose an activation swapping scheme, transferring activations to external memory at the granularity of individual micro-batches, termed "activation blocks," as shown in Figure 10. The scheme comprises three swapping phases: eviction, eviction-loading, and loading. The eviction phase occurs during the warmup phase, where memory usage rises linearly, requiring continuous eviction of activation blocks to maintain target memory levels. The eviction-loading phase operates throughout the steady phase, during which evicted activations are reloaded for backward passes. To avoid exceeding memory limits, one activation block must be evicted before reloading. Leveraging full-duplex communication, eviction and loading can be performed in parallel. The loading phase occurs in the ending phase, where activations are loaded without eviction as backward passes consume them, ensuring stable memory usage.

The various swapping implementations [10, 27] are specific instances of this scheme. By abstracting their details and focusing on the core aspects, we integrate the scheme into the Obscura pipeline's optimization, ensuring compatibility with diverse designs.

Memory Analysis. In the activation swapping scheme, memory usage is determined by two critical parameters: 1) λ, the number of activation blocks transferred in the eviction phase, and 2) β, the fraction of data transmitted per activation block. Let μ^{adj} denote the number of forward passes in the warmup phase of the adjusted stage, and a^{adj} represent the size of activations stored per micro-batch of the adjusted stage. The memory consumption of activations in adjusted stage s after swapping can be expressed as follows.

$$A_s^{adj} = (\mu_s^{adj} - \lambda_s)a^{adj} + (1-\beta)\lambda_s a^{adj} \quad (1)$$

6.2 Swapping-Aware Recomputation

Recomputation and activation swapping in Obscura create a trade-off between recomputation and communication costs. **1)** Increasing recomputation, as shown in Figure 11(a), reduces the number of activation blocks evicted during warmup to meet memory limits, and allows shorter transfer times to over-

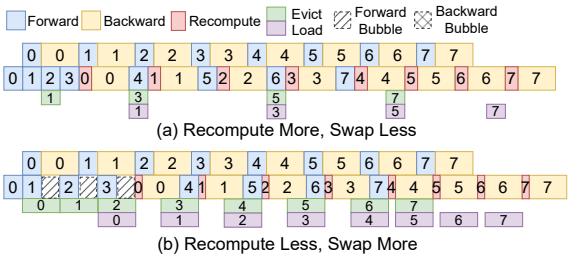


Figure 11: Trade-off Between Recomputation and Swapping.

lap with computation but incurring higher recomputation overhead. **2)** Decreasing recomputation, as in Figure 11(b), lowers recomputation overhead, which bubbles can mostly hide, but requires more swapping to satisfy memory constraints. This includes evicting more activation blocks during warmup and longer transfer times for each activation block that cannot overlap with computation, resulting in significant communication overhead. To balance recomputation and communication overhead, we model the execution time and memory usage of Obscura pipeline and construct our optimization problem.

Execution Time. We consider the p-way 1F1B pipeline with m micro-batches under even stage partition for LLMs, where the forward time f and backward time $2f$ is identical across stages. The recomputation time is denoted as r, and the last adjusted stage is denoted as stage k.

For swapping, as shown in Figure 11(b), eviction or loading operations have maximum transmission times of f, $3f+r$, and $2f+r$ in the three swapping phases, respectively. Exceeding these thresholds introduces communication overhead, with the eviction-loading phase's cost scaling with the number of micro-batches. To mitigate significant overhead, we enforce the constraint $a^{adj}\beta \leq (3f+r)\times B$, where B is the CPU-GPU communication bandwidth. Thus, the communication cost is

$$\begin{aligned} T_s^C = &\max(\lambda_s(a^{adj}\beta/B - f) - (k-s)(2f+r), 0) + \\ &\lambda_s(a^{adj}\beta/B - 2f - r) + \varepsilon \end{aligned} \tag{2}$$

Eq.2 applies when stage k has no remaining bubbles, as their presence implies that the overhead of *Full* recomputation can be fully hidden, making swapping unnecessary. The first and second terms represent the communication cost incurred in the warmup and ending phases, respectively, with the former having bubbles to mitigate the cost. ε represents the overhead not covered by the first two terms due to variations in swapping design [10, 27], with a range of $[0, 3f+r]$.

The execution time, comprising computation time and bubble time, is expressed as follows:

$$T_s = \begin{cases} 3mf + 3(p-s-1)f, & s > k \\ \max(T_{s+1} + \gamma, \gamma m + (k-s)\gamma + T_s^C), & s \leq k \end{cases} \tag{3}$$

where $\gamma = 3f+r$. T_s denotes the pipeline execution time from stage s to the final stage, primarily determined by computation, bubble delays, and communication overhead. The first term in the max function captures inter-stage data dependencies.

Memory Usage. We define a as the activation size per micro-batch without recomputation. The memory usage of stage s is $M_s = W_s + A_s$, where W_s is the size of model parameters, including optimizer states, and A_s is the memory usage of activations. For adjusted stages, $A_s = A_s^{adj}$; for non-adjusted stages, $A_s = (p-s)\times a$.

The Optimization Objective. Based on the analysis and modeling above, we formulate the optimization problem as

$$\begin{aligned} &min\ T_0 \\ &s.t.\ M_s \leq G, s \in [0, p-1] \\ &a^{adj}\beta \leq (3f+r)\times B \end{aligned} \tag{4}$$

The optimization variables include λ_s, β, a^{adj}, and r. λ_s follows a fixed relationship across stages: stage $s-1$ evicts one more activation block than stage s. Thus, λ_s can be represented as λ_k, with a range of $[2, p-k-2]$. β is a percentage in $[0,1]$. a^{adj} and r correspond the recomputation strategy, the set of operators selected for recomputation.

We quantify the recomputation strategy as follows: We define the operators of pipeline stage as $U_{op} = \{op_0, ..., op_{n-1}\}$, where n represent the total number of operators. Then we have $f \approx \sum_{op_i \in U_{op}} f_{op_i}$ and $a \approx \sum_{op_i \in U_{op}} a_{op_i}$, where f_{op_i} and a_{op_i} denote the computation time and the activation size of op_i. The recomputation strategy is represented as $R_{op}^{0-1} = [x_0, x_1, \ldots, x_{n-1}]$, a binary array where $x_i = 1$ indicates that op_i is recomputed, and $x_i = 0$ indicates it is not. Then, we have $r = \sum_{i=0}^{n-1} R_{op}^{0-1}[i] \times f_{op_i}$ and $a^{adj} = a - \sum_{i=0}^{n-1} R_{op}^{0-1}[i] \times a_{op_i}$.

By replacing a^{adj} and r in Eq.4 with the quantified definition, the optimization problem becomes an integer programming (IP) problem. We solve it by enumerating possible values of the swapping-related variables to derive the corresponding recomputation strategy, then selecting the combination of the swapping configuration and recomputation strategy that achieves the best performance. For instance, if we adopt BPipe's [10] swapping design for our activation swapping scheme, we only need to iterate over λ_k, as β is fixed at 1.

7 Partition Adjustment

To balance computation between adjusted and non-adjusted stages, we first refine the execution time and memory usage model of the Obscura pipeline, accounting for uneven stage partitioning. Based on this model, we propose a stage partition adjustment algorithm to optimize pipeline performance by identifying the optimal partitioning strategy.

7.1 Modeling in Uneven Stage Partition

Considering a pipeline with uneven stage partition, we define f_s, r_s, and a_s as the forward time, recomputation time,

and activation size per micro-batch for stage s, respectively. In pipeline parallelism, execution time is dominated by the slowest stage. We let $C_{adj} = \max_{s \le k}(3f_s + r_s)$ and $C_{non} = \max_{s>k}(3f_s)$ represent the total computation time per micro-batch for the slowest adjusted and non-adjusted stages, respectively. Thus, the execution time for stage s in Obscura pipeline with uneven stage partition can be expressed as follows.

$$\hat{T}_s \approx \begin{cases} T_s(C_{non}), & s > k \\ T_s(C_{adj}), & s \le k \end{cases} \tag{5}$$

Eq.5 is an approximation, where C_{non} and C_{adj} replace the terms $3f$ and $3f + r$ in T_s, respectively. For memory usage, we replace a and a^{adj} in M_s with a_s, as expressed below.

$$\hat{M}_s = M_s(a_s) \tag{6}$$

7.2 Partition Adjustment Algorithm

To balance computation between adjusted and non-adjusted stages, we propose a partition adjustment algorithm that transfers layers from adjusted to non-adjusted stages. For finer-grained adjustments, each transformer layer is split into an attention layer and an MLP layer, resulting in $2L+2$ layers, where L is the number of transformer layers, and the constant 2 accounts for the embedding and decoder head layers. In the even stage partition, the first and last stages contain $2L/p+1$ layers, while intermediate stages contain $2L/p$ layers.

Significant computation imbalance in the Obscura pipeline occurs only with a large number of micro-batches, where slight partition adjustments can eliminate the imbalance. For simplicity, the impact of adjustments on overhead concealment and swapping-aware recomputation is ignored.

The partition adjustment algorithm, in Algorithm 1, takes dictionaries D_{adj} and D_{non} representing adjusted and non-adjusted stages and their layer counts. The goal is to make $\hat{T}_{k+1}$ as close as possible to $\hat{T}_k - C_{adj}$, balancing computation between stages. Lines 5-8 transfer one layer from the slowest adjusted stage a to the fastest non-adjusted stage b. The algorithm outputs the updated dictionaries D_{adj} and D_{non}.

Algorithm 1: Partition Adjustment

Input: D_{non} and D_{adj}
Output: Updated D_{non} and D_{adj}.

```
1  Initialize old_D_non ← D_non, old_D_adj ← D_adj
2  Compute T̂_k, T̂_{k+1}, C_adj based on D_non, D_adj
3  while T̂_k − C_adj > T̂_{k+1} do
4      old_D_non, old_D_adj ← D_non, D_adj
5      a ← arg max_{s≤k}(3f_s + r_s)
6      b ← arg min_{s>k ∧ M̂_s ≤ G}(3f_s)
7      D_adj[a] ← D_adj[a] − 1
8      D_non[b] ← D_non[b] + 1
9      Update T̂_k, T̂_{k+1}, C_adj based on D_non and D_adj
10 D_non, D_adj ← old_D_non, old_D_adj
```

8 Design Reinforcement: Enhancing Concealability in Identifying Adjusted Stages

Problem. As discussed in §5, on one hand, the number of backward bubbles in a stage is inversely proportional to the stage ID, and the maximum concealable overhead is determined by the last adjusted stage. As a result, selecting more stages as adjusted stages reduces the Obscura pipeline's ability to conceal recomputation overhead, eventually degrading its performance to that of the 1F1B pipeline. On the other hand, identifying adjusted stages during pipeline transformation is guided by memory usage and hardware memory capacity. Under conditions of high memory usage or limited memory capacity, a larger number of stages are designated as adjusted, decreasing the performance of the Obscura pipeline.

Opportunity. Recent studies [27] reveal that cost-effective operators in transformer models, such as *RMSNorm*, *SiLU*, and *Mul* in the Llama-2 model, exhibit high space complexity but low computation complexity. Including these operators in the recomputation set (referred to as the *CMB* recomputation strategy) can reduce footprints by approximately 40%, with only a modest 2–3% increase in overhead.

Optimization. To enhance the original identification step, we introduce an alternative approach called **CMB-Identifying**. When encountering OOM issues, instead of immediately designating a stage as adjusted, the *CMB* recomputation strategy is applied first. This method reduces the number of adjusted stages, thereby improving the Obscura pipeline's ability to conceal recomputation overhead.

9 Evaluation

We evaluated Obscura on a single training node. Notably, since Obscura introduces no communication overhead between stages, it can seamlessly scale to multiple nodes.

9.1 Implementation and Experimental Setup

Implementation. We implement Obscura on DeepSpeed by designing a custom scheduler to replace the native scheduler. To enable inter-stage communication within the Obscura pipeline, we convert synchronous NCCL communication to asynchronous and introduce synchronization mechanisms to ensure seamless execution. Swapping operations are integrated into the computation schedule as execution steps, utilizing separate CUDA streams and manual memory management to enable concurrent swapping and memory reuse.

Platform. We evaluate Obscura on a data center training node equipped with 2.0 TB of DRAM, two Intel Xeon Platinum 8352S CPUs (128 threads in total), and eight NVIDIA A100-SXM-80GB GPUs. The GPUs are fully interconnected

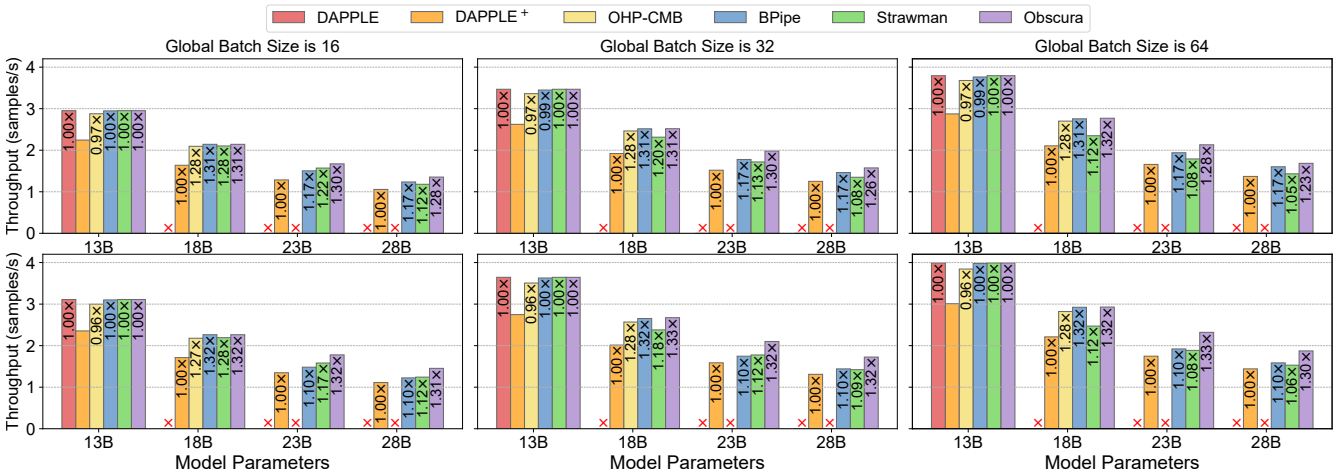


Figure 12: End-to-end performance for training various sizes of Llama-2 and GPT-3 models under different global batch sizes.

via NVLink, providing a bidirectional P2P bandwidth of 600 GB/s, and are connected to the CPUs through PCIe 4.0 x16.

Model, Datasets and Workloads. We evaluate Obscura using various model sizes based on the popular Llama-2 [23] and GPT-3 [3] models, including 13B, 18B, 23B, and 28B parameters for both base models. The GPT-3 models used in our evaluation are based on the open-source implementation from the Megatron-LM repository [11, 16, 20]. To simplify testing, we generate a random sequence before the training loop and feed it into the models. The sequence length is standardized to 4096, and the micro-batch size is set to 1, which are common configurations for multi-node training.

Baselines. We utilize the following pipeline parallelism configurations implemented on DeepSpeed with the 1F1B pipeline as baselines for comparison experiments:

- `DAPPLE` [5]: The state-of-the-art 1F1B pipeline schedule with even stage partition.
- `DAPPLE`$^+$: We evaluate `DAPPLE` using the *Full* recomputation strategy applied to all stages, which reduces memory usage and enables the training of larger models.
- `OHP-CMB`: We apply the recomputation strategy proposed in the work [27] that drops activations of specific operators, including *Norm*, *SiLU/GELU*, and *Mul*.
- `BPipe` [10]: It eliminates the memory bottleneck by transferring activations from earlier stages to later ones using an activation balancing method. As it encounters OOM issues with 23B and 28B models, we apply the *MLP* recomputation strategy for these larger models.

In addition, we measure `Strawman` with the *Full* recomputation strategy applied for the adjusted stages.

9.2 Training Performance

We evaluate the throughput of Obscura and baselines using Llama-2 13B and GPT-3 13B as base models. To analyze Obscura's performance across different scenarios, we scale the model by adding layers and extend the pipeline length by increasing the global batch size. The performance speedups of Obscura and the baselines relative to DAPPLE (for the 13B model) and DAPPLE$^+$ (for models of other sizes) is presented in each column of Figure 12. Notably, Strawman and Obscura are **NOT** applied to the 1F1B pipeline in the 13B model.

Llama-2. Figure 12 illustrates that for the 13B model, DAPPLE operates effectively and significantly outperforms DAPPLE$^+$, which incorporates recomputation. However, as the parameter size increases, DAPPLE fails to train due to excessive memory usage resulting from retaining all activations. In contrast, DAPPLE$^+$ continues training successfully by discarding most activations to reduce memory usage.

OHP-CMB leverages recomputation for cost-effective operators, achieving a computation complexity of $O(bsh)$ for the operators' recomputation and a space complexity of $O(bs^2h)$ or $O(bsh^2)$ for reducing the operators' footprints. This strategy allows OHP-CMB to efficiently train 13B and 18B models with minimal recomputation overhead. However, it cannot support larger models due to limited activation savings.

BPipe utilizes spare memory in later stages to mitigate the memory bottleneck in earlier stages, enabling the training of 13B and 18B models. Furthermore, the high-bandwidth NVLink ensures minimal communication overhead, allowing BPipe to sustain high performance. However, as all activations are retained in GPU memory, the memory usage exceeds GPU capacity for 23B and 28B models. To address this, the *MLP* recomputation strategy is applied, and BPipe achieving just 17% improvements over DAPPLE$^+$ for these larger models because of these recomputation overhead.

Strawman outperforms DAPPLE$^+$ by effectively concealing recomputation overhead. However, its performance deteriorates with increasing model sizes and global batch sizes. Larger models reduce Strawman's ability to conceal recomputation overhead due to the need for more stages to be adjusted stages, while larger global batch sizes increase recomputation demands, further intensifying the overhead.

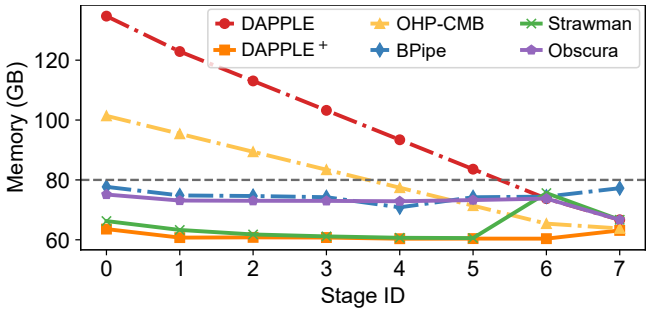


Figure 13: Memory usage in the Llama-2 28B model.

Obscura requires fewer adjusted stages than Strawman by using the **CMB-Identifying** and reduces the recomputation of non cost-effective operators through activations swapping. Therefore, Obscura more effectively conceal recomputation overhead using forward and backward bubbles, even with larger models and global batch sizes. Specifically, Obscura achieves speedups of 29–31% for the 18B and 23B models, 22–27% for the 28B model, and a 22% speedup in the extreme case of a global batch size of 64 for the 28B model.

GPT-3. Obscura and the baselines achieve comparable end-to-end performance to Llama-2. But notably, since GPT-3 uses the *GELU* activation function, which is more cost-effective than the *SiLU* function used in Llama-2 in recomputation, Obscura attains speedups of 32–33% for the 18B and 23B models, 30–32% for the 28B model, and 30% in the extreme case of a global batch size of 64 for the 28B model.

9.3 Memory Analysis

We measure peak memory usage across stages for the Llama-2 28B model with a global batch size of 32. For baselines experiencing OOM, we estimate their memory usage to ensure a fair comparison. The results are presented in Figure 13.

For DAPPLE, only stages 6 and 7 operate within the memory constraints. Memory usage across stages shows an imbalanced linear decline, with stage 0 consuming approximately 2.0× the memory of stage 7. In contrast, DAPPLE$^+$ achieves consistently low and balanced memory usage by discarding nearly all activations across all stages, leaving approximately 20 GB of unused memory in each stage. OHP-CMB reduces activation size by 40% compared to DAPPLE by recomputing only the *Norm*, *SiLU*, and *Mul* operators; however, this optimization is insufficient for training the 28B model. BPipe balances memory by retaining at most $\lceil (p+2)/2 \rceil$ micro-batches of activations across p stages. Without the *MLP* recomputation strategy, its peak memory usage reaches 106 GB during 28B model training, exceeding memory limits. With the strategy applied, memory usage stabilizes at approximately 76 GB. Since only stages 6 and 7 in DAPPLE remain within the memory constraints, Strawman selects stages 0–5 as adjusted stages, resulting in memory usage similar to DAPPLE$^+$, where available memory is not fully utilized.

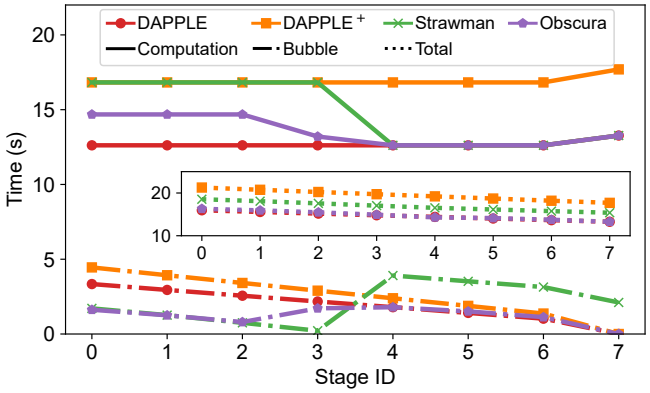


Figure 14: Computation, bubble and total execution time across all stages in the Llama-2 23B model.

In contrast, Obscura selects stages 0–4 as adjusted stages and applies the *CMB* recomputation strategy to stage 5. As a result, stage 5 shows memory usage comparable to OHP-CMB, while stages 6 and 7 align with DAPPLE. For the remaining stages, the identical recomputation strategy and objectives for activation swapping ensure memory usage that close to the constraints, achieving high memory utilization.

9.4 Computation and Bubble Analysis

To assess the effectiveness of recomputation overhead concealment in Obscura, we measure the computation and bubble times across stages for a single iteration of the Llama-2 23B model with a global batch size of 32. Computation time is defined as the cumulative forward and backward pass times for all micro-batches, while bubble time refers to the total duration of all forward and backward bubbles. The total execution time is calculated as the sum of computation and bubble times. For comparison, we estimate the performance of DAPPLE, despite its inability to run the model at this scale. The results are presented in Figure 14.

Due to the discarding and recomputation of nearly all activations, DAPPLE$^+$ incurs an 1/3 increase in computation, bubble, and execution time compared to DAPPLE. To enable training of the 23B model, Strawman designates stages 0–3 as adjusted stages, resulting in computation time for these stages matching those of DAPPLE$^+$, while stages 4–7 align with DAPPLE. By utilizing bubbles to conceal recomputation overhead, Strawman can reduces bubble time for stages 0–3 compared to DAPPLE, achieving better performance than DAPPLE$^+$. However, Strawman still exhibits longer execution time than DAPPLE, as the available bubble time is insufficient to fully conceal the recomputation overhead, evidenced by the bubble time for stage 3 dropping to zero.

Unlike Strawman, Obscura designates stages 0–2 as the adjusted stages. By utilizing swapping techniques to minimize activation footprints, Obscura employs a recomputation strategy focused solely on cost-effective operators for the

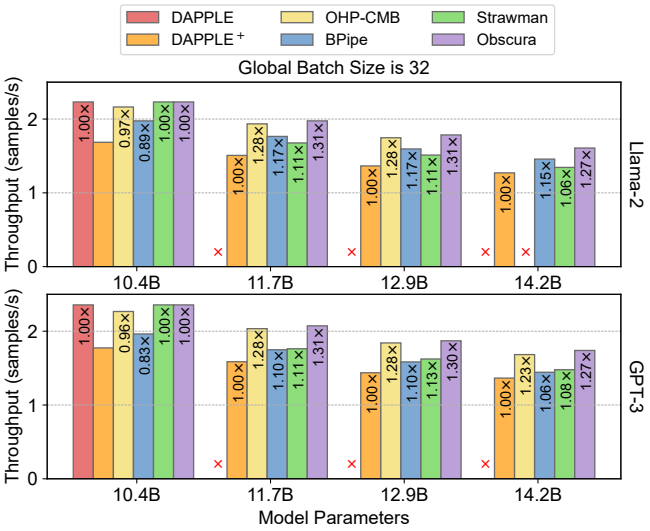


Figure 15: End-to-End performance in 4 stages pipeline.

adjusted stages, leading to only a 13% increase in computation time compared to DAPPLE. In terms of bubble time, the adjusted stages show a reduction of approximately 1.7 seconds compared to DAPPLE. Notably, stage 2 retains bubbles, demonstrating that recomputation overhead is almost entirely concealed. As a result, Obscura achieves performance closely aligned with that of DAPPLE.

9.5 Additional Performance Validation

To demonstrate that Obscura delivers robust performance across diverse training configurations and platforms, we selected another representative 4-stage pipeline configuration for validation. We re-conduct performance evaluations using various sizes of Llama-2 and GPT-3 models with a global batch size of 32. The experiments are performed on a system equipped with four NVIDIA A800 GPUs connected to CPUs via PCIe 4.0 x16, without NVLink. Due to limited P2P bandwidth between GPUs, BPipe achieves 11% and 17% lower throughput than DAPPLE on 10.4B models, mainly due to costly inter-stage activation transfers. For larger models, combining MLP recomputation reduces activation size and enables full overlap of communication and computation, but introduces recomputation overhead. In low-bandwidth settings, recomputation outperforms swapping.

In contrast, Obscura introduces no additional inter-stage communication and maintains original P2P communication between stages. As a result, it consistently achieves strong performance gains of 27–31%.

9.6 Sensitivity Analysis

To evaluate the advantages of Obscura over baseline methods, we perform a sensitivity analysis to examine the impact of dependency relaxation (DR), swapping-aware recomputation

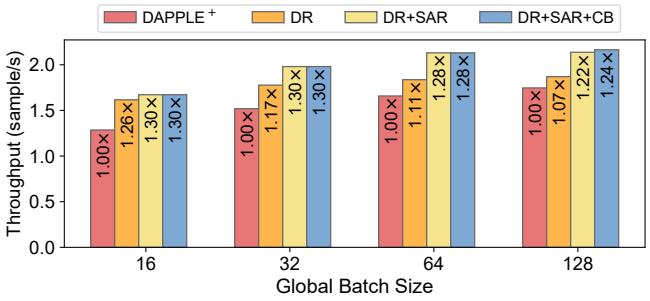


Figure 16: Sensitivity Analysis in the Llama-2 23B model.

(SAR), and computation balancing (CB). Figure 16 presents the performance of DAPPLE$^+$ and Obscura across different combinations of these core techniques, tested under global batch sizes ranging from 16 to 128 for training the Llama-2 23B model. All configurations designate stages 0–2 as adjusted stages, and the results are normalized to DAPPLE$^+$.

At a batch size of 16, bubbles can conceal most recomputation overhead, yielding significant speedups for all Obscura configurations. DR shows a slight performance drop due to recomputation overhead in the final backward pass, which cannot be concealed, and larger per-micro-batch recomputation overhead. At batch sizes of 32 and 64, DR's performance degrades significantly as recomputation overhead exceeds bubbles' capacity. In contrast, DR+SAR and DR+SAR+CB maintain higher performance by prioritizing cost-effective operators, reducing overall recomputation overhead. At a batch size of 128, DR+SAR and DR+SAR+CB experience a performance drop as bubbles cannot fully conceal recomputation overhead. However, DR+SAR+CB mitigates this by redistributing layers between adjusted and non-adjusted stages, balancing the computation load.

10 Conclusions

In this paper, we present Obscura, a computationally efficient pipeline training system, which introduces a novel pipeline transformation that utilizes existing pipeline bubbles to effectively hide recomputation costs, as well as integrates advanced techniques such as dependency relaxation, swapping-aware recomputation, and computation balancing, significantly enhancing overall performance and resource utilization.

11 Acknowledgment

The work described in this paper was supported by the Major Key Project of Peng Cheng Laboratory PCL2023A09, the National Natural Science Foundation of China (62472459, 62172453), and was sponsored by CCF-Huawei Populus Grove Fund (CCF-HuaweiSY202303).

References

[1] Sanjith Athlur, Nitika Saran, Muthian Sivathanu, Ramachandran Ramjee, and Nipun Kwatra. Varuna: scalable, low-cost training of massive deep learning models. In *Proceedings of the Seventeenth European Conference on Computer Systems*, pages 472–487, 2022.

[2] Olivier Beaumont, Lionel Eyraud-Dubois, and Alena Shilova. Efficient combination of rematerialization and offloading for training dnns. *Advances in Neural Information Processing Systems*, 34:23844–23857, 2021.

[3] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[4] Jacob Devlin. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[5] Shiqing Fan, Yi Rong, Chen Meng, Zongyan Cao, Siyu Wang, Zhen Zheng, Chuan Wu, Guoping Long, Jun Yang, Lixue Xia, et al. Dapple: A pipelined data parallel approach for training large models. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 431–445, 2021.

[6] William Fedus, Barret Zoph, and Noam Shazeer. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *CoRR*, abs/2101.03961, 2021.

[7] Julien Herrmann, Olivier Beaumont, Lionel Eyraud-Dubois, Julien Hermann, Alexis Joly, and Alena Shilova. Optimal checkpointing for heterogeneous chains: how to train deep neural networks with limited memory. *arXiv preprint arXiv:1911.13214*, 2019.

[8] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, et al. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *Advances in neural information processing systems*, 32, 2019.

[9] Paras Jain, Ajay Jain, Aniruddha Nrusimha, Amir Gholami, Pieter Abbeel, Joseph Gonzalez, Kurt Keutzer, and Ion Stoica. Checkmate: Breaking the memory wall with optimal tensor rematerialization. *Proceedings of Machine Learning and Systems*, 2:497–511, 2020.

[10] Taebum Kim, Hyoungjoo Kim, Gyeong-In Yu, and Byung-Gon Chun. Bpipe: memory-balanced pipeline parallelism for training large language models. In *International Conference on Machine Learning*, pages 16639–16653. PMLR, 2023.

[11] Vijay Anand Korthikanti, Jared Casper, Sangkug Lym, Lawrence McAfee, Michael Andersch, Mohammad Shoeybi, and Bryan Catanzaro. Reducing activation recomputation in large transformer models. *Proceedings of Machine Learning and Systems*, 5:341–353, 2023.

[12] Mu Li, David G Andersen, Jun Woo Park, Alexander J Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J Shekita, and Bor-Yiing Su. Scaling distributed machine learning with the parameter server. In *11th USENIX Symposium on operating systems design and implementation (OSDI 14)*, pages 583–598, 2014.

[13] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, et al. Pytorch distributed: Experiences on accelerating data parallel training. *arXiv preprint arXiv:2006.15704*, 2020.

[14] Shigang Li and Torsten Hoefler. Chimera: efficiently training large-scale neural networks with bidirectional pipelines. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14, 2021.

[15] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R Devanur, Gregory R Ganger, Phillip B Gibbons, and Matei Zaharia. Pipedream: Generalized pipeline parallelism for dnn training. In *Proceedings of the 27th ACM symposium on operating systems principles*, pages 1–15, 2019.

[16] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, et al. Efficient large-scale language model training on gpu clusters using megatron-lm. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15, 2021.

[17] Xuan Peng, Xuanhua Shi, Hulin Dai, Hai Jin, Weiliang Ma, Qian Xiong, Fan Yang, and Xuehai Qian. Capuchin: Tensor-based gpu memory management for deep learning. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 891–905, 2020.

[18] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. Deepspeed: System optimizations enable

training deep learning models with over 100 billion parameters. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 3505–3506, 2020.

[19] Jie Ren, Samyam Rajbhandari, Reza Yazdani Aminabadi, Olatunji Ruwase, Shuangyan Yang, Minjia Zhang, Dong Li, and Yuxiong He. {Zero-offload}: Democratizing {billion-scale} model training. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 551–564, 2021.

[20] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

[21] Shaden Smith, Mostofa Patwary, Brandon Norick, Patrick LeGresley, Samyam Rajbhandari, Jared Casper, Zhun Liu, Shrimai Prabhumoye, George Zerveas, Vijay Korthikanti, et al. Using deepspeed and megatron to train megatron-turing nlg 530b, a large-scale generative language model. *arXiv preprint arXiv:2201.11990*, 2022.

[22] Zhenbo Sun, Huanqi Cao, Yuanwei Wang, Guanyu Feng, Shengqi Chen, Haojie Wang, and Wenguang Chen. Adapipe: Optimizing pipeline parallelism with adaptive recomputation and partitioning. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pages 86–100, 2024.

[23] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023.

[24] Linnan Wang, Jinmian Ye, Yiyang Zhao, Wei Wu, Ang Li, Shuaiwen Leon Song, Zenglin Xu, and Tim Kraska. Superneurons: Dynamic gpu memory management for training deep neural networks. In *Proceedings of the 23rd ACM SIGPLAN symposium on principles and practice of parallel programming*, pages 41–53, 2018.

[25] Aiyuan Yang, Bin Xiao, Bingning Wang, Borong Zhang, Ce Bian, Chao Yin, Chenxu Lv, Da Pan, Dian Wang, Dong Yan, et al. Baichuan 2: Open large-scale language models. *arXiv preprint arXiv:2309.10305*, 2023.

[26] Bowen Yang, Jian Zhang, Jonathan Li, Christopher Ré, Christopher Aberger, and Christopher De Sa. Pipemare: Asynchronous pipeline parallel dnn training. *Proceedings of Machine Learning and Systems*, 3:269–296, 2021.

[27] Tailing Yuan, Yuliang Liu, Xucheng Ye, Shenglong Zhang, Jianchao Tan, Bin Chen, Chengru Song, and Di Zhang. Accelerating the training of large language models using efficient activation rematerialization and optimal hybrid parallelism. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*, pages 545–561, 2024.

[28] Shixiong Zhao, Fanxin Li, Xusheng Chen, Xiuxian Guan, Jianyu Jiang, Dong Huang, Yuhao Qing, Sen Wang, Peng Wang, Gong Zhang, et al. v pipe: A virtualized acceleration system for achieving efficient and scalable pipeline parallel dnn training. *IEEE Transactions on Parallel and Distributed Systems*, 33(3):489–506, 2021.

[29] Quan Zhou, Haiquan Wang, Xiaoyan Yu, Cheng Li, Youhui Bai, Feng Yan, and Yinlong Xu. Mpress: Democratizing billion-scale model training on multi-gpu servers via memory-saving inter-operator parallelism. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 556–569. IEEE, 2023.

PPipe: Efficient Video Analytics Serving on Heterogeneous GPU Clusters via Pool-Based Pipeline Parallelism

Z. Jonny Kong*
Purdue University

Qiang Xu*
Purdue University

Y. Charlie Hu
Purdue University

Abstract

With the rapid innovation of GPUs, heterogeneous GPU clusters in both public clouds and on-premise data centers have become increasingly commonplace. In this paper, we demonstrate how pipeline parallelism, a technique well-studied for throughput-oriented deep learning model training, can be used effectively for serving latency-bound model inference, *e.g.,* in video analytics systems, on heterogeneous GPU clusters. Our work exploits the synergy between diversity in model layers and diversity in GPU architectures, which results in comparable inference latency for many layers when running on low-class and high-class GPUs. We explore how such overlooked capability of low-class GPUs can be exploited using pipeline parallelism and present a novel inference serving system, PPipe, that employs *pool-based pipeline parallelism* via an MILP-based control plane and a data plane that performs resource reservation-based adaptive batching. Evaluation results on diverse workloads (18 CNN models) show that PPipe achieves 41.1%–65.5% higher utilization of low-class GPUs while maintaining high utilization of high-class GPUs, leading to 32.2%–75.1% higher serving throughput compared to various baselines.

1 Introduction

While applications based on large language models (LLMs) have witnessed remarkable advancements in recent years, video analytics systems [1–4], which leverage extensive networks of cameras deployed in major cities across the U.S. and around the world [3–5], remain heavily reliant on traditional machine learning models such as convolutional neural networks (CNNs). These systems play a critical role in enabling a wide range of applications, including real-time surveillance for security purposes, efficient transportation management through traffic monitoring, enhanced public safety via crowd analysis, and improvements in healthcare through patient monitoring. The global video analytics market is estimated to grow from $3.2 billion in 2023 to $19.1 billion by 2030, at a CAGR of 29.2% [17].

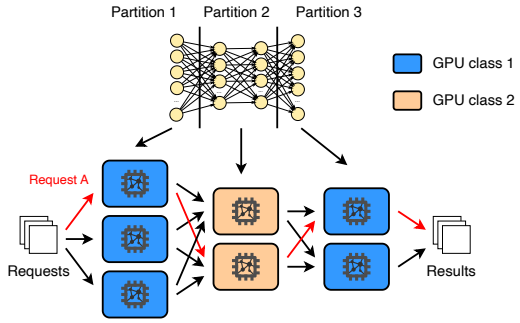


Figure 1: Pool-based pipeline parallelism on a heterogeneous GPU cluster with model partitioning. Each request (*e.g.,* request A) is processed by all partitions sequentially, and may be processed by any of the GPU servers allocated to each partition (GPU pool).

In video analytics systems, streams of video frames from cameras deployed at different locations of interest are uploaded to the cloud servers that perform analytics, *i.e.,* deep neural network (DNN) model inference. Serving such inference requests is challenging for two primary reasons. First, video analytics inference requests often have stringent service level objectives (SLOs), *e.g.,* 200 ms [26, 65]. Second, inference requests from real-world applications can be bursty [11, 34, 47, 62]. Meeting the latency SLO for all inference requests requires provisioning hardware resources for the peak load, which can be costly as the hardware resource becomes under-utilized during off-peak periods.

As with model training, model inference relies on the use of accelerators such as GPUs. With rapid innovation of GPUs [40], newer generations of GPUs are introduced to the market in short release cycles. Yet, their high cost and limited supply have dis-incentivized cloud vendors and private organizations from retiring older generations of GPUs. As a result, these organizations are increasingly operating highly heterogeneous GPU clusters [54].

This paper studies how to serve popular DNN models, *i.e.,* with high volumes of requests [8], on heterogeneous GPU

*Both authors contributed equally to the paper.

clusters. Being able to do so not only allows utilizing low-class computing resources that are otherwise unusable [43] in clusters dedicated to model serving, *e.g.,* in edge clouds or private clouds running AI-based apps [7, 18], but also, as we will show in this paper, can significantly enhance the serving throughput of high-class GPUs.

In particular, we explore the under appreciated benefits of pipeline parallelism among lower-class and higher-class GPUs in online model serving. While the benefits of pipeline parallelism have been well studied for throughput-oriented model training [14, 29, 32, 43, 56], its potential for model serving under latency (SLO)-constrained settings on heterogeneous GPU clusters has been largely unexploited. Intuitively, the benefits of partitioning a model and pipelining the partitioned inference among mixed high-class and low-class GPUs appear limited. For example, if the high-class GPU is 10× faster (*i.e.,* lower latency) than the low-class GPU for a given model, then simply running 1/10 of the model layers on the low-class GPU already leads to 1.9× longer total latency.

We instead make a key observation about the performance characteristics of DNN model inference on heterogeneous GPUs: *the two forms of diversity in model inference on a heterogeneous GPU cluster — diversity in model layers and in GPU types — can interact with each other synergistically.* First, a DNN model typically has many layers with diverse operations and tensor dimensions, leading to varying GPU utilization on a given GPU. Second, more importantly, for the same DNN model, the relative per-layer inference latency on different classes of GPUs can vary significantly across the model layers. This observation suggests that partitioning the DNN model in a GPU-aware manner and executing each partition on the GPU type that runs most effectively can improve the effectiveness of all GPU types and hence the inference throughput of the whole cluster. Effectively, lower-class GPUs offload part of the model inference from higher-class GPUs with minimum elongation of the end-to-end inference latency.

While this observation provides guidelines on efficient pipeline parallelism on heterogeneous GPU servers, partitioning a model and running the partitions along a *chain* of GPUs, as done in pipeline parallel DNN training [25, 39], is too stringent and leads to suboptimal partitions: all partition stages must have matching latencies to avoid pipeline stalls, which restricts the flexibility of model partitioning and leaves less opportunity for low-class GPUs to run layers they are efficient at.

To this end, we present PPipe, a model serving system that harnesses mixed GPU types in heterogeneous GPU clusters to maximize serving throughput. PPipe is built on three key ideas. First, to improve scheduling flexibility, it employs *pool-based pipeline parallelism* where each model partition is associated with a *pool* of GPU servers, and each request can be processed by any GPU in each partition pool along the pipeline, as shown in Figure 1. *Such pooled pipeline parallelism allows different partitions to have different numbers of GPU servers, varying inference latencies, and run with different batch sizes, as long as the inference throughput provided by each pool of GPU servers matches with each other.*

Second, to realize the scheduling flexibility exposed by pool-based pipelined model inference, PPipe generates the optimal configuration of pool-based pipelined model inference, *e.g.,* one that maximizes the model serving throughput of a given cluster while meeting inference SLOs, using Mixed Integer Linear Programming (MILP), which takes as input the per-layer inference latency on all candidate GPUs and under all candidate batch sizes from offline profiling.

Such an MILP-based optimal solution, however, effectively assumes ideal request arrivals, *i.e.,* synchronous arrival of batches in locksteps at all GPUs of the first partition pool, and flow down the pipeline partitions in sync. In practice, the inference requests arrive asynchronously and can be bursty [11, 34, 47, 62], which creates transient high load that overwhelms the throughput prescribed in the MILP solution, introducing several sources of extra delay not accounted for in the MILP solution and leading to SLO violations.

To bridge the gap between the MILP solution and runtime dynamics due to asynchronous and bursty request arrivals, PPipe treats the MILP-based formulation as the control plane which prescribes optimal DNN model partition and GPU allocation, and employs a novel data plane that performs *resource reservation-based adaptive batching* to address the unique challenges in batching pooled-based pipelines: deciding for each batch *which pooled pipeline, which path within the pipeline, and the batch size.* Our scheduler overcomes the above challenges by (1) maintaining (current and future) availability of resources (GPUs and network bandwidth) in the pooled pipelines; and (2) probing them to find the maximal batch of requests that can meet the SLOs of each request when the batch reaches the end of the pipeline path.

We evaluate PPipe using production workloads on top of 100-GPU large-scale simulations and 16-GPU testbeds on Google Cloud consisting of a variety of high- and low-class GPUs such as NVIDIA V100, L4, T4, and P4. Evaluation across 18 CNN models shows that PPipe achieves 44.1%–65.5% higher utilization of low-class GPUs while maintaining high utilization of high-class GPUs compared to various baselines, leading to 32.2%–75.1% higher serving capacity, while successfully processing 99% of the requests without dropping or SLO violations.

In summary, we make the following contributions:

- The first exploration of pipeline-parallel model serving on heterogeneous GPU clusters under latency (SLO)-constrained settings.
- The complete design of PPipe, which employs three design ideas to maximize inference throughput of heterogeneous GPU clusters: pool-based pipeline parallelism, an MILP-based control plane that prescribes optimal pool-based pipeline plans, and a data plane that performs resource reservation-based adaptive batching to handle run-

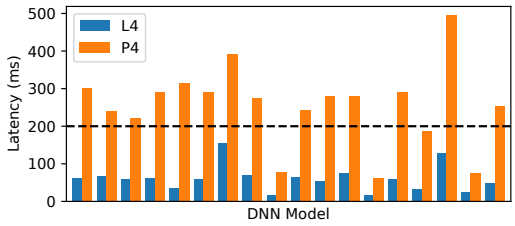


Figure 2: Inference latency of 18 popular DNN models (Table 2) under batch size 4 on different GPU classes.

time dynamics due to asynchronous and bursty request arrivals.

- Extensive evaluation of PPipe showing PPipe outperforms baseline designs by 32.2%–75.1% in inference throughput and 41.1%–65.5% higher low-class GPU utilization.

The source code of PPipe is available at https://github.com/JonnyKong/PPipe.

2 Motivation and Key Idea

We motivate how low-class GPUs can be effectively used to augment high-class GPUs in a heterogeneous cluster in model serving by exploiting pipeline parallelism.

Low-class GPUs fail to meet the inference latency SLO. On a highly heterogeneous GPU cluster, the inference time on old or low-class GPUs is usually several times longer than that on newer or high-class GPUs. For the 18 DNN models we use for evaluation (§7.1), the inference time (on the highly optimized TensorRT [10] inference framework) on the low-class NVIDIA P4 is 3.0x–7.9x longer than that on the high-class NVIDIA L4, as shown in Figure 2. Even if a model runs on the low-class GPU without violating latency SLO, it can barely perform batched inference, which could significantly improve GPU utilization and throughput [6, 12, 47]. As shown in Figure 2, only 22% of the DNN models can run on the low-class GPU (NVIDIA P4) at batch size 4 without exceeding 200 ms, a latency SLO target commonly used among video analytics pipelines [26, 65].

Key insight: Diversity in per-layer inference delay across GPUs. Given the high latency of low-class GPUs, the benefit of partitioning and running a model across low- and high-class GPUs, if done in a GPU-oblivious manner, will be limited. For example, if the high-class GPU is 10× faster (*i.e.,* lower latency) than the low-class GPU for a given model, then simply running 1/10 of the model layers on the low-class GPU already leads to 1.9× longer total latency. Our key observation is that there exist two forms of diversity in model inference on a heterogeneous GPU cluster: diversity in model layers and in GPU types, and they can interact with each other synergistically.

In particular, for many popular CNN backbone architectures, *e.g.,* EfficientNet [49] and ResNet [23], later layers have more channels compared to earlier layers, but with lower feature dimensions. Such architectural differences among layers

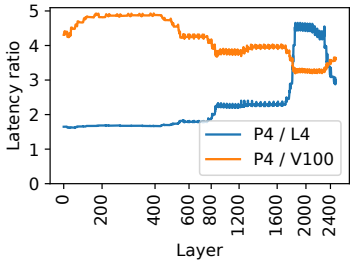


Figure 3: The ratio of inference latency on NVIDIA P4 over L4 and P4 over V100 across EfficientNet-B8 [49] layers.

within a DNN model can lead to different computational properties on GPU accelerators. To gain insight into this, we measure the *latency ratio*, *i.e.,* the ratio of the inference latency of the same layer on different GPU types for all the layers within a DNN model. Figure 3 shows that for EfficientNet-B8 on NVIDIA P4 over L4 and P4 over V100, respectively, with a moving window of 128 layers. The inference latency ratio on P4 over L4 is about 1.7 for early layers, indicating these layers have closer inference latencies on both P4 and L4. On the other hand, later layers have much higher latency ratios, and those layers will suffer significant slowdown running on P4 over L4. If we were to partition the DNN model and run it on P4 and L4, we should place earlier layers on P4 and later layers on L4, which provides higher chances to keep the inference time below the latency SLO and enables batching opportunities. All 18 DNN models we studied (Table 2) exhibit varying latency ratios across layers and we omit the rest due to page limit.

Interestingly, the latency ratios between P4 and V100 show completely different trends on EfficientNet, where earlier layers exhibit much higher latency ratios than later layers. In this case, one would run on P4 later layers instead. Such differences in the trends of latency ratios happen due to GPU design tradeoffs, architectural improvements, and their interaction with DNN layers of different characteristics. For example, GPUs with more SMs or higher ops:bytes ratio provide more benefits for layers of larger size or higher arithmetic intensity [19].

Such varying trends in per-layer latency ratios on different GPUs suggest that partitioning a DNN model in a *GPU-aware* manner is critical in exploiting pipeline parallelism so that high-class and low-class GPUs can work on model layers that they are optimized for, which improves their efficiency and hence the inference throughput of the whole cluster.

Key idea: pool-based pipeline parallelism. To apply model partitioning to exploit the diversity of per-layer inference ratio on different GPUs, a simple approach is partitioning a model and pipelining the inference of partitions along a *chain* of GPUs similarly as in pipeline parallelism in DNN training [25, 39], and feature maps generated by one GPU are transferred to the next downstream GPU. Such single-chain inference pipelines have the advantage of simple scheduling and coordination, but also come with two major drawbacks:

(1) To avoid pipeline stalls, all partitions need to have similar inference latencies. However, such a partitioning strategy is too stringent and will lead to suboptimal partitions, *e.g.,* leaving layers with high latency ratios (on high/low-class GPUs) to run on the low-class GPU or having a large feature map at the partition point. (2) Many GPUs cannot take advantage of heterogeneous inference when the cluster contains more GPUs of one class than the other.

To provide more scheduling flexibility, we instead associate each partition with *a pool of GPU servers* of the same type, and each request can be processed by any GPU allocated to the first partition, and then continue the inference on any GPU in the second partition, and so on, as shown in Figure 1. This approach mitigates the drawbacks of single-chain pipelines: different partitions can have different numbers of GPU servers (*e.g.,* N_1 servers in pool 1, N_2 servers in pool 2), have different inference latencies (*e.g.,* $t1$ and $t2$ for the 2 pools), or even run with different batch sizes (*e.g.,* $b1$ and $b2$ for the 2 pools), as long as the inference throughput provided by each pool of GPU servers matches well with each other (*e.g.,* $N_1 * b_1/t_1 = N_2 * b_2/t_2$), and the total latency is below the latency SLO. In an optimal partitioning, multiple such *pooled pipelines* may be employed at the same time, employing different ways of partitioning the DNN model running on different GPU pools.

3 Prelude to PPipe: Basic MILP Formulation

To exploit pool-based pipelined inference, one needs to figure out the optimal way to partition a DNN model, the placement of the DNN partitions onto GPU servers, and the batch size for the GPUs in each partition. It is relatively straightforward to formulate an MILP problem to figure out the optimal solution. We briefly describe the MILP formulation below and present the full mathematical formulation in Appendix §A.1.

Inputs. The MILP formulation takes as input the GPU count of each GPU type, the interconnect bandwidth of the target cluster, and the inference latency SLO. To decide the optimal model partitions, it also requires the intermediate feature map sizes and the inference latencies of individual layers within a DNN model under different batch sizes and on different GPU models, which can be obtained from the profiling output of TensorRT [10].

Encoding model partition and placement. Suppose the GPU cluster consists of 2 GPU types and we restrict a DNN model to be divided into at most 3 partitions. The placement of DNN model partitions falls into one of 14 potential pooled pipelines: if partitioned into 2 partitions, each partition can run on a pool of either GPU types (4 pipeline options); if partitioned into 3, each of the 3 partitions again has the choice to run on a pool of either GPU types (8 pipelines); the DNN model can also directly run on a pool of either GPU type without partitioning (2 pipelines). As discussed in §2, the optimal solution may contain multiple pooled pipelines.

For each partition within a pooled pipeline, we need to decide on the exact partition points, *i.e.,* the first and last layers. To this end, we construct a set of binary decision variables for each partition indicating whether each layer is the first or last layer of a partition. Apart from that, we also create decision variables to represent the batch size and the number of GPUs used by each partition.

Constraints. The total latency of each pooled pipeline, including the inference latency of each partition and the feature map transfer latency between partitions (both can be derived from the batch size), should be below the latency SLO; the total GPU count used by all partitions pertaining to a specific GPU type should not exceed the GPU count for that GPU type. Finally, the inference throughput of a pooled pipeline is bottlenecked by the partition of lowest throughput, where each partition's throughput can be calculated based on its batch size, inference latency, and the number of GPUs allocated to the partition. We observe that CNN models commonly used in video analytics pipelines are typically not memory-constrained on datacenter GPUs; hence, the MILP formulation does not account for GPU memory.

Objective. By default, we try to maximize the total inference throughput of the GPU cluster, which is the sum of the throughputs of all pooled pipelines employed by the MILP solution. The MILP formulation can also be configured for other objectives like minimum server cost [24] or provisioned power [28]. In the presence of multiple DNN models, given the ratio between the DNNs' workloads, the MILP formulation computes the normalized throughput for each DNN (throughput divided by the DNN's workload percentage), and maximizes the lowest normalized throughput among the DNNs.

Outputs. The solver of the MILP formulation outputs the pooled pipelines employed by the optimal plan, *i.e.,* those being allocated at least 1 GPU. For each pipeline, the solver outputs the DNN model partition points, the batch size used by the GPUs in each partition, and the number of GPUs allocated to each partition.

In essence, MILP holistically determines an optimal set of pooled pipelines by selecting the model partition points for each pipeline, along with the GPU type, count, and batch size for each partition in each pipeline, all of which affect the overall throughput of the GPU cluster.

4 Challenges in Developing a Working System

While the MILP formulation above provides the optimal plan in theory, turning it into a working DNN serving system faces several practical challenges, as discussed below.

C1: Extensive search space of the MILP formulation. The MILP formulation needs to decide the first and last layers of a DNN partition, whose complexity depends on the number of layers in a DNN model. For the set of representative models

in our evaluation (§7.1), the average layer count is 613.2. The partition points need to be searched for all partitions across all pipelines, making the search space combinatorial. The search space is further inflated by additional dimensions including inference batch size and GPU count used by each partition. With such a vast search space, it takes more than 7 hours (running the Gurobi [21] solver on a Google Cloud n1-standard-64 instance) to obtain the optimal solution for 80 layers, making it impractical to adapt to changing workload, *e.g.,* diurnal load [28].

C2: Asynchronous and bursty request arrival. In essence, the MILP formulation outputs a solution that assumes ideal inference request arrival. Suppose a pipeline solution consists of two partitions with 40 ms inference latency each, and the inference throughput is 1000 requests per second. The MILP solution effectively assumes that 40 requests arrive at the same time every 40 ms, which are simultaneously processed by all GPUs allocated to the first partition, and then forwarded to the second partition, and so on.

In reality, in an online inference system, inference requests arrive asynchronously and in a bursty manner, which can disrupt the MILP solution with two forms of extra delays: (1) Early arriving requests have to wait for later requests to form a batch to be dispatched to a GPU in the first partition, incurring *initial batching delay* (D1); (2) The staggered batched inference initiated on the GPUs in the first partition will cascade through the remaining partitions in the pipeline. In such staggered pipelined inference, it is possible when a GPU in partition i finishes inference on a batch, all of the GPUs in partition $i+1$ are still busy running other batches, causing *inter-partition queuing delay* (D2). Such queuing delay is further complicated when partitions use different batch sizes, requiring the split and merge of batches which creates complex dependencies between the GPUs of different partitions.

To incorporate the above extra delays at runtime, we could add a predefined margin to the latency SLO as input to the MILP formulation [44, 47] which will output adjusted (still fixed) batch sizes. But simply adding a static margin cannot handle bursty request arrival, which can still result in either too many or too few transient requests compared to the statically adjusted target batch size. Such dynamic conditions require *dynamically adjusting the batch sizes.*

C3: Network contention. We observe that heterogeneous clusters such as Google Cloud Platform (GCP) and Amazon Web Services (AWS) come with high-bandwidth networks that theoretically can finish the transfer of a feature map with a small percentage of the total inference latency. For example, GCP's P4 instances have a bandwidth of 32 Gbps, which can theoretically transfer feature maps of CenterNet (with batch size 1) which range from 3 MB to 50 MB in 0.8–13.2 ms. However, it is common for multiple GPUs to collocate on the same server, sharing the server's network bandwidth. This can lead to network contention between GPUs (D3) that disrupts pipeline schedules and causes SLO violations. The contention becomes more severe when we divide each GPU into multiple virtual GPUs (§5.3), which increases the number of "GPUs" on the same server that likely transfer feature maps at the same time. Note that this issue cannot be addressed by conservatively allocating each virtual GPU an equal share of the available bandwidth, as it results in low bandwidth for all virtual GPUs, significantly limiting the benefit of pipeline parallelism.

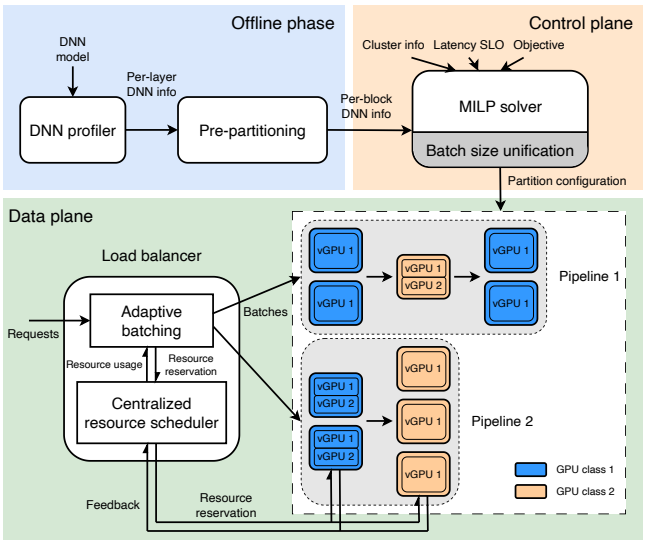


Figure 4: PPipe architecture.

5 PPipe Design

As discussed above, designing a practical pool-based pipeline parallel inference serving system faces a key challenge: the MILP formulation does not capture or handle runtime dynamics due to asynchronous and bursty request arrival or delayed feature map transfer from network contention. We tackle these challenges by splitting PPipe, our *pool-based* pipeline parallel inference serving system for heterogeneous GPU clusters, into a control plane and a data plane. First, PPipe treats the MILP-based formulation as the control plane that prescribes optimal DNN model partitions and GPU allocation for each pooled pipeline. Second, to handle delays caused by asynchronous and bursty request arrivals (D1 & D2) and network contention (D3), PPipe employs a novel data plane that performs *resource reservation-based adaptive batching* to ensure the request batches injected into the pooled pipeline meet their latency SLOs.

5.1 Architecture Overview

Figure 4 shows the architecture of PPipe, which consists of an offline phase, a control plane, and a data plane.

Offline phase. In the offline phase, the DNN model profiler profiles a model's per-layer inference latency on different GPU types. To reduce the search space of the MILP solver

(C1), we design a pre-partitioning method that groups the layers of each DNN model into blocks which are fed into the MILP solver (§5.2). With this method, the MILP solver only needs to find partition points among a few blocks instead of hundreds of layers, significantly reducing the search space. We profile each model and each block independently, and each block is profiled on every GPU type and batch size. The profiling is fast, requiring only a few hours to cover all 18 DNN models. Since the addition of models happens infrequently, and each model will be served over a long time, the one-time offline profiling cost is amortized and manageable. When solving MILP, the latency of each partition is computed as the sum of the latencies of its constituent blocks.

Control plane. The control plane, which runs the MILP solver, takes as input the profiling information of DNN blocks and the inference latency SLO for each DNN, the cluster information (GPU count for each GPU model), along with high-level objective, *e.g.,* maximum throughput, and outputs the partitioning of each DNN model and allocation of GPU resources across partitions in each pooled pipeline (detailed in §3). We observe that the synchronization among partitions (**C2**) could be much simpler if partitions within the same pooled pipeline all use the same batch size. As such, we enhance the MILP formulation with *batch size unification*.

The MILP solver runs periodically, triggered dynamically in response to workload changes, such as shifts in the load ratio when serving multiple DNNs, which typically occur once every one or a few hours [28, 68]. Note that the MILP solver runs on the CPU, and is asynchronous to the inferences running on the GPUs. Migrating to a new MILP plan involves reassigning GPUs to different partitions or pooled pipelines, and having the data plane dispatch the requests according to the updated plan. To minimize migration latency, each GPU asynchronously preloads the new model weights into memory ahead of the switch, without interrupting the ongoing inference of the existing model; this is feasible because GPU memory is generally not a bottleneck for vision models. Once all GPUs complete loading, PPipe pauses ingesting new requests to perform a pipeline flush, which takes about 1x the SLO of the currently serving DNNs (in the order of 100s of milliseconds). After the flush, all GPU switch to the new weights simultaneously and the data plane resumes request dispatching. In essence, each migration incurs a downtime of several hundred milliseconds, which is negligible compared to the interval between migrations.

Data plane. The data plane groups inference requests into batches and executes them through the pools of GPUs in each of the pipelines prescribed by MILP. To address the extra delays D1–D3, we design a novel adaptive batching scheme that selects the pooled pipeline to execute the next batch, one GPU from each pool to run the corresponding model partition (the *pipeline path*), and the actual batch size that meets the request SLOs.

5.2 DNN Pre-Partitioning

As discussed in **C1**, the sheer number of layers within a DNN model results in a huge search space for the MILP formulation and prevents the solver from finding good plans in a reasonable amount of time. To this end, we devised a simple DNN pre-partitioning approach that groups the layers in a DNN model into a few (N) blocks of approximately equal runtime on a selected GPU type; empirical results show that the choice of GPU type has minimal impact on the partitioning. Specifically, we start from the first layer and sequentially group consecutively layers together until their combined runtime is as close as possible to $1/N$ of the runtime of the entire DNN; this process is repeated until we reach the last layer. After grouping layers into blocks, we profile the blocks on different GPU types and with different batch sizes, as needed by the MILP solver. It takes only less than 10 minutes to profile a single DNN model, as each block can be profiled independently; when solving MILP, we obtain the latency of each partition by adding up the latencies of the blocks in it.

With pre-partitioning, the MILP solver only needs to find partition points among the N blocks (we tried $N = 5$ to $N = 20$ and found $N = 10$ provides a good balance between plan optimality and MILP running time), instead of 613.2 layers on average across the set of models (§7). As a result, the MILP runtime is significantly reduced to 3.5 seconds on average over different DNN model and GPU cluster setups.

5.3 Batch Size Unification

We tackle the key challenge of the data plane — bridging the gap between MILP solution and runtime dynamics (**C2** and **C3**) — in two steps. In the first step, we simplify the challenges faced by data plane scheduling with *batch size unification*.

As discussed in **C2**, mismatch of batch sizes between partitions within a pooled pipeline requires batches to be merged and split which complicates scheduling of batched inference across partitions. Things can be substantially simplified if all partitions within the same pooled pipeline use the same batch size. However, as the GPUs have different computational capacities, in the plans generated by the MILP planner, high-class GPUs tend to use larger batch sizes compared to low-class GPUs to improve GPU utilization and inference throughput. Naively forcing all partitions within the same pooled pipeline to use a uniform batch size can lead to underutilization of high-class GPUs and degrade overall system performance.

Instead of forcing all partitions of a pooled pipeline to use the same batch size, we vary the GPU size to equalize the batches per GPU. Specifically, we incorporate virtual GPUs into the MILP formulation, so that the MILP solver can take into account the throughput and inference latency differences between batches on different partitions and make

holistic decisions in choosing a unified batch size. To this end, instead of feeding a GPU as a whole to MILP, we feed four possible virtual GPU types: 1/1, 1/2, 1/3 and 1/4 of a physical GPU (this is achieved with Multi-Process Service (MPS) [38] during runtime). The use of virtual GPUs only mildly expands the search space of the MILP solver as there are only 4 virtual GPU types. Further, we profile the per-block inference latencies under not only different batch sizes and GPU types, but also different virtual GPU types.* Finally, we add additional constraints to the MILP formulation requiring all partitions within the same pooled pipeline to use the same batch size. The mathematical representation of the enhanced MILP formulation is provided in Appendix §A.2.

5.4 Resource Reservation-Based Adaptive Batching

With batch size unification, PPipe stills needs to perform adaptive batching at runtime, *i.e.,* dynamically forming and scheduling batches, in serving asynchronous and bursty inference requests.

Compared to pipelined inference over a single chain of GPUs, dynamic batching in pool-based pipelines faces unique new challenges. In a chain of GPUs, the scheduler just needs to find the largest batch size that satisfy the end-to-end SLO. In a cluster of pooled pipelines (output by the MILP solution), the batching scheduler has to make three decisions for each batch: *which pooled pipeline, which path within the pipeline, and the batch size.* The decisions are further complicated by: (1) the optimal batch size depends on which pipeline the batch is sent to; (2) the optimal pipeline path in turn depends on the batch size and resource availability in the pooled pipeline.

Resource reservation-based adaptive batching. Our resource-reservation-based adaptive batching scheduler overcomes the above challenges by (1) maintaining (current and future) availability of resources (GPUs and network) in the pooled pipelines; and (2) probing them to find the maximal batch of requests that can meet the SLOs of each request when the batch reaches the end of the pipeline path.

The scheduler works in two steps, both using a stateless probing procedure `probe()`. The procedure takes a specific pooled pipeline and a hypothetical batch size as input, and outputs the pipeline path that minimizes the end-to-end (E2E) inference time under current resource availability, with an example shown in Figure 5a. The E2E inference time includes the per-stage inference time, network transfer time, and the waiting time for required resources (GPU and NIC) along the pipeline path. The detailed algorithm of `probe()` is explained later.

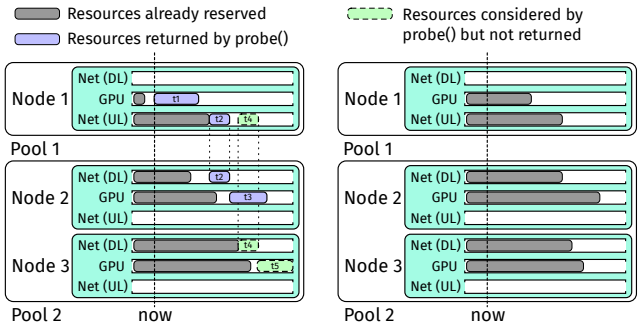
In Step 1, we identify the pooled pipeline i that can complete a batched inference at the pipeline's unified batch size bs_i (as determined by the MILP solution) with the shortest waiting time under the current resource availability. This is achieved by invoking `probe()` for each pooled pipeline with its corresponding unified batch size, and selecting the pipeline that has the lowest waiting time, where the waiting time is computed as the sum of delays waiting for each required resource along the pipeline path. Using waiting time as the metric effectively balances the load between the pipelines because it is a good indication of the load of the pipeline, *i.e.,* the lower the waiting time, the lower the load on the pipeline.

The E2E inference time of sending a batch of size bs_i down the selected pipeline i in Step 1, however, may not meet the SLO (as the batch size bs_i generated by MILP assumed synchronized request arrival). In Step 2, using the chosen pooled-pipeline i, we search for the actual largest batch size that can meet the SLOs and the corresponding pipeline path. Specifically, we iteratively invoke `probe()` with progressively smaller batch sizes, starting from bs_i (the batch size from the MILP solution), until the completion time of the pipeline path returned by `probe()` falls within the deadline of the first (oldest) pending request. Finally, one of three actions is taken depending on the chosen batch size and the number of pending requests: (1) If the deadline cannot be met even with batch size 1, the oldest request will be dropped and the adaptive batching process starts over from choosing a pooled pipeline (Step 1); (2) if the number of pending requests is smaller than the chosen batch size, the batching engine waits for more requests (till the last moment when the requests in queue can still be processed without SLO violation); (3) otherwise, the resources of the selected pipeline path returned by `probe()` are *reserved* (by calling `reserve()`), *i.e.,* their availability are updated as shown in Figure 5b, and the requests at the head of queue are grouped into a batch of the chosen batch size and dispatched to the selected pipeline path according to the resource reservation. Since the batch size is based upon the actual remaining time of the requests in the queue, the extra delay D1 (initial batching delay) is taken into account, and the requests in the batch are guaranteed to meet their SLOs.

The pseudocode for our adaptive batching algorithm is provided in Appendix §A.3.

The probing procedure. Since adaptive batching works at the batch level, the probing procedure needs to be fast to keep up with the request arrival rate. To this end, we implement `probe()` based on a greedy algorithm. In a nutshell, `probe()` works by sequentially selecting one GPU from each GPU pool along the pooled pipeline. For each GPU pool, it goes through all GPUs allocated to the pool and selects the GPU that minimize the completion time for that partition stage. During this process, `probe()` determines which resources—GPUs, uplink bandwidth, and downlink bandwidth—need to be reserved for specific future time intervals. Note that since feature map transfer require simultaneous availability of network resources on both the sending and receiving sides, `probe()` ensures the allocation of network resources that satisfy both

*We capture the interference between virtual GPUs during profiling by running the same DNN on all virtual GPUs of the same physical GPU in parallel.



(a) `probe()` takes as input a candidate pooled pipeline and batch size, and outputs the optimal pipeline path and the intervals of resources to be used (in blue).

(b) `reserve()` marks the resource intervals as reserved until the end of the intervals returned by `probe()`.

Figure 5: The resource reservation mechanism on an example pooled pipeline comprising two partitions, with 1 and 2 GPUs, respectively.

the uplink of the preceding GPU and the downlink of the current GPU.

We illustrate the workflow of `probe()` with an example in Figure 5a. Consider a pooled pipeline consisting of two pools, containing one and two GPUs respectively. For simplicity, we assume each GPU resides on a separate node, although in practice, multiple GPUs may reside on the same node and share the same network resources. In the first pool, `probe()` selects node 1 as it is the only option, which requires reserving its GPU for time duration `t1`. In the second pool, `probe()` needs to decide between nodes 2 and 3. Selecting node 2 requires the simultaneous reservation (as shown by the vertical dotted line) of node 1's uplink and node 2's downlink network during `t2`, as well as node 2's GPU during `t3`, after which the request completes; alternatively, selecting node 3 results in a completion time by the end of `t5`. Since node 2 results in an earlier completion time than node 3, `probe()` selects node 2. In the end, it returns the selected pipeline path with associated resource allocated: node 1's GPU during `t1`, node 1's uplink and node 2's downlink network during `t2`, and node 2's GPU during `t3`.

Since `probe()` is based on real-time resource availability, it directly takes into account extra delays D2 (inter-partition queuing delay) and prevents D3 (network contention delay). Furthermore, `probe()` runs in real time, and scales linearly with the number of virtual GPUs allocated to the pooled pipeline. The pseudocode for `probe()` and `reserve()` are provided in Appendix §A.3.

Feedback correction. The scheduler maintains resource reservation tables that keep track of when each resource will be used. However, this scheduler's view of resource usage might deviate from reality due to variations in inference time and network bandwidth. To this end, we let all nodes report back to the scheduler when the reserved resources were actually used immediately after every resource usage. The scheduler updates the resource usage table accordingly. The feedback correction mechanism ensures that the scheduler's view of resource usage is synchronized with reality at all times.

Extra SLO margin in the control plane. While our resource reservation-based adaptive batching algorithm ensures requests meet their SLOs, we notice that the resulting batch size may be much smaller than that in the MILP output due to extra delays D1–D3, causing large deviations from the MILP plan. To bridge the gap between control plane planning and data plane execution, we deduct an empirically determined margin from the SLO when running the control plane MILP solver, so that the adaptive scheduler picks the same batch sizes as in the MILP output most of the time.

Dispatching Complexity. In the worst case, dispatching a batch requires a number of `probe()` function calls equal to the product of the number of pipelines in the cluster and the number of candidate batch sizes. Each `probe()` function call has a time complexity linear to the number of GPUs within the corresponding pipeline. Consequently, the adaptive batching algorithm incurs low runtime overhead, as we will demonstrate in §7.2.

6 Implementation

Offline phase and control plane. We implement the offline phase and control plane in Python in 2.7 kLOC. We use Gurobi [21] as the MILP solver. We work with DNN models in their ONNX format and TensorRT format interchangeably, since the ONNX format provides flexibility, while the TensorRT format provides high inference performance.

Data plane. We implement both a discrete-event simulator for modeling large-scale GPU clusters and a prototype implementation for PPipe's data plane, in about 9.0 kLOC. The simulator is written in Java and maintains a global event queue sorted by timestamp, executing events in chronological order. Supported event types include request arrivals, batch dispatches, per-partition executions, and feature map transfers, etc. At each simulation step, the simulator dequeues the next event, invokes the corresponding event handler, which updates the system states and may produce additional events to be added to the event queue (*e.g.,* a request arrival event adds the request to the waiting queue and triggers a scheduler event).

The prototype implementation is written in a combination of Julia and C++, using TCP for control message exchanges and NVIDIA NCCL for transferring feature maps between nodes. To minimize the feature map transfer latency, we quantize float32 feature maps to float16 (only at partition boundaries), effectively reducing the transfer size by half. We find such quantization has negligible impact on task accuracy. The accuracy dropped by 0.00%, 0.01%, and 0.01% for object recognition, object detection, and instance segmentation tasks, respectively.

Table 1: Heterogeneous Cluster (HC) setups.

Setup	GPUs	Setup	Instances	GPUs	BW (Gbps)
HC1-L	25× L4, 75× P4	HC1-S	4× g2-standard-16, 2× n1-highcpu-16	4× L4, 12× P4	50
HC2-L	25× L4, 75× T4	HC2-S	1× g2-standard-48, 6× n1-highcpu-32	4× L4, 12× T4	32
HC3-L	25× V100, 75× P4	HC3-S	2× n1-highcpu-16, 12× n1-highcpu-16	4× V100, 12× P4	50
HC4-L	25× V100, 75× T4	HC4-S	1× n1-standard-64, 6× n1-highcpu-32	4× V100, 12× T4	32

Table 2: DNN models used in the evaluation.

Recognition	Detection	Segmentation	Others
ConvNext [35]	ATSS [64]	APCNet [22]	Color-v2 [63]
EfficientNet [49]	CenterNet [15]	DNL-Net [57]	
GoogleNet [48]	FSAF [69]	EncNet [61]	
RepVGG [13]	GFL [33]	FCN [36]	
WideResNet [59]	RTMDet [37]	GCNet [9]	
	EfficientDet [50]	NonLocalNet [53]	

7 Evaluation

In this section, we evaluate PPipe's serving performance under a variety of DNN models from different tasks, considering various combinations of low-class and high-class GPUs. We show that PPipe can serve 32.2%–75.1% more requests compared to various baselines while meeting 99% SLO attainment, on the discrete-event simulator with 100 GPUs. Additionally, on 16-GPU clusters deployed on Google Cloud, PPipe achieves 16.7%–52.8% higher serving throughput. We also conduct sensitivity analysis to show the impact of GPU composition and SLO on PPipe's performance.

7.1 Methodology

Cluster configuration. We consider 4 heterogeneous cluster setups, labelled as HC1−HC4 in Table 1. Each setup consists of a large (L) 100-GPU variant used for the discrete-event simulator, and a small (S) 16-GPU variant deployed on Google Cloud. Note that a Google Cloud VM instance can host multiple GPUs, resulting in each HC having a varying number of VMs while maintaining a consistent number of GPUs. Accounting for the scarcity of high-class GPUs [54], our default configuration includes 25 high- and 75 low-class GPUs for each HC's large variant, and 4 and 12 for the small variant. We further evaluate PPipe's performance under different ratios of high- and low-class GPUs in §7.6. Note that for GPU-equipped VMs, Google Cloud provisions network bandwidth based on the number and type of its GPUs, leading to different interconnect bandwidths across HCs. Furthermore, the effective bandwidth for both large and small clusters is only 1/5 the claimed values in Table 1 to accommodate the observed 5× network tail latency on Google Cloud.

Workloads. Following prior works [34, 62], we use Microsoft's Azure Function (MAF) traces from 2019 [46] and 2021 [66], which were originally derived from Azure serverless function invocations, as representative inference serving workloads. When serving multiple DNNs in parallel (§7.2), functions are assigned to DNNs in a round-robin manner to determine the workload ratio among DNNs. The MAF 2019 trace only includes per-minute aggregated request counts for each serverless function, and thus we issue requests using Poisson arrival at the given target load. Conversely, the MAF 2021 trace includes per-request arrival timestamps, and thus we upscale the trace to the target load and issue requests accordingly. The Poisson-emulated 2019 trace is less bursty than the 2021 trace, and thus we refer to the two traces as "Poisson" and "Bursty", respectively.

DNN models. We select 18 popular DNN models from public DNN registries such as TorchVision [51], OpenMMLab [41], and OpenVINO model zoo [42]. The selected DNNs serve a variety of popular computer vision tasks, as shown in Table 2.

Metrics. We employ two key metrics to evaluate the inference serving capability. First, *SLO attainment* represents the percentage of requests that are successfully processed without being dropped or violating the SLO, under a specific offered load. Second, we measure the maximum load that the system can handle at 99% SLO attainment.

Baselines. The large amount of recent works on model serving on heterogeneous clusters do not incorporate pipeline parallelism, and thus we abstract them into a state-of-the-art baseline, denoted as NP below. We also compare PPipe with the only prior work that exploits pipeline parallelism across heterogeneous GPUs, DART [55], which uses a single-chain-based pipeline of GPUs. To isolate the benefit of pool-based pipeline parallelism, we enhance all baselines to use PPipe's data plane, *i.e.,* the resource reservation-based adaptive batching (§5.4) in presenting the overall results (§7.2 & §7.3). We then evaluate the benefit of PPipe's second novel design, resource-reservation-based adaptive batching, in §7.4.

- **No-Partitioning (NP)**. NP executes the entire DNN on either high-class or low-class GPUs without partitioning. This way of serving DNNs on a heterogeneous cluster is representative of various prior works [11,24,27,28,30,31,45,60]. In particular, the allocation is done by solving PPipe's MILP formulation without model partitioning. When integrated with PPipe's reservation-based adaptive batching, NP effectively dispatches the largest possible batch to the next available GPU while meeting the SLO for each request in that batch.
- **DART-r**. DART [55] is an inference framework that partitions a DNN onto heterogeneous CPU and GPU cores. However, vanilla DART constructs a pipeline by chaining all available GPUs, each serving one model partition. This restricts its use to small clusters, as long pipelines incur significant overhead due to frequent feature map transfers. To address this limitation, we introduce DART-r, a modified version that replicates DART configurations for pairs of low- and high-class GPUs (more efficient than longer pipelines from fewer feature map transfers). If one GPU class has more GPUs than the other, the leftover GPUs are allocated to individually run entire DNNs without partitioning.

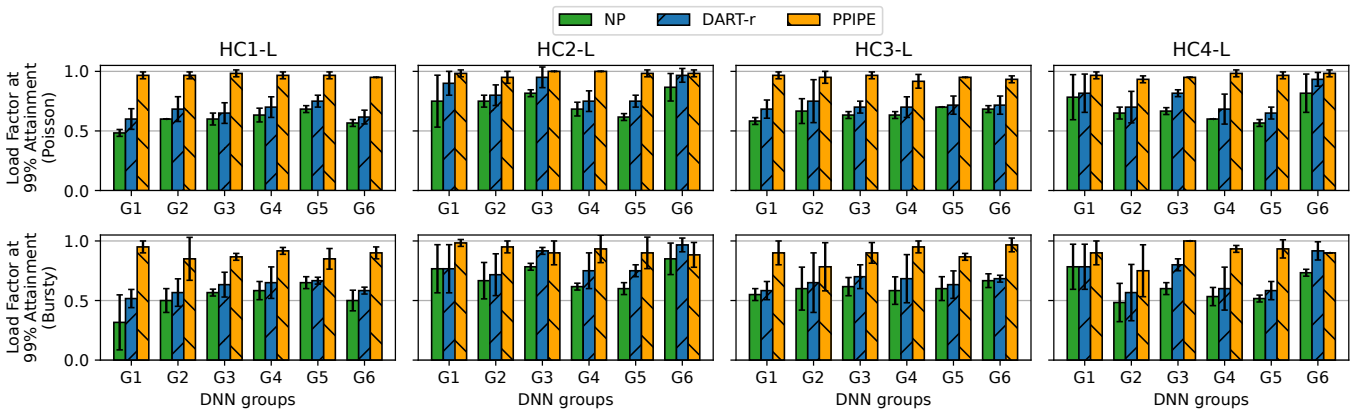


Figure 6: Maximum load factor of each framework under 99% SLO attainment on 100-GPU clusters, under "Poisson" and "Bursty" workloads.

Setup. Following prior work [34], for each DNN, we set the default SLO to be 5× its inference latency on the fastest GPU (NVIDIA L4) at batch size 1, resulting in SLOs ranging from 23.4 ms to 165.6 ms. This reflects the assumption that end users can tolerate higher latency for more complex models but are indifferent to the underlying GPU used. We further evaluate PPipe under other SLOs in §7.6, ranging from 2× to 10×. As mentioned in §5.4, to bridge the gap between control plane planning and data plane execution caused by the extra delays, a 40% margin is deducted from DART-r and PPipe's MILP formulation, and in NP in picking the maximum batch sizes that satisfy the SLOs. When comparing PPipe with the baselines, we use *load factor* 1.0 to denote the throughput in the output of PPipe's MILP. We generate requests using "Poisson" and "Bursty" traces respectively, with an average request rate (λ) ranging from 0.05 to 1.0 times the load factor, at an interval of 0.05. For each λ, the experiment lasts 30 seconds.

7.2 End-to-end Results

Overall results. In this section, we evaluate PPipe's capability of serving DNNs over 100-GPU clusters (HC1-L to HC4-L) on the discrete-event simulator. We randomly divide the 18 DNNs into 6 groups of 3 DNNs each (G1–G6), and serve DNNs within each group in parallel. During runtime, we record the maximum load ratio that each DNN can achieve under 99% attainment.

Figure 6 shows under cluster configurations HC1-L to HC4-L, the maximum load factor achieved under 99% SLO attainment, averaged over the DNNs in each group. First, we find that PPipe consistently outperforms NP. PPipe achieves on average 48.0% higher load factors than NP on the "Poisson" trace (64.9%, 34.8%, 46.7%, and 45.5% higher for HC1-L to HC4-L, respectively), and 75.1% on the "Bursty" trace (161.6%, 34.1%, 50.4%, and 54.1% higher), showing the advantage of PPipe's pipeline parallel inference scheme. Second, compared to DART-r, PPipe achieves 32.2% higher load factors on the "Poisson" trace (47.2%, 17.3%, 34.8%, and 29.3% higher on HC1-L to HC4-L, respectively), and 35.8% on the "Bursty" trace (50.4%, 18.1%, 40.2%, and 34.5% higher), showing the advantage of PPipe's pool-based pipelined inference over DART-r's chain-based pipelined inference. Finally, while all frameworks suffer reduced load factors under the "Bursty" trace (61.1%, 69.4%, and 90.3% for NP, DART-r, and PPipe respectively), compared to "Poisson" (66.8%, 74.9%, and 96.5%), the improvement of PPipe over the two baselines remain high, showing PPipe's robustness to varying request arrival patterns.

Figure 8 shows each framework's temporal utilization of high- and low-class GPUs. For brevity, we only show the utilization on the "Poisson" trace. While all frameworks achieve high utilization of high-class GPUs, NP and DART-r show zero or low utilization of low-class GPUs. On average, NP, DART-r and PPipe achieve low-class GPU utilizations of 8.1%, 29.5%, and 73.6% respectively. NP's low utilization is caused by the inference time of whole DNNs on low-class GPUs often higher than the SLO, prohibiting low-class GPUs from being used. While DART-r employs DNN partitioning, it chains only one low-end GPU with each high-end GPU, resulting in under-utilization of excess low-end GPUs when their number exceeds that of high-end GPUs.

SLO attainment under varying load factors. Figure 7 shows the SLO attainment under varying load factors. For brevity, we show only the SLO attainment for DNN group G1, which includes EfficientNet-B8, EncNet, and RtmDet, and only under the "Poisson" trace. The attainment is averaged over the 3 DNNs, *e.g.,* on HC1-L at load factor 1.0, PPipe achieve SLO attainments of 99.3%, 97.6%, and 98.4% on the 3 DNNs respectively, resulting in an averaged attainment of 98.4%, which is shown in the figure at load factor of 1.0.

We observe that PPipe outperforms both NP and DART-r, achieving the highest SLO attainment under the same load factors. Consequently, it achieves a higher load factor while ensuring 99% SLO attainment (Figure 6). For example, on HC1-L, PPipe achieves 99% SLO attainment under load factors up to 0.95, meaning it can handle at least 95% of the load calculated by the MILP solver. In contrast, NP and DART-r's

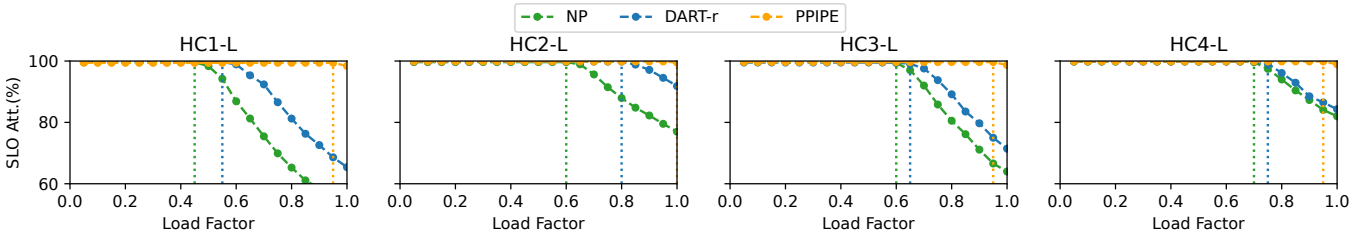


Figure 7: SLO attainment of DNN group G1 under the "Poisson" trace with varying load factors, averaged over the 3 DNNs in the group. The vertical dotted line denotes the load factor at which each system reaches 99% SLO attainment.

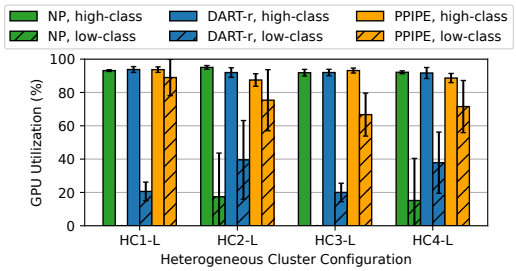


Figure 8: GPU temporal utilization under 99% SLO attainment, averaged over DNNs for each cluster configuration.

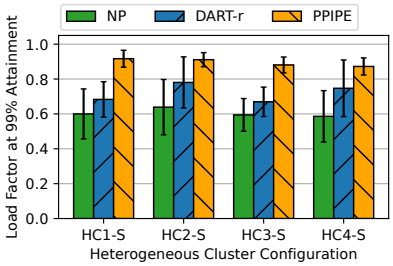


Figure 9: Maximum load factor each cluster configuration can achieve under 99% SLO attainment on the testbed, averaged over the DNNs.

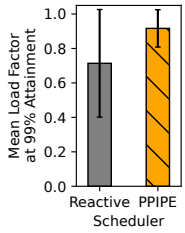


Figure 10: SLO attainment with the reactive scheduler and PPipe's data plane scheduler.

SLO attainment dips below 99% as the load factor exceeds 0.45 and 0.55, respectively. This is due to the fact that a load factor of 1.0 represents the serving capacity of PPipe, which is higher than the capacity of NP or DART-r, leading to the dropping of requests beyond their respective serving capacities. Note that although PPipe achieves higher load factors than the baselines, it may not reach the full load factor of 1.0, as seen with HC1-L, HC3-L, and HC4-L. This is due to unpredictable request arrival patterns (*e.g.,* Poisson), which cannot be fully accounted for by the MILP solver, as discussed in §4.

MILP runtime. PPipe's MILP solver takes 3.5 seconds on average, showing the effectiveness of PPipe's pre-partitioning (§5.2) in reducing the MILP complexity. This latency is negligible compared to the intervals between re-running the MILP — triggered by fluctuations in incoming load which occur relatively infrequently, *e.g.,* once every hour [28, 68].

Adaptive batching overhead. PPipe's resource-reservation-based adaptive batching mechanism is lightweight: on a 100-GPU cluster, dispatching a batch requires an average of only 3.58 `probe()` calls, incurring a total overhead of less than 9 microseconds. This overhead is negligible compared to the batch execution latency, even under bursty request patterns.

7.3 Testbed Results

We verify PPipe's DNN serving capability on real-world 16-GPU heterogeneous cluster testbeds (HC1-S to HC4-S) deployed on Google Cloud. Due to the testbed's smaller GPU counts, instead of serving DNNs in groups of three as in §7.2, we serve one DNN at a time with Poisson-arrival requests. Figure 9 shows the maximum load factor under 99% SLO attainment averaged over the 18 DNNs.

First, we observe that PPipe consistently outperforms NP, achieving 42.6%–52.8% higher load factors at 99% SLO attainment across the cluster configurations. This validates the advantage of PPipe's model-parallelism inference in a real-world setting. Secondly, compared to DART-r, PPipe achieves 16.7%–34.1% higher load factors, which shows the advantage of PPipe's pool-based pipelined inference. In summary, PPipe's significant performance gains over NP and DART-r, previously demonstrated in the discrete-event simulator, remain consistent when deployed on a real-world testbed.

7.4 Ablation Study: Benefit of Resource Reservation

As discussed in §5.4, PPipe's resource reservation-based data plane dynamically schedules requests onto GPUs to overcome the delays caused by bursty request arrival and network contention. To demonstrate its effectiveness, we compare it to a *reactive*, distributed adaptive batching scheduler. The reactive scheduler performs adaptive batching independently for each GPU pool in a pooled pipeline. For each pool, it selects the largest possible batch size that meets the pool's SLO (as determined by the MILP solver). A similar idea was used in previous model-granularity pipeline scheduling in [47].

Figure 10 shows the maximum load factor achieved by the two data plane designs under 99% SLO attainment on HC2-L, averaged over the DNNs, under the "Poisson" arrival. PPipe achieves an average load factor of 0.92, while reactive only achieves 0.71. The primary factor contributing to this performance degradation is that the reactive, distributed scheduler

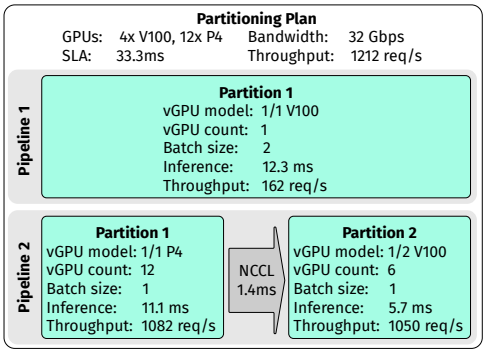


Figure 11: Partitioning plan for the FCN model on HC3-S.

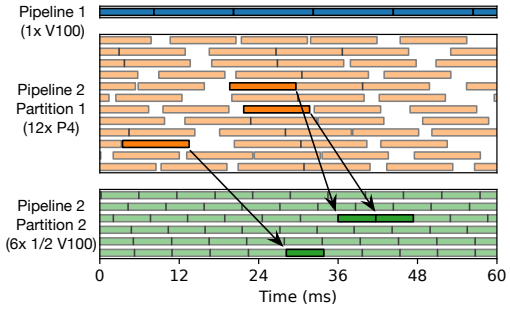


Figure 12: An example timeline serving the FCN model on HC3. Each row represents a vGPU, and each box corresponds to a batched inference. The highlighted pairs of boxes denote the same batches across the partitions within a pooled pipeline.

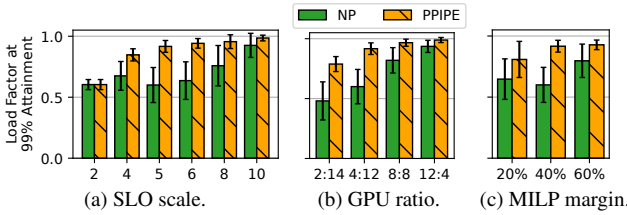
lacks resource usage tracking which leads to batches being scheduled onto servers with saturated network links, causing bloated transfer delays. For example, for EfficientNet-B8, where PPipe achieves 99% SLO attainment at 1.0 load factor, the reactive scheduler only manages a 0.35 load factor. With 10 Gbps effective bandwidth, feature map transfer between the first and second partition should take 5.1 ms, but the reactive scheduler leads to transfer times of 18.9 ms on average and 35.7 ms at the 99% percentile, resulting in excessive request drops at the second partition in order for the remaining requests to meet their SLOs.

7.5 Microscopic Analysis

In this section, we provide a microscopic analysis of PPipe, using the FCN model served on the HC3-S cluster testbed on Google Cloud as an example, where PPipe achieves a load factor of 0.95 under 99% SLO attainment.

Plan structure. Figure 11 shows the partitioning plan generated by PPipe's MILP solver for cluster HC3-S, which consists of 4× V100 and 12× P4 GPUs. The inference latency of the FCN model on the fastest GPU (NVIDIA L4) is 6.66 ms, establishing an SLO of 33.3 ms under an SLO scale of 5. The resulting plan comprises two pipelines, one with a single partition and the other with two partitions.

The first pipeline consists of a single V100 GPU, performing inference with a batch size of 2, where each batched inference takes 12.3 ms. The theoretical throughput of this pipeline is $(2 \times 1/0.0123) = 162$ requests per second. The second pipeline consists of two partitions, with 12 P4 and 3 V100 GPUs respectively, with 1.4 ms of feature map transfer time in between. Employing batch size unification (§5.3), both partitions perform batch size 1 inference. To achieve such a unified batch size, PPipe divides the three V100s in the second partition into six virtual GPUs using NVIDIA MPS. Furthermore, we observe that the two partitions yield similar throughputs of 1082 and 1050 requests per second respectively, resulting in a total throughput of 1050 (the minimum of the two). This shows PPipe's ability to balance resource allocation between partitions to achieve matched throughput.



(a) SLO scale. (b) GPU ratio. (c) MILP margin.

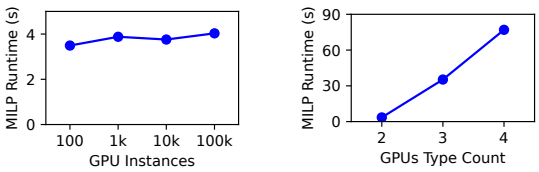
Figure 13: Sensitivity of PPipe to various factors. Results are averaged over 18 DNNs on HC1-S.

Runtime behavior. Figure 12 shows an example timeline of the DNN inference on each virtual GPU. We observe that PPipe performs inference back-to-back in pipeline 1, as well as pipeline 2 partition 2, fully using their GPUs. Note that pipeline 2 partition 1 experiences underutilization, due to the fact that it was provisioned with slightly higher serving throughput than partition 2 (Figure 11). Furthermore, the figure showcases that PPipe's pool-based pipeline allows a batch to be processed by any GPU within each partition, allowing different partitions to use different numbers of GPUs to accommodate different per-partition inference latencies.

7.6 Sensitivity Analysis

In this section, we study PPipe's sensitivity to various factors, on the cluster HC1-S testbed deployed on Google Cloud.

Varying SLO scales. Our main evaluation, which follows AlpaServe [34], uses 5x the inference latency as the SLO and shows significant performance improvement. We further evaluate PPipe considering various SLO scales ranging from 2x to 10x, as shown in Figure 13a. Although 2x and 10x SLOs are less used in practice, we include them to illustrate and validate PPipe's expected behavior relative to the baselines. With an SLO scale of 2, PPipe shows no improvement over NP, as such stringent SLOs render the utilization of low-class GPUs impractical for either NP or PPipe. Consequently, PPipe resorts to running entire DNNs on high-class GPUs, essentially falling back to NP. Conversely, as the SLO scale increases to 10, PPipe's improvement over NP becomes marginal, due to the fact that more DNNs can now meet the SLO running



(a) Varying GPU instances. (b) Varying GPU types.

Figure 14: Scalability of PPipe's MILP-based control plane.

entirely on low-class GPUs, and hence the low-class GPUs can be utilized in NP, thereby narrowing the gap between NP and PPipe.

Varying GPU ratios. In Figure 13b, we evaluate PPipe under varying ratios of high-class (NVIDIA L4) to low-class (NVIDIA P4) GPUs, which shows that PPipe attains more improvements over NP on clusters with fewer high-class GPUs. For instance, with a high-low ratio of 2:14, PPipe achieves a 64.03% higher load factor; as the percentage of high-class GPUs increases, PPipe's improvement diminishes, reaching 5.64% at a high-low ratio of 12:4.

Varying SLO margin size. As discussed in §7.1, a 40% margin was subtracted from the SLO in the MILP formulation of both PPipe and NP. The impact of the margin size is two-fold — a larger margin size reduces the ideal-case serving capacity (*i.e.,* what load factor 1.0 signifies), but increases the load factor achievable under 99% SLO attainment in practice. Figure 13c shows that under varying margin sizes, PPipe's load factor increases with larger margin sizes, but plateaus as the margin size increases beyond 40%. Furthermore, PPipe achieves the highest gain of 52.8% over NP under the margin size of 40%, but also maintains a relatively high improvement over NP at 20% and 60% margin sizes, of 24.9% and 16.4% respectively.

Varying GPU types and counts. We analyze how the runtime of PPipe's MILP-based control plane scales with the number of GPU instances and types in a cluster. First, we scale up the HC1-L cluster (Table 1) from 100 to 100k GPUs. Figure 14a shows that the MILP runtime remains nearly constant. This is because adding more GPU instances does not introduce additional MILP variables, and the problem's complexity stays unchanged. Second, as the GPU types increase to 3 and 4, Figure 14b shows the MILP runtime increases to 35.3 and 77.0 seconds, indicating the number of GPU types has a higher impact on MILP runtime. However, the increased runtime remains insignificant compared to the interval between MILP re-executions, which occurs on an hourly scale [28, 68].

8 Related Work

Pipelined partitioned DNN serving. Several works on DNN serving exploit pipeline parallelism for different scenarios and objectives. AlpaServe [34] improves serving throughput by employing pipeline parallelism to facilitate multiplexing of GPUs across multiple DNNs, but it does not take advantage of heterogeneous GPUs. Another line of works specialize in serving RNN or transformer-based models [16, 58] in a pipelined fashion. These works do not consider heterogeneous GPU clusters either. Finally, DART [55] partitions DNNs onto a chain of CPUs and GPUs. However, it does not scale to large GPU clusters, as it will generate as many partitions as the number of GPUs in the cluster, resulting in frequent transfer of feature maps which is costly.

Whole DNN serving on heterogeneous clusters. Such works typically focus on serving multiple DNNs concurrently, and study the placement of whole DNNs on servers with the goal of either minimizing the cost of cloud VMs [30, 45, 60], or maximizing the throughput of on-premise clusters [28, 31]. Furthermore, several works study serving video analytics applications that utilize multiple DNNs forming a Directed Acyclic Graph (DAG) [11, 24, 27]. None of above works exploits the diversity across the layers within a DNN.

Whole DNN serving, homogeneous clusters. Several works focus on model serving on homogeneous clusters [12, 20, 34, 47, 62]. However, these works lack resource allocation mechanisms that leverage DNN partitioning or GPU heterogeneity.

Model parallel DNN training. Various works exploit tensor or pipeline parallelism (or both) in model training [14, 25, 29, 32, 39, 43, 52, 56, 67]. Compared to inference, training has a different set of scenarios and requirements, *e.g.,* no need to meet SLOs or tackle non-deterministic request arrivals.

9 Conclusion

In this paper, we presented PPipe, a system for making effective use of mixed GPUs on heterogeneous clusters in serving video analytics applications. The key innovation of PPipe is three-fold: pool-based pipelined model inference, an MILP-based control plane that prescribes optimal pipeline plans, and a data plane that performs resource reservation-based adaptive batching to handle runtime dynamics due to asynchronous and bursty request arrivals. Evaluation results on production workloads show that PPipe achieves 32.2%–75.1% higher serving throughput compared to various baselines. In future work, we will explore extending PPipe to support transformer-based models, for instance, by partitioning them across high- and low-class GPUs. This would improve the utilization of low-class GPUs and enhance the overall serving throughput, while still meeting SLO requirements.

Acknowledgments

We thank the anonymous reviewers and our shepherd Purushottam (Puru) Kulkarni for their helpful comments. This work is supported in part by NSF grants 2112778, 2211459, and 2415216.

References

[1] The most surveilled cities in the world. https://www.usnews.com/news/cities/articles/2020-08-14/the-top-10-most-surveilled-cities-in-the-world, Last accessed, April 1, 2024.

[2] This is the most heavily surveilled city in the US: 50 CCTV cameras per 1,000 citizens. https://cybernews.com/editorial/this-is-the-most-heavily-surveilled-city-in-the-us-50-cctv-cameras-per-1000-citizens/, Last accessed, April 1, 2024.

[3] One Legacy of Tiananmen: China's 100 Million Surveillance Cameras. https://www.wsj.com/articles/BL-CJB-22562, Last accessed, December 1, 2021.

[4] One Surveillance Camera for Every 11 People in Britain, Says CCTV Survey. https://www.telegraph.co.uk/technology/10172298/One-surveillance-camera-for-every-11-people-in-Britain-says-CCTV-survey.html, Last accessed, December 1, 2024.

[5] A World With a Billion Cameras Watching You Is Just Around the Corner. https://www.wsj.com/articles/a-billion-surveillance-cameras-forecast-to-be-watching-within-two-years-11575565402, Last accessed, December 6, 2019.

[6] Ahsan Ali, Riccardo Pinciroli, Feng Yan, and Evgenia Smirni. Batch: Machine learning inference serving on serverless platforms with adaptive batching. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15, 2020.

[7] Azure Private Multi-access Edge Compute (MEC), 2023. https://azure.microsoft.com/en-us/solutions/private-multi-access-edge-compute-mec.

[8] Azure Cognitive Service for Vision. https://azure.microsoft.com/en-us/products/cognitive-services/vision-services/.

[9] Yue Cao, Jiarui Xu, Stephen Lin, Fangyun Wei, and Han Hu. Gcnet: Non-local networks meet squeeze-excitation networks and beyond. In *Proceedings of the IEEE/CVF International Conference on Computer Vision Workshops*, pages 0–0, 2019.

[10] Nvidia Corporation. NVIDIA TensorRT, 2017. https://developer.nvidia.com/tensorrt.

[11] Daniel Crankshaw, Gur-Eyal Sela, Xiangxi Mo, Corey Zumar, Ion Stoica, Joseph Gonzalez, and Alexey Tumanov. Inferline: Latency-aware provisioning and scaling for prediction serving pipelines. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, SoCC '20, page 477–491, New York, NY, USA, 2020. Association for Computing Machinery.

[12] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J. Franklin, Joseph E. Gonzalez, and Ion Stoica. Clipper: A Low-Latency online prediction serving system. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 613–627, Boston, MA, March 2017. USENIX Association.

[13] Xiaohan Ding, Xiangyu Zhang, Ningning Ma, Jungong Han, Guiguang Ding, and Jian Sun. Repvgg: Making vgg-style convnets great again. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 13733–13742, 2021.

[14] Yifan Ding, Nicholas Botzer, and Tim Weninger. Hetseq: Distributed gpu training on heterogeneous infrastructure. *Proceedings of the AAAI Conference on Artificial Intelligence*, 35(17):15432–15438, May 2021.

[15] Kaiwen Duan, Song Bai, Lingxi Xie, Honggang Qi, Qingming Huang, and Qi Tian. Centernet: Keypoint triplets for object detection. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 6569–6578, 2019.

[16] Jiarui Fang, Yang Yu, Chengduo Zhao, and Jie Zhou. Turbotransformers: an efficient gpu serving system for transformer models. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 389–402, 2021.

[17] Inc Global Industry Analysts. Intelligent video analytics - global strategic business report, October 2024.

[18] Google Distributed Cloud Edge, 2023. https://cloud.google.com/distributed-cloud/edge/latest/docs/gpu.

[19] GPU Performance Background User's Guide, 2023. https://docs.nvidia.com/deeplearning/performance/dl-performance-gpu-background/index.html.

[20] Arpan Gujarati, Reza Karimi, Safya Alzayat, Wei Hao, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace. Serving {DNNs} like clockwork: Performance predictability from the bottom up. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 443–462, 2020.

[21] Gurobi Optimization, LLC. Gurobi Optimizer, 2008. https://www.gurobi.com/solutions/gurobi-optimizer/.

[22] Junjun He, Zhongying Deng, Lei Zhou, Yali Wang, and Yu Qiao. Adaptive pyramid context network for semantic segmentation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 7519–7528, 2019.

[23] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016.

[24] Yitao Hu, Rajrup Ghosh, and Ramesh Govindan. Scrooge: A cost-effective deep learning inference system. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '21, page 624–638, New York, NY, USA, 2021. Association for Computing Machinery.

[25] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, et al. Gpipe: Efficient training of giant neural networks using pipeline parallelism. In *Advances in Neural Information Processing Systems*, volume 32, 2019.

[26] Junchen Jiang, Ganesh Ananthanarayanan, Peter Bodik, Siddhartha Sen, and Ion Stoica. Chameleon: scalable adaptation of video analytics. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 253–266, 2018.

[27] Ram Srivatsa Kannan, Lavanya Subramanian, Ashwin Raju, Jeongseob Ahn, Jason Mars, and Lingjia Tang. Grandslam: Guaranteeing slas for jobs in microservices execution frameworks. In *Proceedings of the Fourteenth EuroSys Conference 2019*, pages 1–16, 2019.

[28] Liu Ke, Udit Gupta, Mark Hempstead, Carole-Jean Wu, Hsien-Hsin S. Lee, and Xuan Zhang. Hercules: Heterogeneity-aware inference serving for at-scale personalized recommendation. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 141–154, 2022.

[29] Jin Kyu Kim, Qirong Ho, Seunghak Lee, Xun Zheng, Wei Dai, Garth A Gibson, and Eric P Xing. Strads: A distributed framework for scheduled model parallel machine learning. In *Proceedings of the Eleventh European Conference on Computer Systems*, pages 1–16, 2016.

[30] Baolin Li, Rohan Basu Roy, Tirthak Patel, Vijay Gadepally, Karen Gettings, and Devesh Tiwari. Ribbon: Cost-effective and qos-aware deep learning model inference using a diverse pool of cloud computing instances. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '21, New York, NY, USA, 2021. Association for Computing Machinery.

[31] Baolin Li, Siddharth Samsi, Vijay Gadepally, and Devesh Tiwari. Kairos: Building cost-efficient machine learning inference systems with heterogeneous cloud resources. In *Proceedings of the 32nd International Symposium on High-Performance Parallel and Distributed Computing*, pages 3–16, 2023.

[32] Dacheng Li, Hongyi Wang, Eric Xing, and Hao Zhang. Amp: Automatically finding model parallel strategies with heterogeneity awareness. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, editors, *Advances in Neural Information Processing Systems*, volume 35, pages 6630–6639. Curran Associates, Inc., 2022.

[33] Xiang Li, Wenhai Wang, Lijun Wu, Shuo Chen, Xiaolin Hu, Jun Li, Jinhui Tang, and Jian Yang. Generalized focal loss: Learning qualified and distributed bounding boxes for dense object detection. *Advances in Neural Information Processing Systems*, 33:21002–21012, 2020.

[34] Zhuohan Li, Lianmin Zheng, Yinmin Zhong, Vincent Liu, Ying Sheng, Xin Jin, Yanping Huang, Zhifeng Chen, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. AlpaServe: Statistical multiplexing with model parallelism for deep learning serving. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 663–679, Boston, MA, July 2023. USENIX Association.

[35] Zhuang Liu, Hanzi Mao, Chao-Yuan Wu, Christoph Feichtenhofer, Trevor Darrell, and Saining Xie. A convnet for the 2020s. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 11976–11986, 2022.

[36] Jonathan Long, Evan Shelhamer, and Trevor Darrell. Fully convolutional networks for semantic segmentation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 3431–3440, 2015.

[37] Chengqi Lyu, Wenwei Zhang, Haian Huang, Yue Zhou, Yudong Wang, Yanyi Liu, Shilong Zhang, and Kai Chen. Rtmdet: An empirical study of designing real-time object detectors. *arXiv preprint arXiv:2212.07784*, 2022.

[38] Multi-Process Service, 2023. https://docs.nvidia.com/deploy/mps/index.html.

[39] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R Devanur, Gregory R Ganger, Phillip B Gibbons, and Matei Zaharia. Pipedream: Generalized pipeline parallelism for dnn training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 1–15, 2019.

[40] NVIDIA Hopper Architecture, 2023. https://www.nvidia.com/en-us/data-center/technologies/hopper-architecture/.

[41] OpenMMLab Open Platform. https://platform.openmmlab.com/home/.

[42] OpenVINO Model Zoo. https://docs.openvino.ai/2023.2/model_zoo.html.

[43] Jay H Park, Gyeongchan Yun, M Yi Chang, Nguyen T Nguyen, Seungmin Lee, Jaesik Choi, Sam H Noh, and Young-ri Choi. Hetpipe: Enabling large dnn training on (whimpy) heterogeneous gpu clusters through integration of pipelined model parallelism and data parallelism. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 307–321, 2020.

[44] Heyang Qin, Syed Zawad, Yanqi Zhou, Lei Yang, Dongfang Zhao, and Feng Yan. Swift machine learning model serving scheduling: A region based reinforcement learning approach. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '19, New York, NY, USA, 2019. Association for Computing Machinery.

[45] Francisco Romero, Qian Li, Neeraja J. Yadwadkar, and Christos Kozyrakis. INFaaS: Automated model-less inference serving. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 397–411. USENIX Association, July 2021.

[46] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 205–218, 2020.

[47] Haichen Shen, Lequn Chen, Yuchen Jin, Liangyu Zhao, Bingyu Kong, Matthai Philipose, Arvind Krishnamurthy, and Ravi Sundaram. Nexus: A gpu cluster engine for accelerating dnn-based video analysis. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 322–337, New York, NY, USA, 2019. Association for Computing Machinery.

[48] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1–9, 2015.

[49] Mingxing Tan and Quoc Le. EfficientNet: Rethinking model scaling for convolutional neural networks. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 6105–6114. PMLR, 09–15 Jun 2019.

[50] Mingxing Tan, Ruoming Pang, and Quoc V Le. Efficientdet: Scalable and efficient object detection. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 10781–10790, 2020.

[51] Torchvision Models and Pretrained Weights. https://pytorch.org/vision/stable/models.html.

[52] Taegeon Um, Byungsoo Oh, Minyoung Kang, Woo-Yeon Lee, Goeun Kim, Dongseob Kim, Youngtaek Kim, Mohd Muzzammil, and Myeongjae Jeon. Metis: Fast automatic distributed training on heterogeneous gpus. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*, pages 563–578, 2024.

[53] Xiaolong Wang, Ross Girshick, Abhinav Gupta, and Kaiming He. Non-local neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 7794–7803, 2018.

[54] Qizhen Weng, Wencong Xiao, Yinghao Yu, Wei Wang, Cheng Wang, Jian He, Yong Li, Liping Zhang, Wei Lin, and Yu Ding. {MLaaS} in the wild: Workload analysis and scheduling in {Large-Scale} heterogeneous {GPU} clusters. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 945–960, 2022.

[55] Yecheng Xiang and Hyoseung Kim. Pipelined data-parallel cpu/gpu scheduling for multi-dnn real-time inference. In *2019 IEEE Real-Time Systems Symposium (RTSS)*, pages 392–405. IEEE, 2019.

[56] Xiaodong Yi, Shiwei Zhang, Ziyue Luo, Guoping Long, Lansong Diao, Chuan Wu, Zhen Zheng, Jun Yang, and Wei Lin. Optimizing distributed training deployment in heterogeneous gpu clusters. In *Proceedings of the 16th International Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '20, page 93–107, New York, NY, USA, 2020. Association for Computing Machinery.

[57] Minghao Yin, Zhuliang Yao, Yue Cao, Xiu Li, Zheng Zhang, Stephen Lin, and Han Hu. Disentangled non-local neural networks. In *Computer Vision–ECCV 2020: 16th European Conference, Glasgow, UK, August 23–28, 2020, Proceedings, Part XV 16*, pages 191–207. Springer, 2020.

[58] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. Orca: A distributed serving system for {Transformer-Based} generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 521–538, 2022.

[59] Sergey Zagoruyko and Nikos Komodakis. Wide residual networks. *arXiv preprint arXiv:1605.07146*, 2016.

[60] Chengliang Zhang, Minchen Yu, Wei Wang, and Feng Yan. MArk: Exploiting cloud services for Cost-Effective, SLO-Aware machine learning inference serving. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 1049–1062, Renton, WA, July 2019. USENIX Association.

[61] Hang Zhang, Kristin Dana, Jianping Shi, Zhongyue Zhang, Xiaogang Wang, Ambrish Tyagi, and Amit Agrawal. Context encoding for semantic segmentation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 7151–7160, 2018.

[62] Hong Zhang, Yupeng Tang, Anurag Khandelwal, and Ion Stoica. SHEPHERD: Serving DNNs in the wild. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 787–808, Boston, MA, April 2023. USENIX Association.

[63] Richard Zhang, Phillip Isola, and Alexei A Efros. Colorful image colorization. In *Computer Vision–ECCV 2016: 14th European Conference, Amsterdam, The Netherlands, October 11-14, 2016, Proceedings, Part III 14*, pages 649–666. Springer, 2016.

[64] Shifeng Zhang, Cheng Chi, Yongqiang Yao, Zhen Lei, and Stan Z Li. Bridging the gap between anchor-based and anchor-free detection via adaptive training sample selection. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 9759–9768, 2020.

[65] Wuyang Zhang, Zhezhi He, Luyang Liu, Zhenhua Jia, Yunxin Liu, Marco Gruteser, Dipankar Raychaudhuri, and Yanyong Zhang. Elf: accelerate high-resolution mobile deep vision with content-aware parallel offloading. In *Proceedings of the 27th Annual International Conference on Mobile Computing and Networking*, pages 201–214, 2021.

[66] Yanqi Zhang, Íñigo Goiri, Gohar Irfan Chaudhry, Rodrigo Fonseca, Sameh Elnikety, Christina Delimitrou, and Ricardo Bianchini. Faster and cheaper serverless computing on harvested resources. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 724–739, 2021.

[67] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P Xing, et al. Alpa: Automating inter-and {Intra-Operator} parallelism for distributed deep learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 559–578, 2022.

[68] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. {DistServe}: Disaggregating prefill and decoding for goodput-optimized large language model serving. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 193–210, 2024.

[69] Chenchen Zhu, Yihui He, and Marios Savvides. Feature selective anchor-free module for single-shot object detection. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 840–849, 2019.

Appendix

A.1 Mathematical Representation of Basic MILP Formulation

As shown in Table 3, the MILP formulation takes as input the cluster configuration, the latency SLO, and profiling information of the target DNN model. For the convenience of formulating mathematical constraints, the raw inputs are further processed and transformed into different representations. Table 4 lists both output decision variables and intermediate decision variables that will only be used inside the MILP formulation. The MILP solution outputs the partition points for each pipeline, as well as the batch size and number of GPUs used by each partition. The MILP formulation maximizes the total inference throughput across all pipelines in the cluster:

$$\text{maximize} \quad \sum_l x_l \tag{1}$$

The optimization is under the constraint that inference latency of each pipeline does not exceed the latency SLO, and the total number of GPUs allocated across partitions does not exceed the cluster configuration. The constraints are formally formulated below.

Table 3: Inputs to the MILP formulation (top) and values derived from inputs (bottom).

Input	Description
N_k	GPU count of GPU class k
T	The latency SLO
L_{kbi}	The inference latency of layer i under batch size b on GPU class k
S_i	The output feature map size of layer i under batch size 1
D_l	Number of partitions in pipeline l
G_k	A list of tuples (l,d) indicating GPU class k is used for partition d of pipeline l
M	Number of layers in the DNN model
C_{ldbij}	The inference latency of DNN partition consisting of layers i to j (exclusive) with batch size b on GPU class associated with (l,d)
X_{ldbij}	The inference throughput of DNN partition consisting of layers i to j (exclusive) with batch size b on a single GPU associated with (l,d)
Y_{bj}	The transfer latency of the feature map of layer $j-1$ with batch size b

Table 4: Output (top) and intermediate (bottom) decision variables in the MILP formulation.

Variable	Description
$p_{ldbij} \in \{0,1\}$	Whether partition d in pipeline l spans from layer i to j (exclusive) and runs at batch size b
$g_{ldbij} \in \mathbb{N}$	Number of GPUs used by partition d in pipeline l
$t_{ld} \in \mathbb{R}_{\geq 0}$	Inference latency of partition d in pipeline l
$x_{ld} \in \mathbb{R}_{\geq 0}$	Inference throughput of partition d in pipeline l
$n_{ld} \in \mathbb{R}_{\geq 0}$	Transfer latency between partition d and partition $d+1$ in pipeline l
$x_l \in \mathbb{R}_{\geq 0}$	Inference throughput of pipeline l

$$\sum_{bij} p_{ldbij} = 1 \qquad \forall l,d \tag{2}$$

$$p_{ldbij} = 0 \qquad \forall l,d,b,i \geq j \tag{3}$$

$$\sum_{bi} p_{ldbij} = 1 \rightarrow \sum_{b'j'} p_{ld'b'i'j'} = 1 \qquad \forall l,d' = d+1, i' = j \tag{4}$$

$$\sum_{bj} p_{ldbij} = 1 \qquad \forall l,d = 0, i = 0 \tag{5}$$

$$\sum_{bi} p_{ldbij} = 1 \qquad \forall l,d = D_l - 1, j = M \tag{6}$$

$$p_{ldbij} = 0 \rightarrow g_{ldbij} = 0 \qquad \forall l,d,b,i,j \tag{7}$$

$$p_{ldbij} = 1 \rightarrow g_{ldbij} \geq 1 \qquad \forall l,d,b,i,j \tag{8}$$

$$\sum_{bij,(l,d)\in G_k} g_{ldbij} \leq N_k \qquad \forall k \tag{9}$$

$$t_{ld} = \sum_{bij} C_{ldbij} \cdot p_{ldbij} \qquad \forall l,d \tag{10}$$

$$x_{ld} = \sum_{bij} X_{ldbij} \cdot g_{ldbij} \qquad \forall l,d \tag{11}$$

$$n_{ld} = \sum_{bij} Y_{bj} \cdot p_{ldbij} \qquad \forall l,d \tag{12}$$

$$\sum_d t_{ld} + \sum_d n_{ld} \leq T \qquad \forall l \tag{13}$$

$$x_l = \min_d x_{ld} \qquad \forall l \tag{14}$$

Equations (2)–(6) ensure DNN partitions are well formed, *i.e.,* partitions cannot be empty, the last and first layers in adjacent partitions must also be adjacent, and the first partition within a pipeline must start with the first layer, while the opposite applies to the last partition. Equation (9) represents the constraint on the total GPU count. Equations (10)–(12) calculates for each partition the inference latency, inference throughput, and transfer latency, respectively. Finally, Equation (13) enforces the latency SLO constraint.

The formulation can be easily scaled to the case of multiple DNN models, where each DNN model has its own set of decision variables and constraints, with the additional constraint that the total number of GPUs allocated to all DNN models does not exceed the cluster configuration.

A.2 Mathematical Representation of MILP Formulation with Batch Size Unification

Table 5: Inputs to the MILP formulation (with batch size unification) (top) and values derived from inputs (bottom).

Input	Description
N_k	GPU count of GPU class k
T	The latency SLO
L_{kvbi}	The inference latency of block i under batch size b on virtual GPU of size $1/v$ and GPU class k
S_i	The output feature map size of block i under batch size 1
D_l	Number of blocks in pipeline l
G_k	A list of tuples (l,d) indicating GPU class k is used for partition d of pipeline l
M	Number of layers in the DNN model
C_{ldvbij}	The inference latency of DNN partition consisting of blocks i to j (exclusive) with batch size b on $1/v$ virtual GPU of GPU class associated with (l,d)
X_{ldvbij}	The inference throughput of DNN partition consisting of blocks i to j (exclusive) with batch size b on $1/v$ virtual GPU of GPU class associated with (l,d)
Y_{bj}	The transfer latency of the feature map of block $j-1$ with batch size b

Table 6: Output (top) and intermediate (bottom) decision variables in the MILP formulation with batch size unification.

Variable	Description
$p_{ldvbij} \in \{0,1\}$	Whether partition d in pipeline l spans from block i to j (exclusive) and runs at batch size b on $1/v$ virtual GPU
$g_{ldvbij} \in \mathbb{N}$	Number of virtual GPUs used by partition d in pipeline l
$t_{ld} \in \mathbb{R} \geq 0$	Inference latency of partition d in pipeline l
$x_{ld} \in \mathbb{R}_{\geq 0}$	Inference throughput of partition d in pipeline l
$n_{ld} \in \mathbb{R}_{\geq 0}$	Transfer latency between partition d and partition $d+1$ in pipeline l
$x_l \in \mathbb{R}_{\geq 0}$	Inference throughput of pipeline l

The inputs and decision variables to the MILP formulation with batch size unification (Table 5 and Table 6) are similar to that of the basic MILP formulation (§A.1), with the exception that both the model profiling inputs and decision variables now include an additional dimension representing virtual GPUs, and that the profiling inputs are for blocks instead of layers, and the same applies to the last two dimensions of the output decision variables. The MILP formulation optimizes for the same throughput objective:

$$\text{maximize} \quad \sum_l x_l \tag{15}$$

The optimization is under a similar set of constraints as shown below.

$$\sum_{vbij} p_{ldvbij} = 1 \qquad \forall l,d \tag{16}$$

$$p_{ldvbij} = 0 \qquad \forall l,d,v,b,i \geq j \tag{17}$$

$$\sum_{vi} p_{ldvbij} = 1 \rightarrow \sum_{v'j'} p_{ld'v'bi'j'} = 1 \quad \forall l,b,\ d' = d+1,\ i' = j \tag{18}$$

$$\sum_{vbj} p_{ldvbij} = 1 \qquad \forall l,d=0,i=0 \tag{19}$$

$$\sum_{vbi} p_{ldvbij} = 1 \quad \forall l,d = D_l - 1, j = M \tag{20}$$

$$p_{ldvbij} = 0 \rightarrow g_{ldvbij} = 0 \qquad \forall l,d,v,b,i,j \tag{21}$$

$$p_{ldvbij} = 1 \rightarrow g_{ldvbij} \geq 1 \qquad \forall l,d,v,b,i,j \tag{22}$$

$$\sum_{vbij,(l,d)\in G_k} g_{ldvbij}/v \leq N_k \qquad \forall k \tag{23}$$

$$t_{ld} = \sum_{vbij} C_{ldvbij} \cdot p_{ldvbij} \qquad \forall l,d \tag{24}$$

$$x_{ld} = \sum_{vbij} X_{ldvbij} \cdot g_{ldvbij} \qquad \forall l,d \tag{25}$$

$$n_{ld} = \sum_{vbij} Y_{bj} \cdot p_{ldvbij} \qquad \forall l,d \tag{26}$$

$$\sum_d t_{ld} + \sum_d n_{ld} \leq T \qquad \forall l \tag{27}$$

$$x_l = \min_d x_{ld} \qquad \forall l \tag{28}$$

The major differences lie in Equations (18) and (23). In Equations (18), the dimension b is not summed over, which enforces that the same batch size used by the first partition will also need to be used by the next partition. In Equations (23), the decision variable g represents the count of *virtual* GPUs, thus, dividing it by v gives us the number of *physical* GPUs that the partition uses (note that v is not a decision variable and such divisions are allowed in MILP).

A.3 The Resource Reservation-Based Adaptive Batching Algorithm

Algorithm 1 shows the pseudocode for the resource reservation-based adaptive batching algorithm, which was explained in §5.4. To recap, the algorithm first identifies a pooled pipeline that can complete a batched inference at the pipeline's unified batch size bs_i with the shortest waiting time (lines 4–8). It then searches for the largest batch size that can satisfy the SLO (lines 9–14).

The two supporting functions used by §5.4, `probe()` and `reserve()`, is shown in Algorithm 2. The helper function `earliestSlot(res, t, l)` returns the earliest time (no earlier than t) when a list of resources `res` are free for duration l. First, since feature map transfer requires the network resources on both sending and receiving sides to be available at the same time, we use `earliestSlot(res, t, l)` to find the earlist available transfer slot that works for both the last GPU's uplink and current GPU's downlink (lines 12–14). After reserving the two links for feature map transfer, we update current time t and then find and reserve the earliest available inference time slot for the current GPU (lines 15–16).

Algorithm 1: Resource reservation-based adaptive batching.

```
1  Inputs: q (pending requests sorted by arrival);
2          pooled_pipelines (pooled pipelines in the cluster);
3  while true do
       // choose the pooled pipeline
4      p* = nil, t* = ∞;
5      for p in pooled_pipelines do
6          r = probe(p, p.bs);
7          if waitTime(r) < t* then
8              t* = waitTime(r), p* = p;
       // choose the pipeline path and batch size
9      bs* = nil, path* = nil, resv* = nil;
10     for bs = p*.bs down to 0 do
11         path, resv = probe(p*, bs);
12         if finishTime(resv) ≤ q[0].deadline then
13             bs* = bs, path* = path, resv* = resv;
14             break;
       // perform request drop, wait, or dispatch
15     if bs == 0 then
16         drop q[0];
17     else if q.length < bs then
18         wait for more requests till requests in q are about to
             miss deadline;
19     else
20         reserve(r);
21         dispatch first bs requests in q according to r;
```

Algorithm 2: Resource reservation functions.

```
1  function probe(pooled_pipeline, bs)
2      Inputs: The assumed selected pooled pipeline and the
         batch size;
3      Outputs: The optimal pipeline path and the resources
         that need to be reserved;
4      t_g = now(), path = [], resv = [];
5      for partition in pooled_pipeline do
6          l_n = calcNetLat(partition, bs);
7          l_i = calcInferenceLat(partition, bs);
8          t* = ∞, r* = [];
9          for gpu in partition do
10             t = t_g, r = [];
               // est. time to transfer feature map
11             if not first partition then
12                 u = lastGpu.netUL, d = gpu.netDL;
13                 t = earliestSlot([u, d], t, l_n);
14                 r += [{u, t, l_n}, {d, t, l_n}], t += l_n;
               // est. time to finish inference
15             t = earliestSlot([gpu], t, l_i);
16             r += [{gpu, t, l_i}], t += l_i;
17             if t < t* then
18                 t* = t, r* = r, gpu* = gpu;
19         t_g = t*, lastGpu = gpu*;
20         path += gpu*, resv += r*;
21     return (path, resv); // resource usage
22 function reserve(resv)
23     for {res, start, dur} in resv do
24         markReserved(res, start, dur);
```

Voltrix: Sparse Matrix-Matrix Multiplication on Tensor Cores with Asynchronous and Balanced Kernel Optimization

Yaqi Xia*
School of Computer Science
Wuhan University

Weihu Wang*
School of Computer Science
Wuhan University

Donglin Yang
NVIDIA Corporation

Xiaobo Zhou†
IOTSC
University of Macau

Dazhao Cheng†
School of Computer Science
Wuhan University

Abstract

Sparse Matrix-Matrix Multiplication (SpMM) is crucial in scientific computing and machine learning. Despite advancements in GPU architectures, efficiently leveraging Tensor Cores for SpMM remains challenging. The core issue is the mismatch between the inherently sparse nature of the matrices and the dense computational patterns. Existing methods struggle with substantial overheads in loading data to computation units and cannot adequately manage data imbalance across computations, thereby limiting the high computational throughput potential of Tensor Cores.

In this paper, we introduce Voltrix-SpMM, a revolutionary GPU kernel design that overcomes these challenges. First, we implement an asynchronous data loading pipeline that employs a bit-wise compressed format for sparse matrices and bulk memory copy instructions for dense matrices. This innovative design enables efficient data access and incorporates a warp-specialized producer-consumer model to seamlessly overlap data loading with computation. Second, we develop a persistent and I/O co-balanced kernel mechanism that features a two-stage partition strategy to achieve balance between input and output. Implemented with CUDA 12.6, Voltrix-SpMM substantially improves performance, delivering an average speedups of 36.5x and 1.8x over Tensor Core-based TC-GNN and DTC-SpMM respectively, and an average 1.7x speedup over the CUDA Core-based RoDe, fully unleashing the power of Tensor Cores for SpMM.

1 Introduction

Sparse Matrix-Matrix Multiplication (SpMM), which refers to the multiplication of a sparse matrix with a dense matrix, plays a pivotal role in a wide range of scientific computing applications [1,3,13,20,22,40,45–47,52], from simulations and linear algebra to advanced machine learning. Recent studies have identified SpMM as a significant performance bottleneck in these applications, particularly in the training of Graph Neural Networks (GNNs), where SpMM accounts for over 80% of the total computational cost during training [43].

GPUs have become the primary processing units for modern high-performance computing and have been widely adopted to accelerate SpMM computations. Recent studies have utilized CUDA cores, the fundamental processing units within NVIDIA's GPU architecture, to speed up the multiplication process [12,16,17,24,32]. With advancements in GPU power, NVIDIA introduced a specialized unit known as the Tensor Core, beginning with the Volta architecture [29]. Tensor Cores offer significantly higher computational throughput than traditional CUDA cores and are specifically optimized for dense matrix multiplication and accumulation operations using fixed-precision arithmetic. However, the inherent sparsity of matrices in SpMM presents challenges. Tensor Cores struggle with the irregular memory access and data sparsity typical of these operations, making them less effective for SpMM without specialized optimizations.

One of the prominent attempts to apply Tensor Core acceleration to SpMM is TC-GNN [43], which introduces sparse graph translation to convert sparse matrices into compressed dense blocks, called TCU blocks. Although this method overcomes the traditional barriers between sparse SpMM workloads and dense computation patterns, the inefficient process of loading data into Tensor Core units severely limits their full potential. Our experiments have shown that data loading consumes over 80% of kernel execution time, significantly constraining the effective use of Tensor Cores. Recently, DTC-SpMM introduces an asynchronous loading technique that overlaps computation with data loading through a pipeline [10]. However, the benefits of this approach are limited by several factors: firstly, DTC-SpMM's asynchronous loading instruction can only process 16-byte chunk at a time, which is insufficient for the high-dimensional data typical in applications like GNN training, where matrix sizes often exceed 256 [37]. This limitation necessitates numerous loading instructions, diminishing the pipeline's potential benefits. Secondly, the single-layer pipeline of DTC-SpMM results in

*Contributed equally to this work.
†Co-corresponding authors.

minimal overlap between computation and data loading, and the required synchronization at the warp level introduces additional overhead, further impairing the expected performance gains.

In addition to inefficient data loading, the irregular data distribution in SpMM further exacerbates performance bottlenecks during kernel execution on GPUs. A critical challenge lies in efficiently partitioning unevenly distributed data across different Streaming Multiprocessors (SMs) to achieve hardware load balancing. Current methods typically focus either on input balance or output balance. TC-GNN attempts to address output balance by controlling how each Cooperative Thread Array (CTA) processes rows of TCU blocks. However, this approach results in significant variations in the data processed per CTA, leading to underutilization of some SMs and overloading of others. Conversely, DTC-SpMM prioritizes input balance by assigning each CTA a fixed number of TCU blocks, aiming for a more uniform workload distribution. While this strategy reduces discrepancies, it fails to address imbalances during the write-back phase. Moreover, the atomic operations required to ensure accurate results during write-back introduce substantial overhead, thereby undermining the potential performance benefits of the optimization.

We introduce Voltrix-SpMM, a revolutionary GPU kernel design that addresses the outlined challenges and fully leverages the capabilities of Tensor Cores for accelerated SpMM computations. Voltrix-SpMM features two key innovations: First, a novel asynchronous data loading pipeline that establishes seamless overlap between computation and data loading. Unlike current SpMM kernels that manage these processes within the same warp, our model uses a warp-specialized workflow control to distinctly separate these tasks. This approach allows for fine-grained, multi-tiered overlap, significantly reducing the overhead associated with data loading. Second, a persistent and I/O co-balanced kernel mechanism that ensures even partitioning of irregular workloads across the underlying SM units. By integrating coarse-grained input partitioning with fine-grained output partitioning, we achieve an optimal balance between input and output, while eliminating the overhead linked to atomic operations.

The first innovation, the warp-level asynchronous pipeline, significantly reduces data loading overhead while achieving seamless overlap with computation. Specifically, we design a bit-wise compressed format for sparse matrices that is both vectorization-friendly and conflict-free, accelerating access and transformation. For dense matrices, we utilize the bulk asynchronous loading instruction TMA [30], which reduces the number of instructions while enabling the asynchronous loading of large data blocks. Leveraging our warp-specialized producer-consumer model, we decouple data access from computation at the warp level. This separation allows for finer granularity and higher overlap between data loading and computation through a multi-tiered pipeline.

In the second innovation, a persistent and I/O co-balanced kernel mechanism, we align each CTA task with SMs and ensure persistent execution, thus achieving a seamless software-to-hardware balance. Through a greedy and heuristic search algorithm, we determine the optimal partition points for each CTA task, ensuring co-balance between input and output. Additionally, by performing coarse-grained partitioning at the input stage and fine-grained partitioning at the output stage, we resolve the issue of data crossing row boundaries, avoiding the additional overhead of atomic operations.

In summary, our key contributions are as follows:

1. Through in-depth analysis, we identify the primary challenge in current Tensor Core-accelerated SpMM methods: a significant gap between high-throughput computation units and inefficient, heavy data loading processes.

2. We design a bit-wise sparse matrix compression format to enable efficient data access via vectorized loading, and a bulk asynchronous loading instruction to load dense matrices with reduced number of instructions.

3. We propose a warp-specialized producer-consumer model to efficiently pipeline data access and computation, along with a multi-tiered pipeline to effectively conceal the overhead associated with data loading.

4. We develop a persistent and I/O balanced kernel mechanism, ensuring uniform kernel execution across hardware that balances both input and output processing.

Voltrix-SpMM, implemented with CUDA 12.6, has been integrated into PyTorch 2.5 [33]. On the SuiteSparse dataset [7] and 12 real-world graph datasets, it achieves average speedups of 36.5x and 1.8x over the Tensor Core-based TC-GNN and DTC-SpMM respectively, and 1.7x over the CUDA Core-based RoDe [32]. Additionally, it delivers 2.0x speedup over the popular GNN training framework DGL [2] in end-to-end training. Notably, Voltrix-SpMM is the first to fully unleash the power of Tensor Cores, achieving speedups that surpass those of traditional CUDA Core-based methods.

2 Background and Motivation

2.1 Matrix-Matrix Multiplication on Hopper

General Matrix-Matrix Multiplication (GEMM) is a core operation in deep learning [39, 50]. On NVIDIA Hopper [30], GEMM performance is enhanced through warp-specialized and persistent kernel designs [31]. Warps are split into producers, which use Tensor Memory Accelerator (TMA) to load tiles into shared memory, and consumers, which perform Warp-Group Matrix Multiply-Accumulate (WGMMA) operations. This producer-consumer model enables overlap of data load and computation, maximizing Tensor Core utilization.

Hopper's persistent kernel strategy assigns a fixed number of Cooperative Thread Arrays (CTAs), typically equal to the

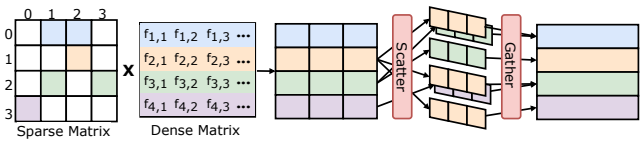


Figure 1: Illustration of SpMM workflow.

number of Streaming Multiprocessors (SMs), allowing them to remain resident and avoid relaunch overhead. This design also overlaps epilogue storage with the next stage's prologue and compute loop, enhancing pipeline efficiency.

Sparse Matrix-Matrix Multiplication (SpMM) multiplies a sparse matrix with a dense one, using only the non-zero elements [7]. Unlike GEMM, SpMM identifies non-zero indices to guide a `scatter` from the dense matrix and a `gather` for accumulation, as shown in Figure 1. However, due to sparsity and irregularity, SpMM cannot directly adopt GEMM's warp-specialized and persistent kernel designs: 1. TMA and WGMMA require dense data blocks, wasting resources on zeros. 2. Irregular non-zero patterns cause workload imbalance, making persistent kernels inefficient.

This paper introduces a BMat encoding for fast non-zero identification, a multi-tiered async loading scheme to maximize bandwidth, and a balanced partitioning strategy that enables an effective sparse-aware persistent kernel.

2.2 SpMM on Tensor Cores

NVIDIA Tensor Cores are specialized units that accelerate matrix operations essential for deep learning. They efficiently perform dense mixed-precision matrix multiplications, such as TF32 inputs with FP32 accumulation, computing

$$C = A \times B + C$$

where A and B have dimensions $m \times k$ and $k \times n$. Matrices are distributed across thread registers (fragments) within a warp, arranged in layouts like row- or column-major [29]. Applying Tensor Cores to SpMM is challenging because, unlike dense models (e.g., Transformers [19, 38] and CNNs [14, 21]) with regular memory access, SpMM exhibits irregular access and sparse computation patterns that conflict with the fixed-size fragment requirements of Tensor Cores.

TC-GNN [43] represents the first significant effort to bridge the gap between SpMM and Tensor Core capabilities by introducing a novel sparse graph translation technique. As illustrated in the top part of Figure 2, this approach compresses a sparse matrix into multiple condensed sparse blocks, referred to as TCU blocks. Each block is carefully sized to align with the computational dimensions of Tensor Cores. This transformation allows TC-GNN to effectively map irregular computation workloads onto the regular computational framework of Tensor Cores, thereby enabling these specialized units to handle sparse computation tasks efficiently.

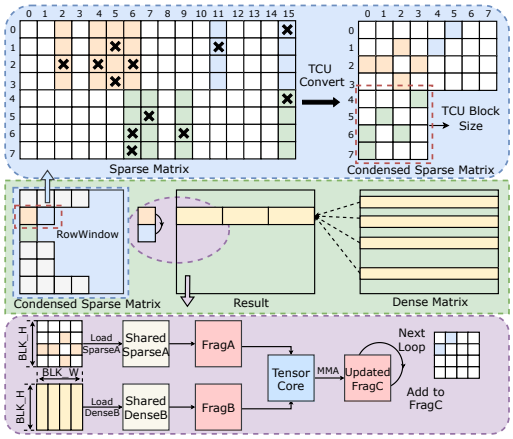


Figure 2: Illustration of the typical Tensor core-based SpMM implementation, TC-GNN's workflow.

In TC-GNN, the TCU block data for each row (a RowWindow) is processed iteratively. The workflow of TC-GNN operates as follows: First, the compressed TCU block is expanded into a BLK_W × BLK_H matrix, named SparseA, where BLK_W and BLK_H are specifically set to 8 and 16, respectively, to fit the dimensions of the Tensor Core MMA operations. Then, based on the column index of the TCU block, the corresponding row data from the dense matrix is identified and transformed into a BLK_W × BLK_H matrix, DenseB. Both matrices are first loaded into shared memory and then into registers for MMA operations, as depicted in the bottom part of Figure 2. The results are stored and subsequently used to update the output matrix C, which is accumulated with the next SparseA in the RowWindow in the subsequent loop iteration, ensuring the correctness of the results.

2.2.1 Tensor Core's Hunger: Starved by Data Loading

Utilizing Tensor Cores significantly enhances computational throughput for sparse operations, providing potential performance gains over traditional CUDA Cores. For instance, in optimized conditions, Tensor Cores can achieve up to 495 TFLOPS for TF32 datatype matrix multiplications, in contrast to the 67 TFLOPS for FP32 typically delivered by CUDA Cores in similar scenarios [30]. However, despite these impressive advancements, the overall efficiency of kernel execution is often bottlenecked by the data loading process. Our experiments across multiple datasets, as illustrated in Figure 3, reveal that data loading accounts for over 80% of the execution time. This significant overhead, due to loading matrices SparseA and DenseB into shared memory, severely restricts the overall computational efficiency, thereby diminishing the full potential of Tensor Core acceleration.

To address the data-loading bottleneck in Tensor Core-accelerated SpMM computation, DTC-SpMM [10] introduces a pivotal innovation: a pipeline arrangement that effectively

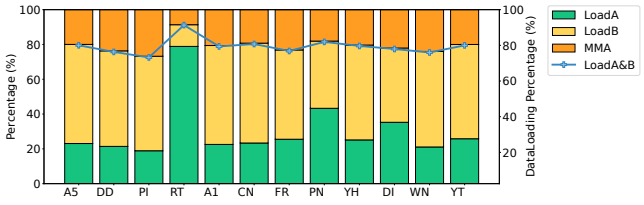


Figure 3: Breakdown of TC-GNN.

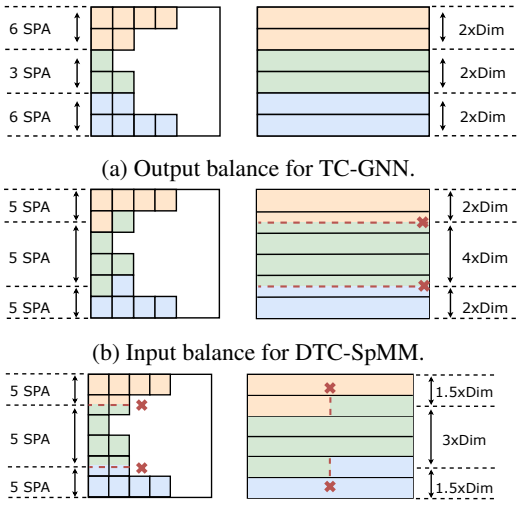
overlaps the loading of SparseA with Tensor Core computations, mitigating loading time. However, the typically large dimensions of the dense matrix in GNNs, often exceeding 256, mean one SparseA corresponds to multiple tiled DenseB matrices, creating a data copy overhead that significantly exceeds that of SparseA. Our experimental results, as shown in Figure 3 , reveal that in most cases, loading the dense matrix (DenseB) accounts for over 60% of the total execution time. Thus, despite the full overlap of SparseA loading, the primary challenge persists: the prolonged stalls caused by loading DenseB continue to dominate kernel execution time.

Furthermore, the pipelining strategy in DTC-SpMM struggles to effectively parallelize the loading of DenseB with MMA computations for several reasons: (1) DTC-SpMM leverages the `LDGSTS` SASS instruction [28] for asynchronous loading, which initiates data transfers without stalling computation. However, this instruction is limited to loading only 16 bytes of data per operation. When applied to the DenseB matrix, the substantial number of `LDGSTS` instructions required increases dramatically, leading to a high instruction count that undermines the benefits of pipelining. (2) The single-layer and intra-warp pipeline used by DTC-SpMM results in minimal overlap and frequent synchronization within warps, thereby reducing overall efficiency. (3) DTC-SpMM loads DenseB data directly into the registers, bypassing shared memory. This access pattern does not support asynchronous prefetching, further limiting the effectiveness of the pipeline.

Summary. Given these issues, there is an urgent need for a novel pipelining design that can effectively bridge the gap between lengthy data loading times and efficient Tensor Core computations. Such a design should enable enhanced overlap of memory copying with computation, ultimately addressing the persistent data-loading challenge.

2.2.2 Unbalanced Workload: Failure to Balance I/O

NVIDIA GPUs use a Streaming Multiprocessor (SM) architecture, where multiple CTAs are executed concurrently on each SM. In CUDA programming, CTAs are mapped to SMs and execute in parallel. This model is highly efficient when workloads are evenly distributed among CTAs. However, the irregular nature of sparse data often leads to significant imbalances in workload distribution. For example, each row of



(a) Output balance for TC-GNN.

(b) Input balance for DTC-SpMM.

(c) Our input and output co-balance.

Figure 4: Workload partitioning strategies at the CTA level.

the sparse matrix has a different number and arrangement of non-zero entries, leading to a varying number of TCU blocks per RowWindow, as shown in Figure 4. Furthermore, results from multiple TCU blocks within the same RowWindow must be accumulated, adding complexity. This uneven distribution of TCU blocks across RowWindows poses a substantial challenge in maintaining balanced workload distribution across each SM. To address this, current methods employ various workload partitioning strategies at the CTA level, which we categorize into two main types.

Output Balancing. The approach, employed by TC-GNN as illustrated in Figure 4a, assigns each CTA the task of processing all TCU blocks corresponding to a single row within a RowWindow. This strategy ensures that the output from each CTA is confined to the results of one specific RowWindow. While this design standardizes output workloads across CTAs, it leads to a significant input imbalance due to the large variations in the number of TCU blocks per row. Since the input directly influences the volume of data loading and computation, this discrepancy causes some CTAs to complete their tasks quickly while others lag, resulting in inefficient utilization of SM resources and diminished overall performance.

Input Balancing. Conversely, DTC-SpMM adopts the input balancing approach, as demonstrated in Figure 4b. This strategy ensures that each CTA processes an equal number of TCU blocks, distributing the input workload evenly across CTAs. While this method equalizes the amount of data each CTA handles, it complicates the management of output results. Specifically, when one RowWindow contains a large number of TCU blocks, multiple CTAs might need to contribute to the same row. To address this, DTC-SpMM employs atomic operations to accumulate results from different CTAs. Although these atomic operations maintain result accuracy,

they significantly increase computational overhead. On the other hand, when a RowWindow includes only a few TCU blocks, a single CTA may need to write results across multiple rows, which exacerbates the disparities in output overhead among CTAs. Consequently, DTC-SpMM still incurs considerable overhead and does not fully resolve the imbalance issue, thereby impacting its overall efficiency.

Summary. Due to the irregular nature of sparse data in SpMM, attempting to balance the workload solely from either the input or output perspective often leads to an imbalance on the other side. Furthermore, partitioning data across RowWindows requires atomic operations to ensure the correctness of results. Given these challenges, an ideal partitioning scheme should not only achieve input-output co-balance but also maintain coarse granularity for sparse data to prevent crossing RowWindow boundaries, as illustrated in Figure 4c.

3 Voltrix-SpMM Design

Voltrix-SpMM significantly improves sparse matrix-matrix multiplication through three key innovations. First, it utilizes a bitwise-level format to compress sparse matrices, which enables coalesced memory accesses and bank conflict-free decoding. Second, a warp-level asynchronous pipelining model effectively decouples data loading from computation, leading to greater overlap and improved throughput. Third, its persistent and balanced kernel design ensures co-balanced input and output workloads, ultimately boosting overall efficiency.

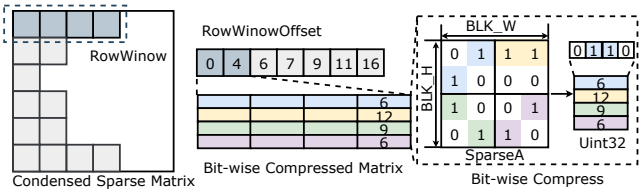
3.1 Warp-level Asynchronous Pipelining

In this subsection, we first introduce the BMat data format for accelerating sparse matrix loading. Then, we present the warp-specialized producer-consumer model that enables multi-tiered data access and asynchronous computation, effectively hiding dense data loading overhead.

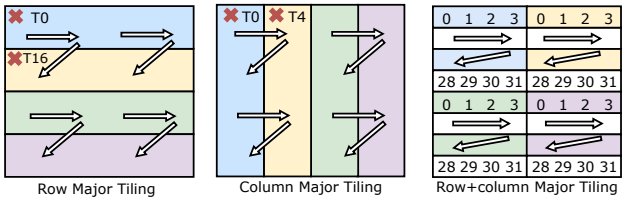
3.1.1 Bit-wise Compressed Data Format

In the design of Tensor Core-based SpMM kernels, SparseA used in MMA calculations is typically represented as a 16x8 dense matrix. However, to accommodate the high proportion of zero elements, which can exceed 90% as reported in DTC-SpMM [10], SparseA is stored in a sparse format during non-computational phases. For instance, TC-GNN uses the CSR matrix format, while DTC-SpMM employs the ME-TCF format. Directly storing SparseA in a dense format would place undue pressure on memory resources and incur substantial data copy overhead during kernel execution.

Therefore, the processing of SparseA involves two steps when transitioning from global memory to fragment: 1. loading SparseA in its sparse format, and 2. converting SparseA from a sparse to a dense format. In the data loading stage, the goal is to minimize both the data volume and the number of



(a) Bit-wise compressed matrix.



(b) Various tiling methods. Different colors represent 128-byte contiguous segments in shared memory, corresponding to one group of 32 banks. The arrows indicate the layout distribution of registers held by threads in a warp during MMA execution.

Figure 5: Loading and decoding of BMat.

loading instructions. In the conversion stage, the goal is to avoid bank conflicts during shared memory access.

While TC-GNN and DTC-SpMM have focused primarily on optimizing one of these aspects, Voltrix-SpMM aims to comprehensively address all these factors. We propose a novel data format that is both vectorization-friendly and conflict-free, termed the bit-wise compressed matrix, or BMat.

How to load? As illustrated in Figure 5a, we use `RowWindowOffset` to record the offset of each RowWindow in the condensed sparse matrix, with each offset corresponding to a SparseA. Each 0 and 1 element of SparseA is represented by just 1 bit, allowing us to compress a 16x8 matrix into a 128-bit BMat. Specifically, a BMat is stored using four Uint32 values, enabling us to access the entire BMat with a single vectorized load instruction `LDGSTS.128`.

How to decode? We address the challenge of tiling SparseA to compress it into four Uint32 values, as different tiling strategies significantly impact conversion efficiency. As illustrated in Figure 5b, one could employ row-major tiling—as adopted by the BitTCF format of ACC-SpMM [51]—resulting in four 4×8 sub-blocks, or column-major tiling, which splits it into four 16x4 sub-blocks. However, we adopt a hybrid row + column-major tiling approach, segmenting SparseA into four 8x4 sub-blocks. This method offers two key benefits:

1. Efficient decoding by each thread in a Warp using thread ID-based shifting, where each Uint32 facilitates streamlined data access. 2. The alignment of each Uint32's BMat decoding with its corresponding submatrix in the MMA computation prevents bank conflicts, enhancing computational efficiency. This contrasts with other tiling methods that often lead to cross-bank accesses, marked by red crosses in Figure 5b.

By optimizing both the loading and conversion phases, our

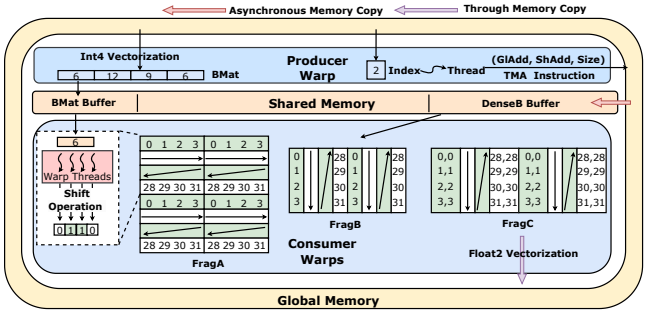


Figure 6: The warp-specialized producer-consumer model.

BMat approach enables more efficient handling of SparseA. Moreover, the decoded content is directly loaded into registers for immediate computation. In contrast to previous methods that required a 16x8-sized Int32 shared memory space, our strategy only uses four Uint32 values to store the BMat, reducing the demand on shared memory resources.

Non-binary cases. To support non-binary SparseA with floating-point values, our BMat format only requires an additional value vector to store the non-zero floating-point values. Within a warp, the i-th thread (thread T_i) constructs a 32-bit position mask M_i. The first $i+1$ bits of this mask are set to 1, while the remaining 31-i bits are set to 0. This mask is then bitwise-and with the BMat to obtain P_i. Subsequently, each thread calculates the bitwise-sum of its P_i using the `__popc()` intrinsic. This bitwise-sum determines the offset of its corresponding non-zero element within the value vector.

For instance, consider an 8x4 sparse matrix SparseA containing floating-point values at indices 3, 7, and 20. The BMat representation for this matrix is B = `0x00100088`. For thread $i = 7$, the binary mask is M_7 = `0x0000000f`. The bitwise-and operation yields P_7 = `0x00100088` & `0x0000000f` = `0x00000008`. The bitwise-sum of P_7, calculated as bitwise-sum(0x00000008), is 1, indicating that the non-zero element associated with this thread is located at the second position in the value vector (using zero-based indexing).

Unlike DTC-SpMM, which requires an explicit offset to track non-zero counts, BMat encodes this information directly. By performing a bitwise-sum on BMat, we can obtain the non-zero count without extra metadata. In our method, we only need to allocate a register within the kernel to accumulate the total count of non-zero elements. This accumulated count directly provides the starting offset for accessing the value vector in the current iteration. This design contributes to more efficient compression and memory access when using BMat.

3.1.2 Warp-Specialized Producer-Consumer Model

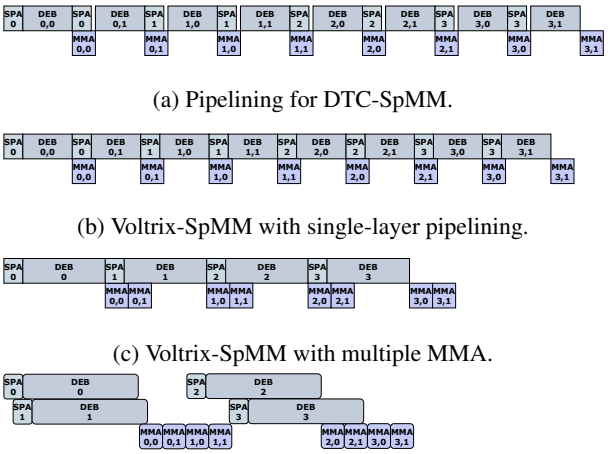
To mitigate the significant data access overhead, particularly the stalls caused by loading SparseA and DenseB, we implement an inter-warp pipelining strategy. In our kernel design, specific warps within a CTA are assigned to load data from global memory into shared memory, while other warps concurrently fetch preloaded data from shared memory to perform Tensor Core computations. This warp-specialized task allocation establishes a producer-consumer model within each CTA, where shared memory serves as a shared buffer to facilitate the overlap of computation and data access.

How to fetch data? As depicted in Figure 6, the Producer is tasked with loading data and issuing instructions, managing the transfer of SparseA and DenseB from global memory to shared memory. For SparseA, as detailed in Section 3.1.1, we employ vectorization techniques to copy the BMat to shared memory using a single INT4 instruction. For DenseB, each SparseA corresponds to a number of DenseB elements equal to BLK_W × D, where D represents the dimension of the dense data. The Producer utilizes its index to locate these elements in global memory and subsequently transfers them to shared memory. This transfer leverages asynchronous TMA instructions for efficiency: 1) TMA requires only a single instruction to access dense data, reducing the overhead associated with multiple `LDGSTS` instructions; 2) TMA allows just one thread to issue the instruction, enabling all the dense data corresponding to a SparseA in a CTA to be loaded using only 8 threads. As a result, a single Warp acts as the Producer to issue the instructions, minimizing the resource wastage caused by excessive producer warps.

The Consumer retrieves data from shared memory and loads it into registers for MMA computation using Tensor Cores, configured to an MMA size of `m16n8k8`. For SparseA, each consumer performs shift operations for decoding. For DenseB, pending operations are performed to prevent bank conflicts in shared memory. After completing all SparseA in a RowWindow, each thread retains two float32 results per MMA. Consequently, we utilize FLOAT2 write-through instructions to directly transfer data from registers to global memory, effectively bypassing shared memory.

How to pipeline? In Voltrix-SpMM design, shared memory buffers are managed using a memory barrier (`MBarrier`) signaling mechanism to efficiently coordinate producer and consumer operations. All barriers are classified as either 'ready' or 'filled'. Initially, 'ready' barriers use the `SYNCS.ARRIVE` instruction to signal the producer to load data, while 'filled' barriers use the `SYNCS.TRYWAIT` instruction to block the consumer. Once the producer finishes loading, it releases the 'filled' barrier with `SYNCS.ARRIVE`, notifying the consumer to transfer the preloaded data for computation. Subsequently, the 'ready' barrier suspends the producer with `SYNCS.TRYWAIT` for the next batch. Conversely, the consumer coordinates with the producer to wait and refill the buffer.

This creates a ping-pong scheduling mechanism, where data loading and computation alternate seamlessly across buffers. As illustrated in Figures 7a and 7b, unlike DTC-SpMM which only overlaps the loading of SparseA with MMA computation, our producer-consumer model facilitates pipelining of MMA with both SparseA and DenseB, allowing



(a) Pipelining for DTC-SpMM.

(b) Voltrix-SpMM with single-layer pipelining.

(c) Voltrix-SpMM with multiple MMA.

(d) Voltrix-SpMM with multi-tiered pipelining.

Figure 7: Different pipelining strategies.

for enhanced potential to conceal data loading overhead.

3.1.3 Fine-Grained Multi-Tiered Pipelining

Although a single-layer pipeline can mitigate some data loading overhead, it often results in low overlap and suboptimal resource utilization. In contrast, our producer-consumer model enables a higher-level extension of the traditional pipeline, maximizing hardware resource usage and facilitating seamless parallelism between computation and data access. By deploying a warp to manage multiple MMAs and buffers within the pipeline, this approach significantly minimizes idle cycles and enhances overall system throughput.

Multiple MMA. Given that the data access phase incurs higher stalls than the computation phase—due to the producer's loading speed lagging significantly behind the consumer's consumption speed—a practical solution is to increase the computation workload for each consumer without altering the producer's throughput. A single CTA can manage a maximum of 512-dimensional dense data, allocated across 32 warps, with each warp handling up to 16 dimensions of dense data. Consequently, any data exceeding 512 dimensions necessitates a reloading of SparseA. As depicted in Figure 7a, a DenseB data chunk is divided into two loading phases, with the MMA computations also split accordingly. We enable each consumer to load multiple tiled DenseB data simultaneously and conduct multiple MMA operations to enhance processing efficiency, as shown in Figure 7c. This strategy not only reduces the instruction issuing overhead associated with loading but also promotes a higher degree of overlap between computational tasks.

Multiple buffer. To maximize bandwidth utilization and enable seamless pipelining, we introduce multiple buffers, as shown in Figure 7d, to accommodate concurrent MMA operations. Each buffer operates independently, allowing the producer to issue data copy instructions for multiple buffers simultaneously. Once a buffer's barrier is set to 'filled', the consumer begins to read the data from that buffer for computation. Upon completing the computation, the consumer immediately accesses data from another buffer, while the previously used buffer is reloaded with new data. This multi-tiered fine-grained pipeline, compared to traditional single-layer pipeline, significantly enhances hardware data copy bandwidth utilization. It bridges the gap between data loading and computation and facilitates more seamless overlap of these processes, ultimately improving overall system performance.

Trade off. While incorporating more MMAs and buffers enhances pipelining potential, this approach does not guarantee improved performance due to two primary factors:

1. Shared memory is a limited resource. Since it shares the same hardware storage with the L1 cache, increasing the number of buffers can reduce cache hit rates for other data, potentially impacting overall performance.

2. Effective pipelining depends on the ability to simultaneously load corresponding DenseB data for different SparseA matrices within the same RowWindow. If a RowWindow contains only a few SparseA matrices, extensive pipelining might lead to underutilization of resources.

Consequently, striking a balance between maximizing overlap and minimizing resource wastage is crucial. In Voltrix-SpMM, we observe that the optimal configuration depends solely on the dimensions of the dense matrix. By pretesting various configurations for dense data of differing dimensions, we can identify the most efficient setup for each scenario.

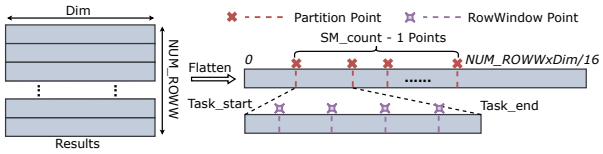
3.2 Persistent and Balanced Kernel

In this subsection, we'll first present our persistent kernel design, which eliminates atomic operations while simultaneously achieving co-balanced input and output. Following this, we'll introduce a greedy and heuristic partitioning strategy developed to identify workload-balanced partition points.

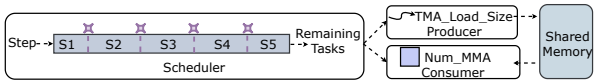
3.2.1 SM-Aligned and Atomic-Free Partitioning

Software-hardware balance. In Section 2.2.2, we discussed the challenges of achieving balance in Tensor Core-accelerated SpMM kernels. A well-balanced GPU kernel evenly distributes tasks across hardware units. However, in CUDA, software-level balance among CTAs doesn't always ensure hardware-level balance across SMs, since the number of CTAs may not match the number of SMs, causing idle SMs during later scheduling stages.

For example, TC-GNN sets CTAs based on RowWindow rows, while DTC-SpMM assigns a fixed number of SparseA elements per CTA. Both adjust CTAs dynamically based on sparse matrix structure. Given millions of SparseA elements, CTAs often far outnumber SMs, increasing overhead associated with CTAs scheduling and prologue/epilogue execution.



(a) Illustration of task partitioning.



(b) Task coordination by the scheduler.

Figure 8: SM-aligned and atomic-free partitioning.

To address this, we adopt an SM-aligned kernel design by fixing the number of CTAs to match the number of SMs. This persistent approach offers two key benefits:

1. It ensures that the software-level distribution of tasks (CTAs) corresponds directly with the hardware-level distribution (SMs). This alignment results in a balanced execution with no idle units.

2. It accommodates the high number of SparseA elements relative to the available SMs, allowing the kernel to run persistently until all tasks are completed. This continuous operation mode reduces the frequent launching and termination of kernel CTAs, which typically incur significant prologue and epilogue execution overhead.

Input-output co-balance. As detailed in Section 2.2.2, TC-GNN suffers from output imbalance, causing uneven data loading and computation across CTAs. DTC-SpMM's input balancing mitigates SparseA distribution issues but introduces atomic addition overhead and output imbalance.

To address these challenges, we propose an atomic-free partition method that balances both input and output while eliminating atomic operations. This dual-perspective approach simplifies partitioning and improves computational efficiency.

On the input side, SparseA is partitioned at the RowWindow level to keep data contiguous and complete within each partition, eliminating the need for atomic operations. On the output side, the dense matrix is partitioned by its dimensions to ensure balanced output distribution. This fine-grained, flexible partitioning fits naturally with our SM-aligned design, allowing the kernel to persist until all tasks finish. As a result, the kernel must continuously track its progress within each RowWindow and monitor remaining tasks, which introduces complexity in boundary detection.

To manage this, we integrate a scheduler within our producer-consumer model tailored to the balanced partitioning method. Task partitioning is based on result allocation, as shown in Figure 8a, by flattening the result matrix into a 1D vector. The total task count equals the number of RowWindow rows times the result matrix's dimension. We divide these tasks evenly across $\mathcal{M}$ SMs by selecting $\mathcal{M}-1$ split points, using a single warp MMA operation (i.e., 16) as the partition unit for finer granularity.

By tracking each task's start and end points, the scheduler effectively defines task and RowWindow boundaries. At runtime (as shown in Figure 8b), tasks are divided into stages aligned with RowWindow boundaries. The scheduler computes remaining tasks in the current stage and instructs the producer how much data to load via TMA. It also dynamically adjusts the number of MMA operations per warp in the consumer accordingly.

While this segmentation removes the need for atomic additions in output, it may require reloading SparseA at boundary stages with incomplete tasks (i.e., tasks that do not fully form a RowWindow). However, since RowWindows typically contain many more rows than SMs and SparseA is efficiently loaded via the BMat format, the overhead of this secondary loading is effectively negligible.

Algorithm 1: Greedy and heuristic-based partition point search algorithm

input : Total cost: $\mathcal{C}_{all}$, SM counts: $\mathcal{M}$
output : Partitioned points $\mathcal{S} = \{0, S_1 \ldots, S_{\mathcal{M}}, \mathcal{C}_{all}\}$

Calculate the average cost $\mathcal{C}_{avg} = \mathcal{C}_{all}/\mathcal{M}$.
for $S_i \in \mathcal{S}, j \leftarrow 1$ **to** $\mathcal{M}$ **do**
  Move the S_i points forward until the accumulated cost between S_{i-1} and S_i exceeds $\mathcal{C}_{avg}$.
  Calculate column position of the end point $Col = S_i \% \mathcal{D}$
  if $Col < \mathcal{D}/8$ **then**
    $S_i = S_i - Col$
  end
  else if $Col > \mathcal{D} * 7/8$ **then**
    $S_i = Si + \mathcal{D} - Col$
  end
end

3.2.2 Input-Output Co-Balance Searching

With our SM-aligned and atomic-free partitioning, the kernel balancing problem is simplified by dividing all tasks evenly among the SMs to ensure balanced workloads. To evaluate different splitting methods, we propose a cost model for the entire SpMM operation:

$$\mathcal{C}_{all} = \sum_{i=0}^{R_W} Num_SPA(i) \cdot cf_1 \cdot \mathcal{D} + R_W \cdot cf_2 \cdot \mathcal{D} + cf_3 \qquad (1)$$

The first term captures input costs across all RowWindows, including overhead from SparseA loading and computation, scaled by $\mathcal{D}$, the dense matrix dimension. The second term represents output costs from write-back operations, with R_W as the number of RowWindows. Coefficients cf_1, cf_2, and cf_3 quantify the costs of input, output, and fixed kernel overheads, respectively, and are tuned to reflect hardware characteristics.

In the experimental section 4.7, we will measure these three coefficients to validate the accuracy of our cost model. From this model, we derive the optimization objective, which is formulated as follows:

$$\mathcal{S}^* = \arg\min_{\mathcal{S}} \left(\max \left\{ \frac{C_i(\mathcal{S})}{\sum_j C_j(\mathcal{S})} \middle| i \right\} + P(\mathcal{S}) \right) \quad (2)$$

Here, $\mathcal{S}$ denotes a partitioning scheme, and $P(\mathcal{S})$ is the penalty from crossing RowWindow boundaries. To find the optimal scheme $\mathcal{S}^*$, we design a greedy, heuristic-based search algorithm, detailed in Algorithm 1.

The algorithm starts by computing the ideal per-SM cost from the total task cost (Line 1). For each SM, it incrementally extends the partition endpoint until the accumulated cost exceeds this average (Lines 3–4). To reduce boundary-crossing overhead, partition points near RowWindow boundaries are adjusted to align with them (Lines 5–10). Finally, a genetic algorithm [8, 15] refines these points for a globally optimal solution, requiring only one iteration in practice.

Utilizing the greedy and heuristic-based partition point search algorithm, we aim to balance the tasks across each SM as evenly as possible while minimizing the additional overhead incurred by crossing RowWindow boundaries.

4 Evaluation

Voltrix-SpMM is a pure CUDA-based [28] library with around 5k lines of code and no third-party dependencies. It extensively utilizes the C++ template metaprogramming paradigm to adapt to our multi-tiered pipelining design, making the code highly configurable and minimizing kernel runtime overhead. We employ inline PTX instructions to leverage specific hardware features of the Hopper GPU [30], including `MMA`, `TMA`, and `MBarrier`, to maximize efficiency.

Voltrix-SpMM is integrated into the popular machine learning framework PyTorch 2.5 [33] and can be easily invoked from both Python and C++ backends using common sparse matrix formats like CSR and COO. Voltrix-SpMM has been open-sourced at `github.com/YaqiXia/Voltrix-SpMM`, encouraging community use and contributions.

4.1 Experimental Setup

Platform. Our experiments are conducted on a Hopper H100 PCIe GPU [30], equipped with 456 Tensor Core units, 14,592 CUDA cores, and 80 GB of graphics memory.

Dataset. As detailed in Table 1, we employ 12 real-world graph datasets, categorized into two types: Type I datasets, with a smaller average row length (less than 20), and Type II datasets, with an average row length close to 500. This distinction in row length influences the data volume in each RowWindow's TCU blocks, impacting efficiency. To evaluate the adaptability of different methods, our selection includes datasets with varied average row lengths. We also incorporate the SuiteSparse dataset [7], a comprehensive collection of sparse matrix benchmarks, for extended validation.

Methodology. We benchmark SpMM against state-of-the-art CUDA Core-based and Tensor Core-based methods. The CUDA Core baselines include Sputnik [12], GE-SpMM [17], RoDe [32], and cuSPARSE [24], while the Tensor Core baselines include TC-GNN [43] and DTC-SpMM [10]. For GNN testing, comparisons are made with GNNAdvisor [42], TC-GNN, and the widely-used DGL framework [2]. Our GNN model features two GCN [20] convolutional layers, each with a hidden dimension of 256.

Table 1: Dataset Statistics

Type	Dataset	Abbr.	Vertex	Edge
I	amazon0505	A5	410,236	4,878,875
	DD	DD	334,925	1,686,092
	PPI	PI	56,944	818,716
	amazon0601	A1	403,394	3,387,388
	com-amazon	CN	334,863	1,851,744
	Yeast	YT	1,714,644	3,636,546
	YeastH	YH	3,139,988	6,487,230
	web-BerkStan	WN	685,230	7,600,595
II	FraudYelp-RSR	FR	45,914	6,805,486
	Reddit	RT	232,965	114,848,857
	ddi	DI	4267	2,140,089
	protein	PN	132,534	79,255,038

4.2 SpMM Performance

First, we evaluate the performance of Voltrix-SpMM against other SpMM kernels using real-world graph datasets and the SuiteSparse collection to provide the sparse matrices on the left-hand side. The dimensions of the dense matrices on the right-hand side are set to 256, 512, and 1024, respectively.

Graph datasets. As illustrated in Figure 9, Voltrix-SpMM achieves superior performance across nearly all graph datasets. It outperforms cuSPARSE with an average speedup of 2.7x and surpasses the state-of-the-art CUDA Core-based method, RoDe, by the 1.9x speedup. Against the Tensor Core-based method, DTC-SpMM, Voltrix-SpMM delivers an average speedup of 1.8x, demonstrating its high efficiency.

Voltrix-SpMM particularly excels as the dimension of the dense matrix increases, achieving speedups of 2.4x, 2.8x, and 3.0x over cuSPARSE for dimensions of 256, 512, and 1024, respectively. This improvement is attributed to our multiple MMA and buffer design, which can load more data per operation at higher dimensions, thereby reducing instruction overhead and making more effective use of bandwidth.

Furthermore, while DTC-SpMM and TC-GNN also utilize Tensor Cores for acceleration, their performance lags behind the CUDA Core-based RoDe by 2% and 69%, respectively. This highlights that despite the high computational throughput of Tensor Cores, significant data loading overhead can restrict their performance, leading to underutilized and 'starved' Tensor Cores. In contrast, Voltrix-SpMM overcomes these limitations with its warp-level asynchronous pipelining and balanced workload design, effectively unlocking the full

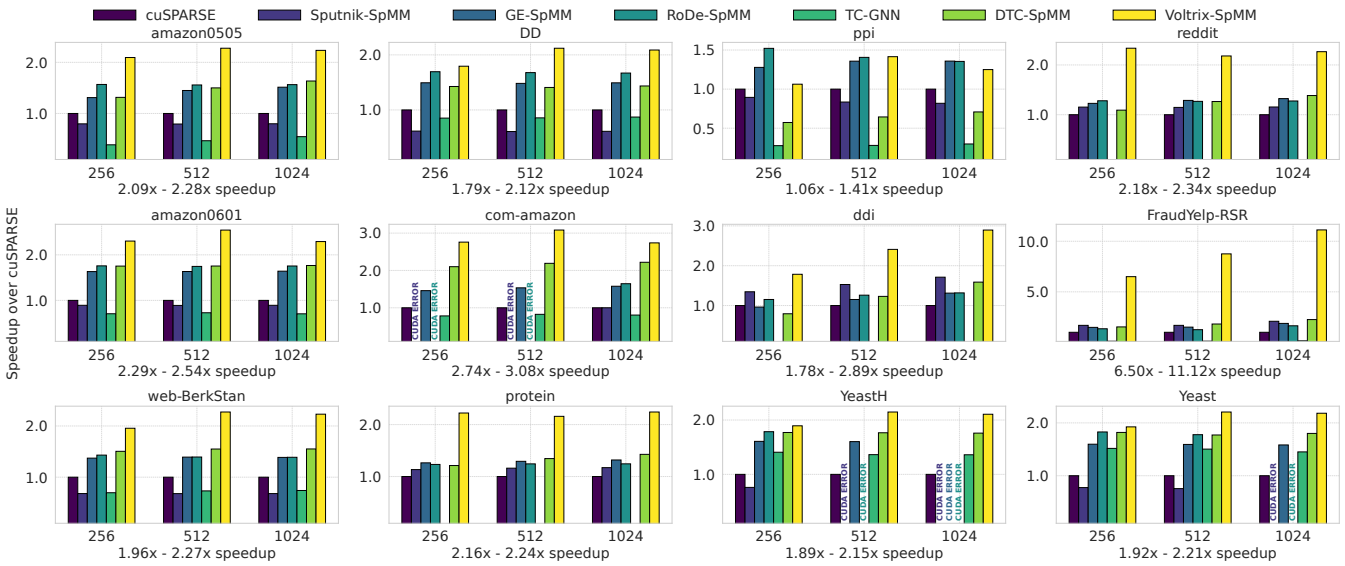


Figure 9: Performance comparison on graph datasets, with each subplot showing Voltrix-SpMM's speedup over cuSPARSE.

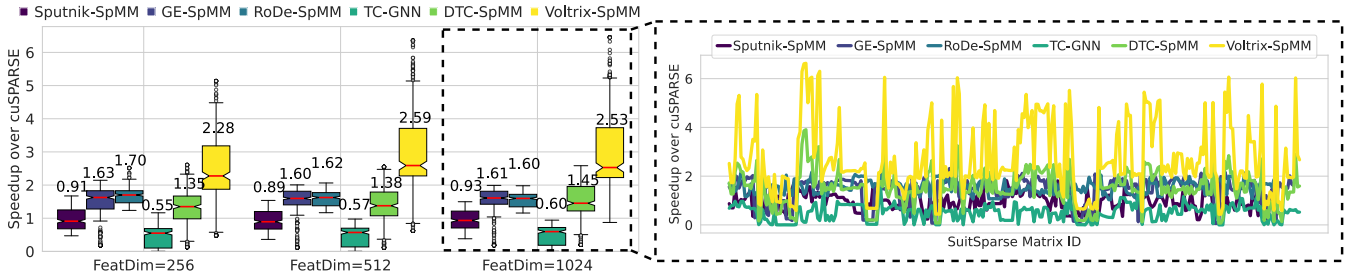


Figure 10: Speedup comparison on SuiteSparse: Box plot shows the overall distribution, while line chart shows detailed speedup.

potential of Tensor Cores for SpMM. While some CUDA Core-based methods, such as Sputnik and RoDe, encounter CUDA errors under specific conditions due to shared memory limitations, Voltrix-SpMM consistently adapts to all scenarios, thanks to its robust persistent kernel design.

SuiteSparse datasets. As shown in Figure 10, Voltrix-SpMM achieves average speedups of 2.5x compared to cuSPARSE. These results not only outperform all other methods but also demonstrate that the effectiveness of our design increases with the matrix dimensions, consistent with results observed on graph datasets. In the SuiteSparse tests, DTC-SpMM and TC-GNN still fall short of the performance achieved by the state-of-the-art CUDA Core-based method, RoDe, lagging by 11% and 70%, respectively. In contrast, Voltrix-SpMM achieves a 1.6x speedup over RoDe. This underscores that existing Tensor Core-based designs typically do not outperform CUDA Core-based methods. Our work is the first to fully leverage the capabilities of Tensor Cores, achieving speedups that surpass those achievable with traditional CUDA Core-based methods.

Mirco-level analysis. To provide a more granular understanding of the performance characteristics of various SpMM implementations, we augment our evaluation with micro-level hardware analysis. Using NVIDIA Nsight Compute, we collected key metrics such as register allocation, DRAM reads (GB), bank conflicts, and Tensor Core Unit (TCU) pipe utilization, as shown in Figure 12. These measurements are based on experiments using the Reddit dataset.

Results show Voltrix-SpMM consistently outperforms DTC-SpMM and RoDe-SpMM across multiple hardware metrics. It uses fewer registers and consumes less memory bandwidth, reflecting the efficiency of our bit-wise compressed data format, BMat.

Furthermore, Voltrix-SpMM exhibits a competitive L2 cache hit rate and significantly fewer bank conflicts, demonstrating effective memory access management, especially during the BMat decoding phase. It also demonstrates a competitive L2 hit rate and significantly fewer bank conflicts, highlighting its effective memory access management, particularly in optimizing BMat decoding phase.

Additionally, Voltrix-SpMM achieves the highest TCU pipe utilization, demonstrating superior pipeline efficiency and

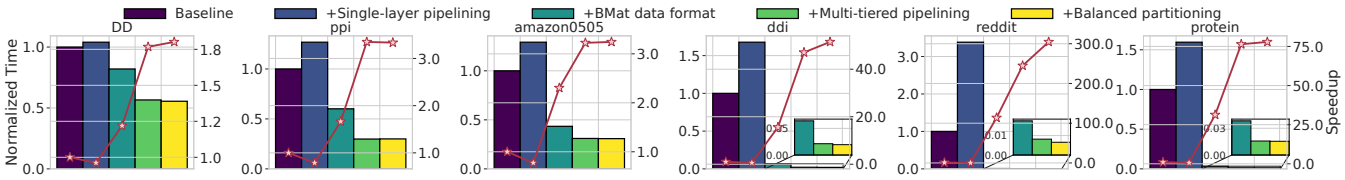


Figure 11: Performance Breakdown: The left axis shows normalized time, and the right axis shows speedup.

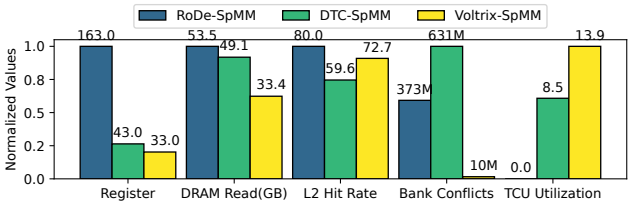


Figure 12: Micro-level performance comparison.

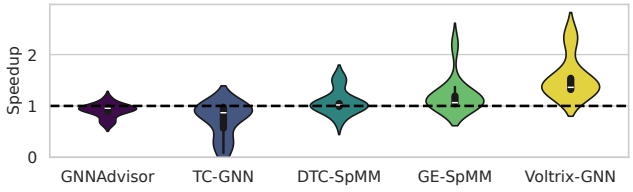


Figure 13: End-to-end GNN testing on different graph datasets: The vertical axis represents the speedup over DGL.

more effective use of GPU tensor cores. In contrast, RoDe-SpMM, limited to CUDA cores, shows zero TCU activity.

In summary, Voltrix-SpMM outperforms the other implementations in terms of memory efficiency and hardware utilization, making it the most optimized choice for SpMM tasks.

4.3 SpMM Performance Breakdown

We assess the performance contributions of Voltrix-SpMM's core components. Experiments are conducted on the graph datasets with dense matrix dimensions set to 256, using TC-GNN as the baseline. We incrementally integrate Voltrix-SpMM's components to gauge the performance improvements, which are illustrated in Figure 11.

Initially, we integrate the bulk asynchronous data transfer instruction, TMA, to establish a single layer of pipelining. However, in most datasets, this addition results in a performance degradation, with an average decrease of 32.6%. This occurs because, while the TMA instruction can transfer a large volume of data per operation, its latency is relatively high. Therefore, with the data dimension set to 256, each operation is limited to transferring only 256 float32 elements, restricting the potential for bulk data transfer. Additionally, the single-layer pipelining provides minimal overlap, further reducing the overall benefits of pipelining.

Adding our bit-wise compressed data format (BMat), we observe over a 77x average speedup, with results on Reddit reaching up to 384x. This significant improvement is due to datasets like Reddit, Protein, and ddi having larger average row lengths, which results in more SparseA blocks per RowWindow after TCU compression. In comparison, TC-GNN struggles with inefficient loading and transformation of SparseA, causing substantial delays. Voltrix-SpMM, with its bit-wise compressed data format, leverages vectorized loading and a conflict-free transformation design, dramatically reducing the processing time for SparseA.

Next, incorporating our multi-tiered pipelining yields an average 2.1x speedup. Although bulk asynchronous transfers introduce some overhead, Voltrix-SpMM counteracts this through multiple MMA operations and buffers, which amortize the instruction overhead while optimizing bandwidth utilization. Our warp-specialized producer-consumer model further facilitates fine-grained, multi-tiered pipelining, achieving higher levels of overlap and yielding significant benefits.

Lastly, integrating our persistent kernel design results in up to a 1.3x speedup. On datasets like Reddit, which feature more uneven data distributions, we observe even greater improvements, showcasing our balance search algorithm's efficacy in addressing challenges associated with uneven data distribution on input-output balance.

Overall, through the strategic design of these four components, Voltrix-SpMM achieves speedups ranging from 2.0x to 304x, underscoring the effectiveness of each module in enhancing SpMM performance.

4.4 GNN End-to-end Training

We evaluate the performance of Voltrix-SpMM in end-to-end GNN training, which we refer to as Voltrix-GNN, exploring its potential in machine learning tasks. We use GCN [20], one of the most widely used GNN models, which comprises multiple graph convolutional layers. Each layer's computation primarily involves a SpMM operation and a GEMM operation, indicating that any computational gaps in SpMM may be balanced by other operations during training.

The experimental results in Figure 13 use DGL [2], which leverages cuSPARSE library for SpMM acceleration, as the baseline. Voltrix-GNN consistently outperforms all baselines

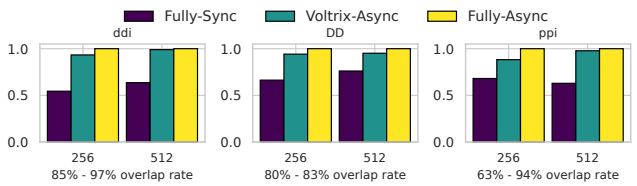


Figure 14: Overlap rate testing: The horizontal axis shows dimensions, and the vertical axis shows normalized speedup.

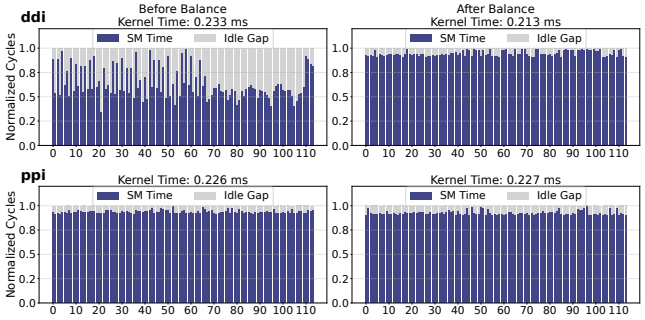


Figure 15: SM-Level active cycle distribution comparison.

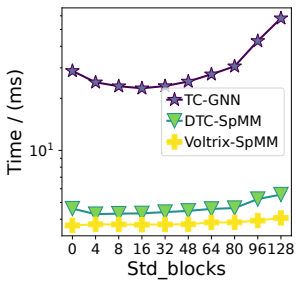


Figure 16: Sensitivity.

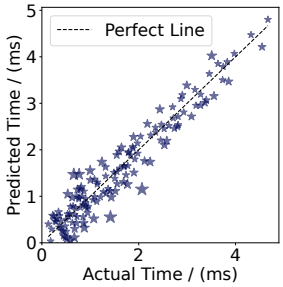


Figure 17: Time alignment.

across diverse graph datasets, with average speedups of 1.55×, 1.77×, 4.01×, 1.45×, and 1.29× over DGL, GNNAdvisor, TC-GNN, DTC-SpMM, and GE-SpMM, respectively. These findings underscore the applicability and outstanding performance of Voltrix-SpMM in GNN training tasks.

4.5 Overlap Rate of Pipelining

We investigate the overlap rate of Voltrix-SpMM's pipelining strategy. Profiling the kernel's time trace is challenging, so we adopt an approximation method to determine the kernel's overlap rate: First, we implement a configuration using a single buffer, termed `Fully-Sync`, where the consumer begins computation only after production has completed, forcing the kernel to operate serially without any pipelining. Next, we configure the kernel to run completely asynchronously by removing all `MBarrier` instructions, which we refer to as `Fully-Async`. This scenario represents the upper bound of our kernel's performance. We define the overlap rate, R_o, as:

$$R_o = \frac{T(Fully_Sync) - T(Voltrix_Async)}{T(Fully_Sync) - T(Fully_Async)} \times 100\% \quad (3)$$

Here, `Voltrix-Async` refers to our method. As depicted in Figure 14, Voltrix-SpMM achieves peak overlap rates of 85% and 97% at 256 and 512 dimensions, respectively. These results demonstrate the effectiveness of our warp-specialized asynchronous data loading pipelining. With near-seamless overlap, Voltrix-SpMM significantly reduces the data loading overhead, showcasing its high efficiency.

4.6 Workload Balance Analysis

Active SM cycles analysis. To evaluate the effectiveness of our workload balancing design, we analyze the distribution of active SM cycles using runtime data from both non-balanced and balanced kernel executions, as shown in Figure 15. The impact of our approach varies across datasets. On imbalanced workloads like ddi, our method significantly improves SM utilization and reduces kernel time by 8.6%. In contrast, for already balanced datasets like ppi, the benefit is marginal, with only a 0.4% increase in kernel time.

While the balancing introduces minor overhead, it is largely hidden by the persistent kernel design, which overlaps epilogue execution with ongoing computation. As a result,our method delivers clear gains on imbalanced workloads while preserving original performance on balanced ones.

Sensitivity analysis. We investigate the balance of different methods to irregular distributed data. By configuring the number of SparseA elements in each RowWindow, we generate sparse datasets with varying levels of uniformity. Using a Gamma distribution [25], we create datasets with a mean of 256 and variances ranging from 0 to 192, and evaluate their performance.

As shown in Fiure 16, under different variance distributions, our method outperforms both DTC-SpMM and TC-GNN in terms of speed. Moreover, as the distribution of SparseA becomes increasingly uneven,our method remains stable with only a 4% performance drop, while TC-GNN experiences a significant 47% degradation. These results demonstrate the superiority of Voltrix-SpMM in balancing I/O while avoiding the high overhead of atomic operations.

4.7 Validation of Cost Model

We evaluate the effectiveness of the cost model outlined in Equation 1, utilizing kernel execution time as the cost measure. To understand the relationship between time and the two variables, Num_SPA and R_W, we generate datasets with varying values for these parameters. Given the linear relationship postulated in our model between the cost and these variables, we employ linear regression for fitting, as demonstrated as:

$$T = \alpha_1 \cdot Num_SPA + \alpha_2 \cdot R_W + \alpha_3 \quad (4)$$

Here, α_1, α_2, and α_3 represent the coefficients of the linear regression model, correlating positively with cf_1, cf_2, and cf_3, respectively. We divide the data into a training set (80%) and a testing set (20%). Using this model, we predict values on the testing set and compare these predictions to the actual values. This comparison is visually represented in Figure 17, where the predicted values are plotted against the actual values on the x and y axes, respectively. The discrepancy between them is indicated by the size of the scatter markers, with smaller sizes denoting a closer fit to the perfect prediction line.

As depicted in Figure 17, nearly all points closely align with the ideal prediction line. Moreover, we calculate the coefficient of determination, $R^2 = 0.92$, indicating a strong linear relationship. An R^2 value exceeding 0.9 reaffirms the high accuracy of our model's fit to the data.

5 Related Work

CUDA Core-accerateled SpMM. Recently, numerous studies have focused on optimizing SpMM computation on GPUs, primarily utilizing CUDA Cores [12, 16–18, 26, 32, 34]. NVIDIA's official sparse matrix library, cuSPARSE [24], offers highly optimized GPU-accelerated routines for SpMM. Merge-SpMM [48] accelerates SpMM using the CSR format, emphasizing instruction-level parallelism. Sputnik [12] introduces a 1D tiling scheme that enhances operand reuse and employs subwarp tiling and reverse-offset memory alignment to optimize vector memory access for misaligned addresses. GE-SpMM [17] integrates a coalesced row caching method for efficient memory access and a coarse-grained warp mapping method to reduce threading overheads. GNNAvisor [42] proposes an adaptive runtime system with a novel 2D workload management strategy to improve GPU utilization. DA-SpMM [6] employs a data-aware heuristic GPU kernel, while HP-SpMM [9] uses dynamic task partitioning and hierarchical vectorized memory access. Lastly, RoDe [32] introduces a row decomposition-based method that enables efficient computation, load balancing, and fine-grained pipelining, setting a new performance benchmarks for SpMM.

Collectively, these methods optimize SpMM across various dimensions but often overlook the potential of Tensor Cores for accelerating computations. This oversight limits their ability to leverage the high throughput of modern GPUs.

Tensor Core-accerateled SpMM. With the advancement of Tensor Core computational throughput, an increasing number of studies have focused on leveraging Tensor Cores to accelerate SpMM [1, 5, 23, 41, 44]. However, most of these efforts target structured SpMM, which assumes a predefined pattern for the non-zero element distribution in sparse matrices, thereby limiting their applicability in real-world scenarios.

In contrast, TC-GNN addresses the more challenging application of unstructured SpMM on Tensor Cores. By introducing a sparse graph translation method that compresses sparse matrices into Tensor Core-sized TCU blocks, TC-GNN enables the efficient mapping of SpMM workloads to Tensor Core units. This approach has been further integrated into SparseTIR [49]. Despite its advances, however, TC-GNN's performance is compromised by inefficient data loading, which leaves the Tensor Cores underutilized. DTC-SpMM [10] attempts to enhance efficiency through several optimization methods, such as bypassing shared memory to accelerate dense data loading and implementing a pipeline arrangement to overlap sparse data loading and computation. However, the intra-warp asynchronous pipeline of DTC-SpMM struggles with the overhead caused by a large number of instructions and the costs of intra-warp synchronization.

While ACC-SpMM [51], FlashSparse [36], Groot [4], and SpInfer [11] make valuable contributions and optimizations beyond DTC-SpMM, they all operate within the same asynchronous design paradigm. To address these limitations, Voltrix-SpMM employs a multi-tiered inter-warp pipeline for enhanced computational overlap through warp specialization and utilizes bulk asynchronous data loading via TMA to efficiently handle high-dimensional data.

Libraries like cuBLAS [27], CUTLASS [31], and FlashAttention-3 [35] optimize GEMM operation using warp specialization and persistent kernel design, but the unique challenges posed by the sparse and irregular workload of SpMM make these approaches inapplicable.

6 Conclusion

In this paper, we introduced Voltrix-SpMM, a novel GPU kernel design that enhances SpMM on Tensor Cores. Voltrix-SpMM maximizes the computational potential of Tensor Cores for unstructured SpMM tasks. Its innovations include a new compression format for sparse matrices, bulk asynchronous data loading and a warp-specialized producer-consumer model, which streamline data handling and computation. Additionally, its SM-aligned, atomic-free partitioning mechanism ensures balanced workload distribution across SMs. The impact of Voltrix-SpMM is significant, achieving an average 1.8x improvement over DTC-SpMM, 1.7x over RoDe, and a 2.0x increase in end-to-end GNN training speeds, proving its effectiveness in real-world applications.

7 Acknowledgment

We sincerely thank all reviewers for their constructive comments. Special thanks to our shepherd Tim Harris for his patient guidance and valuable suggestions through multiple review rounds, which greatly improved this paper's quality.

This work was supported by the National Key Research and Development Program of China (2023YFE0205700), the National Natural Science Foundation of China (62341410), and the Science and Technology Development Fund, Macao S.A.R (FDCT) project (0078/2023/AMJ).

References

[1] Hartwig Anzt, Stanimire Tomov, and Jack J Dongarra. Accelerating the lobpcg method on gpus using a blocked sparse matrix vector product. In *SpringSim (HPS)*, pages 75–82, 2015.

[2] AWS. Deep graph library. https://github.com/dmlc/dgl, 2024.

[3] David M Blei, Andrew Y Ng, and Michael I Jordan. Latent dirichlet allocation. *Journal of machine Learning research (JMLR)*, 3(Jan):993–1022, 2003.

[4] YuAng Chen, Jiadong Xie, Siyi Teng, Wenqi Zeng, and Jeffrey Xu Yu. Groot: Graph-centric row reordering with tree for sparse matrix multiplications on tensor cores. In *Proceedings of the Twentieth European Conference on Computer Systems (EuroSys)*, pages 803–817, 2025.

[5] Zhaodong Chen, Zheng Qu, Liu Liu, Yufei Ding, and Yuan Xie. Efficient tensor core-based gpu kernels for structured sparsity under reduced precision. In *SC21: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14, 2021.

[6] Guohao Dai, Guyue Huang, Shang Yang, Zhongming Yu, Hengrui Zhang, Yufei Ding, Yuan Xie, Huazhong Yang, and Yu Wang. Heuristic adaptability to input dynamics for spmm on gpus. In *Proceedings of the ACM/IEEE Design Automation Conference (DAC)*, pages 595–600, 2022.

[7] Timothy A Davis and Yifan Hu. The university of florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)*, 38(1):1–25, 2011.

[8] Kalyanmoy Deb. Multi-objective optimisation using evolutionary algorithms: an introduction. In *Multi-objective evolutionary optimisation for product design and manufacturing*, pages 3–34. Springer, 2011.

[9] Ruibo Fan, Wei Wang, and Xiaowen Chu. Fast sparse gpu kernels for accelerated training of graph neural networks. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 501–511. IEEE, 2023.

[10] Ruibo Fan, Wei Wang, and Xiaowen Chu. Dtc-spmm: Bridging the gap in accelerating general sparse matrix multiplication with tensor cores. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 253–267, 2024.

[11] Ruibo Fan, Xiangrui Yu, Peijie Dong, Zeyu Li, Gu Gong, Qiang Wang, Wei Wang, and Xiaowen Chu. Spinfer: Leveraging low-level sparsity for efficient large language model inference on gpus. In *Proceedings of the Twentieth European Conference on Computer Systems (EuroSys)*, pages 243–260, 2025.

[12] Trevor Gale, Matei Zaharia, Cliff Young, and Erich Elsen. Sparse gpu kernels for deep learning. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14. IEEE, 2020.

[13] Will Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. In *Proceedings of the Advances in Neural Information Processing Systems (NeurIPS)*, 2017.

[14] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2016.

[15] John H Holland. Genetic algorithms. *Scientific american*, 267(1):66–73, 1992.

[16] Changwan Hong, Aravind Sukumaran-Rajam, Israt Nisa, Kunal Singh, and P Sadayappan. Adaptive sparse tiling for sparse matrix multiplication. In *Proceedings of the ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 300–314, 2019.

[17] Guyue Huang, Guohao Dai, Yu Wang, and Huazhong Yang. Ge-spmm: General-purpose sparse matrix-matrix multiplication on gpus for graph neural networks. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12. IEEE, 2020.

[18] Kezhao Huang, Jidong Zhai, Zhen Zheng, Youngmin Yi, and Xipeng Shen. Understanding and bridging the gaps in current gnn performance optimizations. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 119–132, 2021.

[19] Jacob Devlin Ming-Wei Chang Kenton and Lee Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of naacL-HLT*, volume 1, page 2, 2019.

[20] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2016.

[21] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Proceedings of the Advances in Neural Information Processing Systems (NeurIPS)*, 2012.

[22] Andrew S Lan, Andrew E Waters, Christoph Studer, and Richard G Baraniuk. Sparse factor analysis for learning and content analytics. *The Journal of Machine Learning Research (JMLR)*, 15(1):1959–2008, 2014.

[23] Shigang Li, Kazuki Osawa, and Torsten Hoefler. Efficient quantized sparse matrix operations on tensor cores. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15. IEEE, 2022.

[24] Maxim Naumov, L Chien, Philippe Vandermersch, and Ujval Kapasi. Cusparse library. In *GPU Technology Conference*, volume 12, 2010.

[25] John Ashworth Nelder and Robert WM Wedderburn. Generalized linear models. *Journal of the Royal Statistical Society Series A: Statistics in Society*, 135(3):370–384, 1972.

[26] Yuyao Niu, Zhengyang Lu, Haonan Ji, Shuhui Song, Zhou Jin, and Weifeng Liu. Tilespgemm: A tiled algorithm for parallel sparse general matrix-matrix multiplication on gpus. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 90–106, 2022.

[27] Nvidia. cublas. https://developer.nvidia.com/cublas, 2023.

[28] Nvidia. Cuda. https://docs.nvidia.com/cuda/cuda-toolkit-release-notes/index.html, 2023.

[29] Nvidia. Tensor core. https://www.nvidia.cn/data-center/tensor-cores/, 2023.

[30] Nvidia. Nvidia hopper architecture. https://www.nvidia.com/en-us/data-center/technologies/hopper-architecture/, 2024.

[31] Nvidia. Cutlass. https://github.com/NVIDIA/cutlass, 2025.

[32] Meng Pang, Xiang Fei, Peng Qu, Youhui Zhang, and Zhaolin Li. A row decomposition-based approach for sparse matrix multiplication on gpus. In *Proceedings of the ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 377–389, 2024.

[33] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: an imperative style, high-performance deep learning library. In *Proceedings of the Advances in Neural Information Processing Systems (NeurIPS)*, pages 8026–8037, 2019.

[34] Hongwu Peng, Xi Xie, Kaustubh Shivdikar, Md Amit Hasan, Jiahui Zhao, Shaoyi Huang, Omer Khan, David Kaeli, and Caiwen Ding. Maxk-gnn: Extremely fast gpu kernel design for accelerating graph neural networks training. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 683–698, 2024.

[35] Jay Shah, Ganesh Bikshandi, Ying Zhang, Vijay Thakkar, Pradeep Ramani, and Tri Dao. Flashattention-3: Fast and accurate attention with asynchrony and low-precision. *arXiv preprint arXiv:2407.08608*, 2024.

[36] Jinliang Shi, Shigang Li, Youxuan Xu, Rongtian Fu, Xueying Wang, and Tong Wu. Flashsparse: Minimizing computation redundancy for fast sparse matrix multiplications on tensor cores. In *Proceedings of the 30th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 312–325, 2025.

[37] Jie Sun, Li Su, Zuocheng Shi, Wenting Shen, Zeke Wang, Lei Wang, Jie Zhang, Yong Li, Wenyuan Yu, Jingren Zhou, et al. Legion: Automatically pushing the envelope of multi-gpu system for billion-scale gnn training. In *2023 USENIX Annual Technical Conference (USENIX ATC)*, pages 165–179, 2023.

[38] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in Neural Information Processing Systems (NeurIPS)*, 30, 2017.

[39] Hulin Wang, Yaqi Xia, Donglin Yang, Xiaobo Zhou, and Dazhao Cheng. Harnessing inter-gpu shared memory for seamless moe communication-computation fusion. In *Proceedings of the 30th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 170–182, 2025.

[40] Weihu Wang, Yaqi Xia, Donglin Yang, Xiaobo Zhou, and Dazhao Cheng. Accelerating distributed dlrm training with optimized tt decomposition and micro-batching. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–15. IEEE, 2024.

[41] Yuke Wang, Boyuan Feng, and Yufei Ding. Qgtc: accelerating quantized graph neural networks via gpu tensor core. In *Proceedings of the ACM SIGPLAN Symposium*

on Principles and Practice of Parallel Programming (PPoPP), pages 107–119, 2022.

[42] Yuke Wang, Boyuan Feng, Gushu Li, Shuangchen Li, Lei Deng, Yuan Xie, and Yufei Ding. Gnnadvisor: An adaptive and efficient runtime system for gnn acceleration on gpus. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 515–531, 2021.

[43] Yuke Wang, Boyuan Feng, Zheng Wang, Guyue Huang, and Yufei Ding. Tc-gnn: Bridging sparse gnn computation and dense tensor cores on gpus. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC)*, pages 149–164, 2023.

[44] Haojun Xia, Zhen Zheng, Yuchao Li, Donglin Zhuang, Zhongzhu Zhou, Xiafei Qiu, Yong Li, Wei Lin, and Shuaiwen Leon Song. Flash-llm: Enabling cost-effective and highly-efficient large generative model inference with unstructured sparsity. *Proceedings of the VLDB Endowment*, 17(2):211–224, 2023.

[45] Yaqi Xia, Donglin Yang, Xiaobo Zhou, and Dazhao Cheng. Scaling new heights: Transformative cross-gpu sampling for training billion-edge graphs. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–15. IEEE, 2024.

[46] Yaqi Xia, Zheng Zhang, Hulin Wang, Donglin Yang, Xiaobo Zhou, and Dazhao Cheng. Redundancy-free high-performance dynamic gnn training with hierarchical pipeline parallelism. In *Proceedings of the 32nd International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*, pages 17–30, 2023.

[47] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks? *arXiv preprint arXiv:1810.00826*, 2018.

[48] Carl Yang, Aydın Buluç, and John D Owens. Design principles for sparse matrix multiplication on the gpu. In *Proceedings of the European Conference on Parallel Processing (EuroPar*, pages 672–687. Springer, 2018.

[49] Zihao Ye, Ruihang Lai, Junru Shao, Tianqi Chen, and Luis Ceze. Sparsetir: Composable abstractions for sparse compilation in deep learning. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 660–678, 2023.

[50] Zheng Zhang, Donglin Yang, Xiaobo Zhou, and Dazhao Cheng. Mcfuser: High-performance and rapid fusion of memory-bound compute-intensive operators. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–15. IEEE, 2024.

[51] Haisha Zhao, San Li, Jiaheng Wang, Chunbao Zhou, Jue Wang, Zhikuang Xin, Shunde Li, Zhiqiang Liang, Zhijie Pan, Fang Liu, et al. Acc-spmm: Accelerating general-purpose sparse matrix-matrix multiplication with gpu tensor cores. In *Proceedings of the 30th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, pages 326–338, 2025.

[52] Kai Zhong, Zhenhua Zhu, Guohao Dai, Hongyi Wang, Xinhao Yang, Haoyu Zhang, Jin Si, Qiuli Mao, Shulin Zeng, Ke Hong, et al. Feasta: A flexible and efficient accelerator for sparse tensor algebra in machine learning. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 349–366, 2024.

NetKeeper: Enhancing Network Resilience with Autonomous Network Configuration Update on Traffic Patterns and Anomalies

Zhaoyang Wan[1], Rongxin Han[1], Haifeng Sun[1,*], Qi Qi[1], Zirui Zhuang[1],
Bo He[1], Liang Zhang[2], Jianxin Liao[1], Jingyu Wang[1,*]
[1]*State Key Laboratory of Networking and Switching Technology,*
Beijing University of Posts and Telecommunications
[2]*Huawei Technologies Co., Ltd*

Abstract

Incremental policies and anomaly logs require operators to update network configuration during network operations. However, existing configuration methods lack the capability for intent understanding, traffic analysis optimization, and network dynamic adaptability, complicating overall configuration management.

We propose *NetKeeper*, an autonomous network configuration update framework. *NetKeeper* updates network configurations based on multimodal network intent comprising natural language input and anomaly logs, enabling adaptability to network dynamics and enhancing resilience through analyzing traffic patterns and anomalies. We implement northbound and southbound interfaces to translate network intents from operators and network management platforms respectively, bridging the gap between network intents and network behaviors. A multi-agent reinforcement learning model is designed for network configuration updates based on traffic patterns in dynamic networks. This model divides agents based on configuration parameter types, achieving both network resilience optimization and forwarding policy satisfaction.

Experiments in dynamic network show that *NetKeeper* updates network configurations with 99.6% average policy consistency, improves network performance by 5.3%, and reduces traffic shift by 8.7% on average.

1 Introduction

Network operators today face a wide range of management challenges from numerous network events [37], including service changes, topology changes, facility anomalies, and network congestion [15]. They must analyze network and events to formulate and implement low-level policies across devices throughout the network [17], thereby updating configurations to maintain forwarding policies and ensure service assurance. However, as enterprise networks grow increasingly complex, this manual process becomes more time-consuming and error-prone [17]. Moreover, a study reveals that over half of networks experience at least ten change events monthly [1]. The combination of network complexity and frequent changes necessitates substantial human effort from operators to learn and update networks in response to various events.

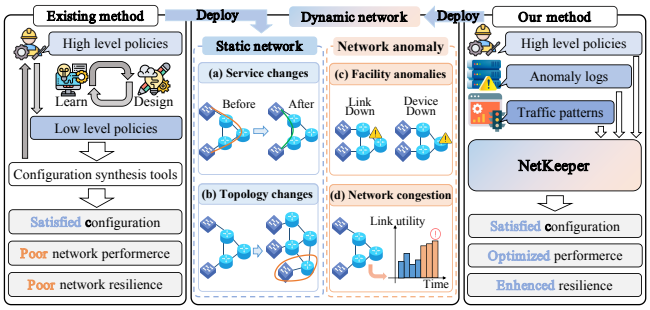


Figure 1: Illustration of intent-driven network configuration update. Comparison of existing methods with ours includes applicable scenarios, intent forms, processes, and functions.

Recently, several configuration update tools [1, 5, 9, 10, 15, 31, 46] are proposed to assist network operators. These tools input network intents in various forms and produce network configuration as output. However, when facing configuration update in dynamic networks, significant challenges still remain in aligning network intents and automating operational processes.

Firstly, different tools [1, 9, 15, 31] design various network modeling approaches. As shown in Fig. 1, before using these tools, operators need to learn specific modeling languages for networks and policies to express intents [43], which not only demands a high level of expertise but also makes policy writing and maintenance challenging. Moreover, anomaly logs related to network failures (e.g., device or link malfunctions) and network congestion from network management platforms [18] are crucial for network update, serving as update guidance and requiring operators to maintain assure services and performance [39]. Existing tools [17, 46] cannot automatically update network configurations based on these logs, necessitating manual analysis and solution development by

*Corresponding author: Jingyu Wang (wangjingyu@bupt.edu.cn) and Haifeng Sun (hfsun@bupt.edu.cn).

operators.

Secondly, these tools are unable to update configuration that effectively perform load balancing or reduce traffic shifts based on analyzing traffic patterns [34], resulting in network updates with poor resilience. This situation requires operators to manually simulate and address complex network performance issues, which is extremely difficult for them. Furthermore, as networks constantly evolve in architecture, services, and traffic patterns [5], existing work [1, 9, 15, 24] that updates configurations based on static networks fails to adapt to these changes, limiting their ability to maintain efficient service in dynamic network environments.

The concept of autonomous networks (AN) [12] represents a paradigm in network management, envisioning networks that can interpret high-level intents, autonomously explore solutions, and continuously optimize their performance. Based on this vision, we propose a framework that takes multimodal network intent as input and updates network configurations based on analysis of traffic patterns and anomalies. However, three challenges remain to be addressed.

How to implement intent translation interfaces for both operators and networks? Network operators utilize natural language for configuration intent description, offering high readability but vague information. Conversely, network anomaly logs, generated as formatted text, lack readability but contain specific information. While works in [1, 9, 15] employ domain-specific languages (DSL) [27] for intent input and works in [17, 46] accept natural language intent for network updates, none of these can effectively handle multimodal intents. Bridging gap between diverse intent representations and effectively mapping them to low-level policies leads to more comprehensive configuration update.

How to achieve traffic-based network configuration update while also enhancing network resilience? Configuration update solely focusing on forwarding policies often fall short in ensuring network performance. Traffic patterns, reflecting service demands, significantly impact the effectiveness of network configurations. Ignoring traffic information during updates can lead to suboptimal settings, causing unexpected performance issues even when policies are met. Existing approaches such as NetRen [15] and AED [1] have limitations in traffic awareness or overlooking configuration impacts on traffic distribution. To achieve effective and resilient network configurations, it's crucial to incorporate traffic-aware mechanisms into the update process, optimizing for both policy compliance and real-world network conditions.

How to realize configuration update for dynamic networks? As networks are dynamic with changing policies and traffic patterns [6], configuration update based on static network snapshots without considering actual network conditions cannot be a one-time, permanent solution, especially when facing network dynamics. Existing approaches to network configuration update, such as those using supervised learning [7, 15] or logical expression construction [9, 10, 24], require manual intervention before update. Whether modifying input network templates or crafting logical expressions, these methods limit automation and adaptability to dynamic networks, as they cannot actively gather network information or respond to changes without manual intervention.

To address aforementioned challenges, we propose *NetKeeper*, an autonomous network configuration update framework. It provides interfaces for both operators and network management systems, updating configurations based on multimodal intents, and enhancing network resilience through analyzing traffic patterns and anomalies. In summary, our contributions are as follows:

- A bidirectional interface of intent translation is proposed, featuring northbound interfaces for operators to manage network policies and southbound interfaces for network management platforms to handle anomalies.
- A goal-oriented traffic-based configuration update model is proposed that can autonomous updates configuration in dynamic network, while enhancing network resilience through analyzing traffic patterns and anomalies and satisfying forwarding policies.
- An autonomous network configuration update framework is proposed, which enables service assurance and self-managing under anomalies, or interacts with operators to update network configuration.

The rest of the article is organized as follows: §2 briefly introduces the related work. §3 outlines the systematic workflow of *NetKeeper*. §4 presents the design of intent translation. §5 details the network configuration update model. §6 discusses and analyses the result of evaluation. §7 discusses the limitations. §8 concludes the paper.

2 Related Work

2.1 Intent-based Networking

Existing works express network intents using various approaches, including domain-specific languages [3, 7, 10, 15, 30], logical expressions [1, 10], and high-level languages for policy expression, as seen in Propane [4], Genesis [40], and Frenetic [36]. For non-natural language intents, operators need to write policies in specific formats before performing updating network. Although Lumi [17] and CONFPILOT [46] can handle intents expressed in natural language, they can not handle anomalies without operators' effort. These tools fail to bridge the gap between non-natural and natural language intents, map intents of different formats to low-level policies, and provide no interfaces to handle various anomalies generated by management platforms, which are crucial for network self-managing.

2.2 Configuration Update

Configuration update aims at synthesizing a set of values that meet policies based on parameters and options involved in

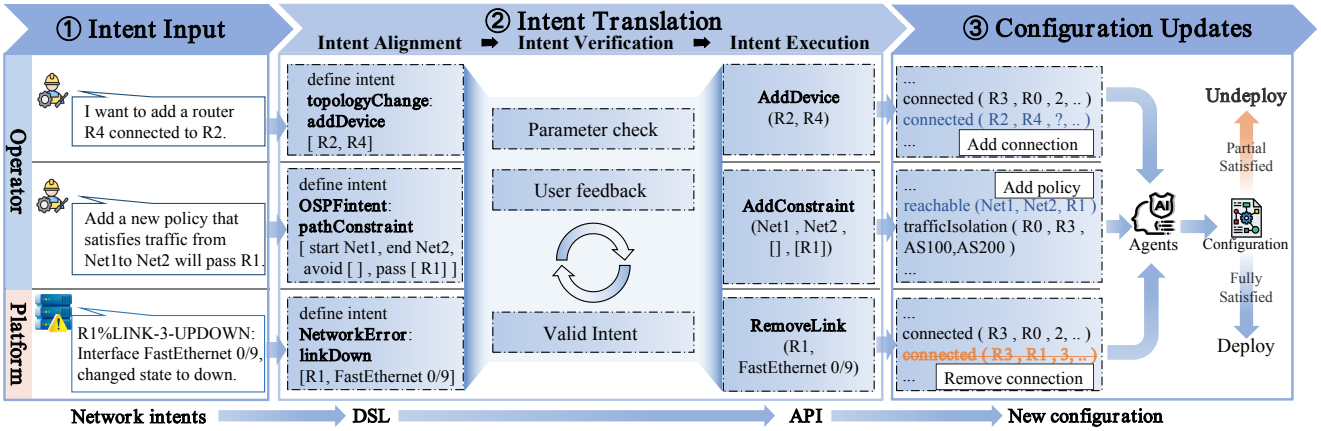
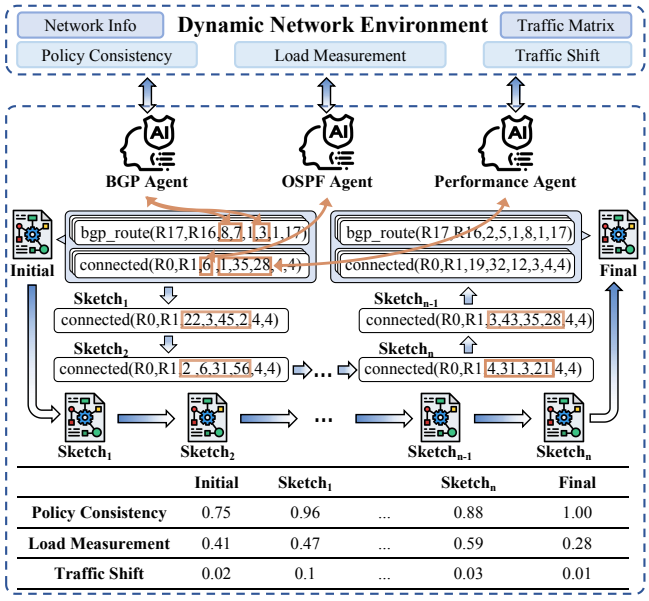


Figure 2: Overview of autonomous network configuration update framework *NetKeeper*. ① Get intent from operator or platform. ② The LLM translates different types of intent into DSL, validates them through a parameter check and feedback, and calls the appropriate APIs based on the DSL. ③ Modify network sketch and update network configuration through agents if configuration is fully satisfied. Three examples are illustrated in the figure (from left to right respectively).



	Initial	Sketch$_1$	...	Sketch$_n$	Final
Policy Consistency	0.75	0.96	...	0.88	1.00
Load Measurement	0.41	0.47	...	0.59	0.28
Traffic Shift	0.02	0.1	...	0.03	0.01

Figure 3: Illustration of configuration update progress.

protocols[35]. Based on different implementation methods, existing approaches can be divided into two categories, including supervised learning [7, 15, 24, 46] and satisfiability modulo theories (SMT) [1, 9, 10]. Original network configuration reflects how traffic forwards, while traffic format can guide optimization. However, both supervised learning and SMT approaches disregard existing configuration parameters and lack the ability to perceive network traffic. More importantly, these methods produce static configurations that cannot adapt to long-term dynamic network, as they lack the ability to interact with and respond to changing dynamic network.

3 NetKeeper in a Nutshell

We begin with a runtime workflow shown in Fig. 2 and demonstrate how *NetKeeper* processes multimodal intents to update network configurations and optimize performance.

First, for the Intent Translation module (§4), *NetKeeper* integrates northbound and southbound interfaces to process natural language intents from operators and anomaly logs from the operation and maintenance platform. These intents are translated into DSL, ensuring their alignment and unification. User feedback and parameter validation further refine the intents through the intent verification, enhancing their accuracy and ensuring the reliability of the LLM used for translation.

The LLM then utilizes the DSL to trigger appropriate APIs, enabling a variety of network updates such as basic configuration changes, topology adjustments, policy modifications, and congestion mitigation. Predefined APIs operate on network configuration files or network sketches [15], where direct edits—e.g., changing IP addresses, link weights, or OSPF metrics—are tailored for predefined operator plans, facilitating quick responses to specific intents.

For global configuration changes, such as network-wide topology revisions or policy optimization, the system first updates the network sketch before applying changes to global configurations. These complex updates employ a multi-agent deep reinforcement learning (MADRL) model [45] described in §5, which optimizes configuration changes based on categorized parameters.

As illustrated in Fig. 3, we design three specialized agents, each responsible for specific parameter categories. These agents interact with the network environment, iteratively generating new configurations. The system evaluates key metrics—policy consistency, load measurement, and traffic shifts—based on real-time traffic matrices and network states. Feedback in the form of rewards guides agents to produce configurations with improved alignment to policies, enhance load measurement, and minimal traffic shift. This iterative process ensures configurations are optimized for both operational goals and network polices.

Table 1: Examples of network intent.

#	Natural language intent
1	Configure IP address 127.39.236.169 and subnet mask 255.255.255.248 for the Ethernet34 port of R55.
2	Set the OSPF Hello packet interval on R5 to 20 ms.
3	Configure access control for R45's own traffic, performing a deny operation on traffic from R19.
4	Remove R6 from the global network.
5	Adjust port weights to form a route from R2 to R8, avoiding R6 and R1 while including the routing nodes R5 and R3.
#	Network anomaly log
6	R1%LINK-3-UPDOWN: Interface Fe 0/9 change state to down.
7	HOST_X%Node down: ping timeout (0 / 5).
8	R1%LINK-3-CONGESTION: Interface Gi1/0/1 output queue full, packets dropped: 1532.

Table 2: Part of DSL-supported operations.

Name	Type	Parameters
assignIp	Basic Config	port/subnet/mask
staticRouteBasic	Static Route	subnet/mask/next_hop
generateAcl	ACL	ip/subnet/action
assignAcl	ACL	router/interface/aclItem
establishNeighbor	OSPF/BGP	ip/mask/area
setPortWeight	OSPF	router/interface/weight
pathConstraint	OSPF/BGP	start/end/avoid/pass
assignAsNumber	BGP	asNumber
bgpPolicy	BGP	policies

4 Intent Translation

Intent translation is the critical process of transforming high-level network intents into actionable configurations. This involves three key stages: semantic alignment, validation through DSL abstraction and execution with APIs. This workflow ensures that diverse inputs converge into a unified representation, are rigorously validated for correctness and completeness, and are reliably converted into operational network updates.

4.1 Intent Alignment

In enterprise network environments, understanding and managing network intents is essential for maintaining operational efficiency. These intents can originate from two primary sources : user-driven natural language inputs and system-generated anomaly logs. Natural language intents reflect high-level objectives, such as "`optimize traffic for critical applications`," while anomaly logs highlight internal system issues, such as "`link congestion on Router B, Interface C.`" Both forms of intent are critical yet inherently different: natural language provides proactive, surface-level goals, whereas logs deliver reactive, diagnostic insights. Addressing complex and dynamic network scenarios requires a consistent approach that integrates these two types of intent into a unified model.

To achieve this, we propose a structured and formalized intent description as a unified method to represent the semantic domain of network intents. This model serves several purposes: it aligns diverse intent representations, bridges the gap between high-level goals and low-level configurations, and provides clear semantic boundaries for translating intents. Furthermore, it establishes a theoretical foundation for the subsequent design of DSL in §4.2, enabling precise and verifiable intent-to-configuration translation.

We formalize this intent description as $IT = \{ND, NP, PO, NR, AT, FP\}$. The components of IT are defined as follows: ND represents network devices such as routers and switches. NP denotes network protocols like Open Shortest Path First Protocol (OSPF) [29] and Border Gateway Protocol (BGP) [32]. PO stands for protocol options, for instance, the OSPF Hello interval. NR refers to network rules, including Access Control Lists (ACL) [33]. AT signifies attributes such as interface IP addresses. Finally, FP represents forwarding policies, which include path selection policies. It's important to note that a specific intent may not necessarily include all elements of this description. For example, intents 1 and 6 in Table 1 can be represented as $IT = \{ND : R55, AT : \{interface : Ethernet34, ip_address : 127.39.236.169, subnet_mask : 255.255.255.248\}\}$ and $IT = \{ND : R1, AT : \{interface : FastEthernet0/9, state : down\}\}$ respectively.

Table 1 presents a series of network intents, encompassing both natural language intents and network anomaly logs, all of which fall within the scope of IT. For operators with varying levels of network expertise, from novices to experts, they can input intents ranging from high-level goals to precise configuration parameters, which include: (1) Port IP allocation or modification. (2) Specific parameter specifications for routing protocols including OSPF and BGP. (3) Addition, removal, or modification of ACL entries. (4) Changes to physical topology including device or link alterations. (5) Specific routing polices. Meanwhile, we process various types of network anomaly logs, including device malfunction and network congestion logs, collected from network management systems [18] to guarantee business continuity and maintain network performance, including: (6) Link failures. (7) Device node failures. (8) Link congestion or underutilization.

4.2 Intent Verification

While IT provides a structured and unified model to describe network intents, it focuses on static representations and lacks the mechanisms for direct validation or feedback. We design a DSL as an abstraction layer between network intents and low-level network configurations, building on the semantic framework provided by IT.

The DSL adopts a syntax inspired by interpretive languages, enabling it to support both the definition and execution of network intents. As shown in Fig. 2②, each DSL intent starts with a context declaration, followed by a sequence of operations and parameters. The DSL operationalizes IT by transform-

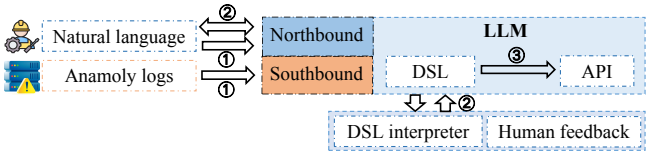


Figure 4: The process of intent translation.

ing elements like *ND*, *NR*, and *FP* into actionable operations. These operations cover a wide range of network tasks, including device configuration, ACL rules, OSPF and BGP settings, topology adjustments, link load optimization, and forwarding policies, part of them are shown in Table 2.

LLM excels at understanding the semantic nuances of natural language inputs, making them a powerful tool for converting high-level operator intentions into structured DSL representations while preserving the intent's original meaning. To facilitate the mapping of unstructured intents into operations in DSL, we utilize LLM fine-tuned with P-tuning [26], a parameter-efficient approach for adapting pre-trained models to downstream tasks. However, LLM suffer from the hallucination problem and cannot guarantee the accuracy of the translation. To further validate and ensure the correctness of generated DSL intents, we introduce a dedicated DSL interpreter. The interpreter checks the syntax correctness, verifies the completeness of parameters, and identifies any logical inconsistencies in the DSL representation.

In addition, LLM plays a key role in incorporating feedback during the translation process. Errors from the compiler are described in natural language and fed back to the user through the LLM, so that the user can further provide feedback to correct missing or incorrect parameters. For example, when a network intent involves an invalid device, such as: `"I want to add a router A1 connected to R2"`, the DSL interpreter would provide feedback: `"A1 is not a valid device in the network. Please provide a device from [R1, R2, ...]."` The system then passes this feedback to the user via LLM, allowing for adjustments to the intent. This bidirectional feedback loop ensures an accurate and iterative translation process between high-level intent and DSL. By combining the reasoning flexibility of the LLM with the robustness of DSL's static validation models, this hybrid approach guarantees both the expressiveness and reliability required for intent translation.

4.3 Intent Execution

While LLM excels in nature language processing tasks, they lack both the specialized knowledge [43] and robust reasoning capabilities required for network updating. Network configuration updates require protocol expertise, device-specific knowledge, and system-wide impact analysis. By utilizing pre-defined APIs, we compensate for these limitations, allowing us to combine the flexibility of LLM in understanding intent with the reliability of well-tested, domain-specific APIs.

We deploy the fine-tuned LLM within the LangChain [41] framework, with a series of APIs established for the LLM to call, enabling it to execute configuration updates. The APIs accept parameters in DSL as input to perform configuration update. Internally, they update configuration either by directly modifying device configurations or by utilizing the configuration update model described in §5.

Table 3: Parameters of OSPF, link attributes and BGP.

Description	Parameter	Description	Parameter
OSPF weight	α	Local preference (LP)	γ_1
Bandwidth	β_1	AS path length (AS)	γ_2
Capacity	β_2	Multi-exit discriminator (MED)	γ_3
Queue length	β_3		

The Fig. 4 illustrates a complete intent translation process: ① The LLM receives the operator's natural language intent through northbound interface or the platform's anomaly logs via southbound interface. ② The LLM translates this input into DSL and refines it through feedback from humans and logs generated by the DSL interpreter. ③ Once the DSL is validated, the LLM invokes the corresponding API to update the network configuration. Otherwise, if the DSL fails validation, no action is performed.

5 Configuration Update

As the final step of *NetKeeper*, the internal operation of these APIs determines how effectively the network adapts to dynamic conditions. To enhance adaptability, we leverage Deep Reinforcement Learning (DRL) [2] within the API execution layer, enabling configuration updates that dynamically optimize network performance while maintaining alignment with network intents.

5.1 Task Description and Goals

Traditional network configuration updates are mainly based on the adjustment of static network protocol parameters to meet forwarding policies, which leads to limited consideration of configuration parameters, a lack of adaptation to dynamic network traffic patterns, and low flexibility. Meanwhile, the configuration update process often ignores the original network state, including policy satisfaction, network performance, etc., fails to effectively utilize existing configuration information for synthesizing and optimizing new configurations, and does not fully consider changes in forwarding behaviors and traffic patterns before and after the update, which can easily result in network instability after the update.

To address these limitations, we need to consider three new factors: (1) link attributes, such as bandwidth, which impact network performance. (2) traffic patterns, including traffic matrix, enabling more targeted configuration optimization. (3) changes in the forwarding plane before and after updates. By leveraging these additional factors, we aim to achieve two main objectives: (1) Optimize network performance and comply with policies by adjusting parameters of BGP, OSPF and link attributes listed in Table 3. (2) Dynamically adjust net-

work configuration based on network environment, enhancing network resilience.

To address these requirements, we seek a solution that can incorporate diverse network attributes, capture dynamic traffic patterns, and adjust to evolving network conditions. DRL [2] naturally fits this need: it is able to learn from real-time environment feedback [25] and optimize decisions toward explicit objectives [28]. With DRL, we can automatically synthesize new configurations adaptive to both the current network state and traffic changes, thus improving performance and stability. Consequently, DRL is well-suited for our task.

We define a series of goals to leverage these capabilities for configuration update:

1. Policy Consistency π: Considering efficient service delivery, secure data transmission, and reliable business continuity, we adopt three kinds of policies (A, B and C can be router, autonomous system or sub-network) : (1) `forward(A, B, C)`: Traffic from A to B must be forwarded to its neighbor C. (2) `reachable(A, B, C)`: Traffic from A to B must pass by C. (3) `isolation(A, B, C, D)`: One of `forward(A, B, C)` or `forward(A, B, D)` must be true at the same time.

Policy consistency defined in Eq. 1 characterizes the degree of satisfaction of forwarding policies. Here, P represents the set of forwarding policies, where p_i denotes an individual policy in P. $Satisfy(p_i)$ is a function that returns 1 if policy p_i is satisfied, and 0 otherwise.

$$\pi = \frac{\sum_{p_i \in P} Satisfy(p_i)}{|P|}. \tag{1}$$

2. Load Measurement ρ: Changes in traffic patterns or routing decisions can disrupt load distribution, causing congestion and degraded service quality. Link utilization is a key metric to measure global load, and minimizing the maximum link utilization helps balance traffic, prevent bottlenecks, and improve overall network performance.

For calculation of ρ, we extend the network performance metrics designed in [38], including bandwidth, capacity, link utilization, queue length and lose rate. We use various network information as input, including the network topology $G = (N, M_{Arc})$, where N is the set of nodes and M_{Arc} is an adjacency matrix describing the network topology, as well as the link weight matrix M_W, traffic matrix M_T, bandwidth matrix M_B, capacity matrix M_C, queue length matrix M_Q, lose rate matrix M_{Lose}, and queue packet size q_s.

We first initialize the link loads ld and maximum link utilization U_{max}. For each traffic class with destination dst, we follow these steps:

1) Calculate the shortest distance d_u^{dst} from each node $u \in N$ to dst.
2) Identify shortest-path node pairs (u,v) to dst where $M_{Arc(u,v)}^{dst} = 1$ indicates traffic from u reaches dst via v:

$$M_{Arc}^{dst} = \{(u,v) \in M_{Arc} : d_u^{dst} - d_v^{dst} = M_{W(u,v)}\}. \tag{2}$$

3) Calculate δ_u^{dst}, the number of next-hop nodes on shortest paths from u to dst, to determine the Equal-Cost Multi-Path (ECMP) [16] options for each node:

$$\delta_u^{dst} = |v \in N : (u,v) \in M_{Arc}^{dst}|. \tag{3}$$

4) Calculate the actual traffic load $ld_{(v,w)}^{dst}$ on link (v,w) towards destination dst, where link capacity, bandwidth, and queue length impose upper limits, while also considering upstream traffic accumulation, ECMP load balancing, and packet loss:

$$ld_{(v,w)}^{dst} = \min(M_{B(v,w)}, M_{C(v,w)}, M_{Q(v,w)} \cdot q_s, \sum_{(u,v) \in M_{Arc}^{dst}} \frac{1}{\delta_u^{dst}} \cdot (d_v^{dst} + (1 - M_{Lose(u,w)}) \cdot ld_{(u,w)}^{dst})). \tag{4}$$

After processing all traffic classes, we aggregate these values to determine the total load on each link $(v,w) \in M_{Arc}$:

$$ld_{(v,w)} = \sum_{dst \in N} ld_{(v,w)}^{dst}. \tag{5}$$

Finally, we compute the utilization for each link $(v,w) \in M_{Arc}$ and update the maximum utilization:

$$M_{U(v,w)} = ld_{(v,w)} / M_{C(v,w)}, \tag{6}$$

$$U_{max} = \max(M_{U(v,w)}, U_{max}). \tag{7}$$

3. Traffic Shift τ: Changes in protocol configuration parameters can affect the forwarding plane of the entire network, causing changes in device routing tables, which bring traffic shifts. Traffic shift can lead to increased latency, load imbalance, and inconsistent performance, thereby impacting network quality [23]. We consider the degree of traffic shift introduced by the new configuration, which we define as Eq. 8.

$$\tau = \frac{1}{|ND| \cdot |PF|} \sum_{nd \in ND} \sum_{pf \in PF} \begin{cases} 1, & \text{if } nh_{t-1}(nd, pf) \neq nh_t(nd, pf), \\ 0, & \text{otherwise.} \end{cases} \tag{8}$$

Here, ND and PF represent all network devices and reachable network prefixes, respectively. $|ND| \cdot |PF|$ is the total routing table entries. The function $nh_t(nd, pf)$ denotes the next hop from device nd to prefix pf at timestep t. The traffic shift cost increments by 1 for each entry where $nh_{t-1}(nd, pf) \neq nh_t(nd, pf)$, indicating a change in routing. Higher traffic shift causes more severe mismatches between network resources and traffic demands, leading to overload in some areas and underutilization in others.

5.2 Agents Setup

In §5.1, we define the configuration update task, which requires updating three types of parameters from Table 3. This categorization is based on their distinct roles and functions in the network: BGP parameters manage inter-domain routing policies and path selection, OSPF parameters handle intra-domain routing and path calculation, while link attributes reflect physical characteristics and determine traffic-carrying capacity. We set parameter ranges to reflect generalized resource constraints inherent in real-world networks, capturing the limitations typically encountered during configuration updates.

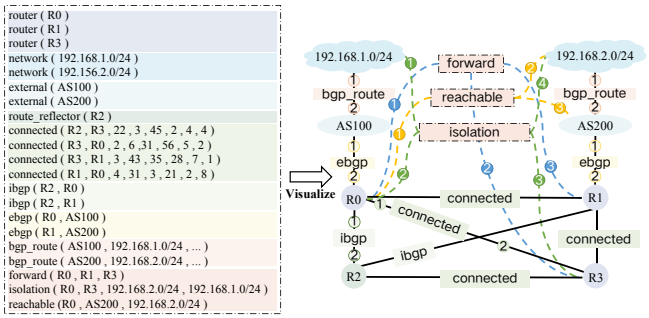


Figure 5: An example of network sketch.

For single-agent approach, configuration update task presents significant challenges due to its vast solution space, complex parameter interactions, and the need for specialized knowledge across different network domains (BGP, OSPF, and link attributes). To address these challenges, we employ a multi-agent for configuration update. This approach divides the complex task among multiple agents, each specializing in a specific parameter type, thereby reducing solution space complexity, enabling domain-specific optimization, and promoting inter-agent collaboration.

Hence, we employ three agents for configuration update: (1) A_{OSPF}: responsible for OSPF configuration update. (2) A_{BGP}: responsible for BGP configuration update. (3) A_{perf}: responsible for link attributes update. We define the set of agents as I, which contains three types of agent.

5.3 Environment Model

1. State Space S: We define the state $s \in S$ as the network sketch introduced in [15]. Fig. 5 illustrates a network sketch abstracting network information and forwarding policies as nodes with attributes. The sketch is visualized using different colors to represent various node types and connections between nodes.

Based on the network sketch, we encode the node features and connection relationships to obtain the state $s = (V, E)$, representing the network configuration. Here, $V = \{v_1, ..., v_n\}$ is the set of nodes, such as router, ibgp and isolation, where each $v_i = [f_{i,0}, \ldots, f_{i,19}]$ represents a node with its features. The edge information $E = (E_{adj}, E_{type}, E_{pos})$ consists of three components. $E_{adj} \in \mathbb{R}^{2\times m}$ is the adjacency matrix with m edges, where each column $[src_i, dst_i]$ represents an edge. $E_{type} = \{E_{type,1}, ..., E_{type,m}\}$ denotes edge types. $E_{pos} = \{E_{pos,1}, ..., E_{pos,m}\}$ indicates edge ordering information. For example, the *ibgp* node in Fig. 5 is represented as $ibgp(R2, R1)$ in the sketch. It connects to both $R1$ and $R2$ nodes, thus establishing two links numbered 1 and 2.

2. Action Space A: We define the action as $a_{i,k} = [f_{0,k}, \ldots, f_{n,k}]^T$, where k represents parameters in Table 3, n represents node number and i represents agent. Based on the agent setup in Section 5.1, different agents generate specific actions: A_{OSPF} produces $a_{OSPF,\alpha}$, A_{BGP} generates a_{BGP,γ_1}, a_{BGP,γ_2}, and a_{BGP,γ_3}, while A_{perf} creates a_{perf,β_1}, a_{perf,β_2}, and a_{perf,β_3}. These actions are combined into a joint action a. Subsequently, Alg. 1 is employed to update the current state s using this joint action, generating a new state s' by replacing the corresponding parameter k in s (lines 3 to 5).

3. Reward Function R: To output a configuration that satisfies policies and enhances network resilience, we need to guide the agents towards higher π with lower ρ and τ by designing reward functions R_s for policy consistency (Eq. 9) and network resilience R_d (Eq. 12), where τ and ρ jointly affect the latter. Agents receive larger rewards as π increases, while ρ and τ decrease. In addition, policy satisfaction is treated as the highest-priority objective, as it directly impacts the correctness of network operation, whereas load measurement and traffic shift are secondary goals to be optimized as much as possible. To enforce this prioritization, we introduce dynamic rewards (Eq. 10 and Eq. 11) in the design.

$$R(s,a)_{pol} = K \cdot \pi + R_s + R_d, \tag{9}$$

$$R_s = \begin{cases} 0, & \text{if } t = 0 \text{ or } a_t \neq a_{t-1}, \\ R_s - 1, & \text{if } a_t = a_{t-1}, \end{cases} \tag{10}$$

$$R_d = \begin{cases} 0, & \text{if } t = 0, \\ \sum_{p_i \in P}(Satisfy(p_i)_t - Satisfy(p_i)_{t-1}), & \text{if } t > 0. \end{cases} \tag{11}$$

$$R(s,a)_{res} = K \cdot ((1-\rho) + (1-\tau)). \tag{12}$$

Eq. 9 combines rewards for policy consistency with stationary and dynamic rewards to enhance the model's perception of action impacts on the state. The stationary reward R_s (Eq. 10) penalizes consecutive repetitions of the same action to prevent the agent from getting stuck in local optima. At timestep t, R_s resets to 0 when $t = 0$ or $a_t \neq a_{t-1}$; otherwise, decreasing by a fixed amount. The dynamic reward R_d (Eq. 11) encourages exploration of new configurations to meet previously unmet policies while preserving currently satisfied ones. Negative rewards are applied when previously satisfied policies become unsatisfied, balancing exploration and policy adherence. Given that R_S and R_d are integers while π, ρ, and τ are decimals not exceeding 1, a postive constant K is set to align the reward magnitudes.

Different agents use varied reward function combinations based on their specific tasks:

$$R(s,a)_{A_{OSPF}} = R(s,a)_{pol} + R(s,a)_{perf}, \tag{13}$$

$$R(s,a)_{A_{BGP}} = R(s,a)_{pol}, \tag{14}$$

$$R(s,a)_{A_{perf}} = R(s,a)_{perf}. \tag{15}$$

In Eq. 13, because OSPF weight impacts on traffic forwarding, affects both policy consistency and network performance, we combine two reward types. Due to the parameters generated by A_{BGP} and A_{OSPF} affect policy consistency and network performance respectively, their reward function values each include one component.

To enable agents to perceive traffic patterns and optimize network configuration, we provide a traffic matrix M_T, where

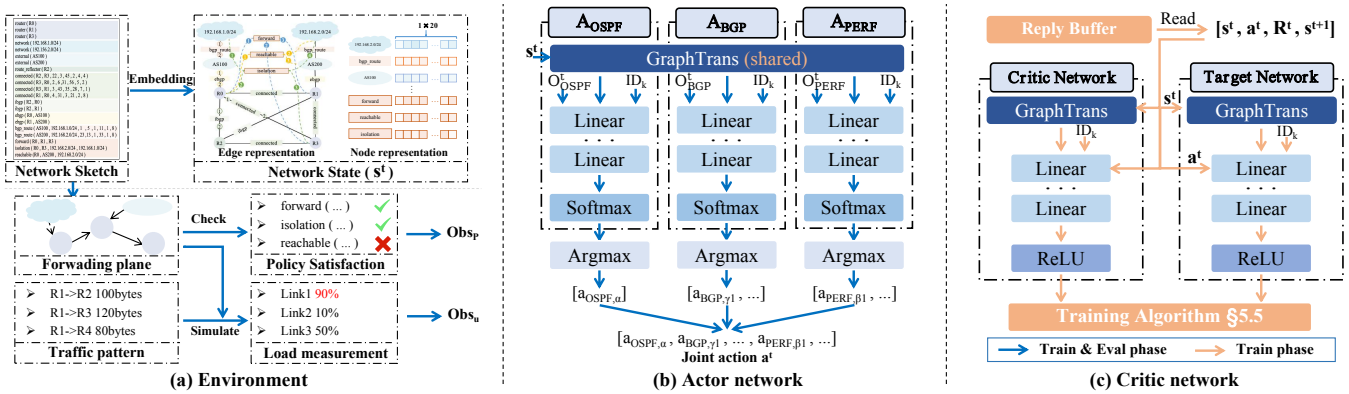


Figure 6: Architecture of configuration update model.

Algorithm 1: State Update Process

Input: Current state $s = (V, E)$, Joint action a
Output: New state s'

Initialize s' as a copy of s;
for *each action* $a_{i,k} \in a$ **do**
  $i_k \leftarrow$ index corresponding to parameter type k;
  for *each node* j *in* V' **do**
    $V'_j[i_k] \leftarrow a_{i,k[j]}$;
  end
end
return $s' = (V', E)$;

$M_{T(i,j)}$ records the traffic from $node_i$ to $node_j$. Agents can obtain observation information through the environment shown in Fig.6 (a), including policy satisfaction status and the utilization of links. Policy satisfaction is defined as a vector $Obs_p = [Satisfy(p_1), \ldots, Satisfy(p_i), \ldots, Satisfy(p_n)]$, where n is the number of polices. p_i is validated by computing its forwarding plane from the network configuration, which reveals how traffic is routed and whether the policies are correctly enforced. Link utilization is defined as a vector $Obs_u = [u_1, \ldots, u_m]$, where m is the number of links, and u_i reflects the utilization of the i-th link, indicating the congestion level of links. Each u_i is computed through Eq.2 - Eq.7.

5.4 Model Architecture

In our network, multiple agents update different parameters based on network information. Their collective decisions affect configurations, complicating individual contribution assessment. We employ an Actor-Critic (AC) [22] network with a centralized critic and decentralized actors. This balances efficient local decision-making by actors with global performance optimization by the critic. The overall architecture of the configuration update model is shown in Fig. 6, comprising actor networks corresponding to the agents, critic network and network environment.

The actor network generates actions based on the current state, implementing the agent's policy. It is designed as encoder-decoder architecture, where all agents share a common network sketch encoder $ENC_{network}$ in the lower layer to acquire the same network information, implemented using Graph Transformer (GraphTrans) [44]. The decision network $DEC_{LIN,i}$ of each agent i consists of L_{DEC} linear layers. Due to varied tasks, agents receive different observations, resulting in different input dimensions for their decision networks. Specifically, $O_{OSPF} = [Obs_p, Obs_u]$, $O_{BGP} = [Obs_p]$, and $O_{perf} = [Obs_u]$, where O_i represents the observation list for agent $i \in I$.

$$x_i = ENC_{network}(s^t), \tag{16}$$

$$x'_{i,k} = [x_i \| O_i \| ID_k], \tag{17}$$

$$h_{i,k} = \text{softmax}(DEC_{LIN,i}(x'_{i,k})), \tag{18}$$

$$a^t_{i,k} = \arg\max(h_{i,k}). \tag{19}$$

To obtain $a^t_{k,i}$, the action for parameter k generated by agent i at timestep t, as described in Eq. 16, we first embed the state s^t at timestep t using $ENC_{network}$. Then, the embedding x_i, along with the environment's observation O_i and the index of parameter ID_k, are merged through Eq. 17. ID_k represents the parameter generation ID where $k \in \{\alpha, \beta_1, \beta_2, \beta_3, \gamma_1, \gamma_2, \gamma_3\}$. The action probability distribution of k, denoted as $h_{i,k}$, is predicted through $DEC_{LIN,i}$ and softmax using Eq. 18. Finally, the action $a^t_{i,k}$ is obtained via arg max through Eq. 19.

$$y_i = ENC_{network}(s^t), \tag{20}$$

$$y'_{i,k} = [y_i \| a^{t-1}_{i,k} \| a^t_{i,k} \| ID_i], \tag{21}$$

$$Q(s^t, a^t_{i,k}) = \text{ReLU}(DEC_{LIN}(y'_{i,k})). \tag{22}$$

Critic network is used to evaluate the quality of the actor's decisions. The critic network has a similar architecture to the actor network. It ultimately needs to output a Q-value; we replace softmax and argmax with $ReLU$. The critic is only used during the training phase. To obtain $Q(s^t, a^t_{i,k})$, representing the estimated value of agent i taking action $a^t_{i,k}$ for parameter k in state s^t at timestep t, similarly to Eq. 16, state s^t is encoded using Eq. 20. We merge the encoding y_i with the agent ID ID_i, the current timestep's action $a^t_{i,k}$, and the previous timestep's action $a^{t-1}_{i,k}$ in Eq. 21 as input to predict $Q(s^t, a^t_{i,k})$ through Eq. 22.

5.5 Training Algorithm

The relationship between actions and rewards is often obscure, and the combination of actions influences the final outcome, necessitating careful consideration of each action's contribution. This makes training multiple agents particularly challenging, especially in credit assignment and decision-making under partial observability. We choose the counterfactual multi-agent (COMA) algorithm [11], which adopts the same architecture in §5.4. This approach allows for more accurate evaluation of each agent's contribution and informed decision-making with limited local information.

The training algorithm takes a network sketch s with n nodes and traffic matrix M_T as inputs. We initialize the actor network θ^{μ}, critic network θ^{Q}, target network $\theta^{Q'}$ (the latter two for improving training stability [14]), replay buffer B, agent mask M_k, and other parameters. We set EP training episodes, each ending when π reaches 1 or after TM timesteps.

In each episode, we initialize the environment and obtain state s^0. For each timestep t, We use the actor network θ^{μ} to select actions a^t for all agents, employing a decaying ε-greedy strategy. This strategy balances exploration (random strategy) and exploitation (learned policy) over time. We then execute these actions through Alg. 1 to obtain the new state s^{t+1}. We calculate the reward value R^t using Eq. 9 and Eq. 12 , and store the historical experience (s^t, a^t, R^t, s^{t+1}) in B.

We begin the training process when the data in B exceeds one entry. First, we use the target network $\theta^{Q'}$ to calculate the target Q-values $Q(s^t, a^t_{i,k})$ through Eq. 20 to Eq. 22. Subsequently, we compute the baseline $baseline_{i,k}$ and advantage $Advantage_{i,k}$. $baseline_{i,k}$ represents the average expected value of all possible actions for a given agent i and parameter k, calculated as the dot product of the action probability distribution $h_{i,k}$ and $Q(s^t, a^t_{i,k})$ in Eq.23. $Advantage_{i,k}$ measures how much better the chosen action is compared to $baseline$, computed by subtracting the $baseline_{i,k}$ from the Q-value of the selected action in Eq. 25. For nodes that do not include the parameter k, we set the advantage value to 0 through M_k.

$$baseline_{i,k} = h_{i,k} \cdot Q(s^t, a^t_{i,k}), \tag{23}$$

$$Q'(s^t, a^t_{i,k}) = \{Q(s^t, a^t_{i,k})_{j,\arg\max z} \mid j \in \{1, \ldots, n\}, z \in h_{i,k}\}, \tag{24}$$

$$Advantage_{i,k} = (Q'(s^t, a^t_{i,k}) - baseline_{i,k}) \odot M_k. \tag{25}$$

Finally, we calculate the actor network's loss, followed by backpropagation. Eq. 26 selects the most probable actions $h'_{i,k}$ from the original distribution $h_{i,k}$. Eq. 27 then computes the loss by averaging the product of advantage and log probability of these selected actions. This process aims to increase the likelihood of advantageous actions.

$$h'_{i,k} = \{z_{j,\arg\max z} \mid j \in \{1, \ldots, n\}, z \in h_{i,k}\}, \tag{26}$$

$$Loss_{Actor} = -\frac{1}{n} \sum_{j=1}^{n} Advantage_{i,k} \cdot \log(h'_{i,k}). \tag{27}$$

The critic network is updated to estimate Q-values, representing expected future rewards. This process begins with constructing inputs for each timestep using Eq. 20 to Eq. 22. Subsequently, Q-values $Q'(S^{t'}, a^{t'}_i)$ are calculated using the evaluation critic network θ^{μ}. These Q-values, along with corresponding rewards, are then selected based on action probabilities h_j using Eq. 28 and Eq. 29. Finally, masks are applied to the selected Q-values through Eq. 30.

$$Q'(s^{t'}, a^{t'}_{i,k}) = \left\{Q(s^{t'}, a^{t'}_{i,k})_{j,\arg\max z} \mid z \in h_{i,k}, j \in \{1, \ldots, n\}\right\}, \tag{28}$$

$$R^{t'} = \left\{R^{t'}_{j,\arg\max z} \mid z \in h_{i,k}, j \in \{1, \ldots, n\}\right\}, \tag{29}$$

$$Q'_M(s^{t'}, a^{t'}_{i,k}) = Q'(s^{t'}, a^{t'}_{i,k}) \odot M_k. \tag{30}$$

We use the $TD(\lambda)$ algorithm for learning updates, as shown in Eq. 31, which calculates the target value $r^{t'}$ for the critic network. If the episode ends, $r^{t'}$ equals the immediate reward $R^{t'}$. Otherwise, it combines the current Q-value with a correction term. This term includes the immediate reward $R^{t'+1}$, discounted future Q-value $\gamma \cdot Q'(s^{t'+1}, a^{t'}_{i,k})$, and current Q-value $Q'(s^{t'}, a^{t'}_{i,k})$. The equation uses lr_c and discount rate γ to balance current and future rewards.

$$r^{t'} = \begin{cases} R^{t'}, & \text{if } done, \\ Q'_M(s^{t'}, a^{t'}_{i,k}) + lr_c \cdot (R^{t'+1} & \\ \quad + \gamma \cdot Q'_M(s^{t'+1}, a^{t'}_{i,k}) - Q'_M(s^{t'}, a^{t'}_{i,k})), & \text{otherwise.} \end{cases} \tag{31}$$

We calculate $Loss_{Critic}$ using MSE between predicted $Q'_M(s^{t'}, a^{t'}_i)$ and target rt' values in Eq. 32, then backpropagate to update θ^{Q}.

$$Loss_{Critic} = \frac{1}{n} \sum_{j=1}^{n} \left(Q'_M(s^{t'}, a^{t'}_{i,k})_j - r^{t'}_j\right)^2. \tag{32}$$

We update the target critic network $\theta^{Q'}$ with the parameters of the evaluation critic network θ^{Q} at fixed intervals.

6 Evaluation

6.1 Setup

Dataset. We build two datasets: A network intent dataset and a network sketch dataset. The network intent dataset includes natural language intents, both manually written and template-generated, as well as anomaly logs collected from network maintenance platform and template-generated. Both contain mappings from network intents to DSL to API calls. The dataset size is 11,000, with a 7:4 ratio of intents to anomalies, and a 10:1 ratio of training to test data. The network sketch dataset is based on Topology Zoo [21] and comprises 258 real-world network topologies. These topologies are categorized into three groups: *small (S)* with 18–30 nodes, *medium (M)* with 30–56 nodes, and *large (L)* with 56–170 nodes, with 8 representative topologies sampled from each group. To further characterize task complexity, we generate three subsets based on the number of forwarding policies: 3×2, 3×8, and 3×16 with a total number of 1,024.

Environment for DRL. We use a Python-based network simulation environment for both online training of the agent and evaluation of *NetKeeper*. The environment incorporates

Table 4: Parameters in COMA algorithm.

Parameter	Value
Episodes (EP)	10
Timestep (TM)	200
Replay Buffer Size	100
Batch Size	16
Discount Factor (γ)	0.85
Initial Exploration Rate	1.0 (decaying)
Final Exploration Rate	0.01
Exploration Decay Rate	0.99 per timestep
Soft Target Update Interval	16 timestep
K in reward function	10
Disobey reward in reward function	-1

Table 5: Parameters of network.

Parameter	Value	Parameter	Value
OSPF weight (α)	[1, 64]	Local preference (γ_1)	[1, 64]
Bandwidth (β_1)	(64, 128]	AS path length (γ_2)	[1, 64]
Capacity (β_2)	(64, 128]	Multi-exit discriminator (γ_3)	[1, 64]
Queue length (β_3)	(64, 128]	Packet size (q_s)	5
Lose rate (M_{Loss})	[10, 50] %		

variations in topology size, routing protocol configurations (OSPF/BGP), forwarding policies, and traffic patterns, enabling the agent to adapt to diverse network scenarios. For testing, the agent is evaluated on fixed subsets of the network sketch dataset, ensuring consistent performance evaluation across varying topology sizes and policy complexities.

Testing Platform and Experiment Parameter. All experiments are performed on a machine equipped with an NVIDIA GeForce RTX 3090 GPU with 24GB VRAM, 64GB RAM, and a 13th Gen Intel Core i9-13900K processor. To ensure the fairness of our experiments, all experiments are conducted using the same seed. We fine-tuned the ChatGLM3-6B[13] through P-tuning using Adam [19] optimizer with a learning rate of 0.02 for 400 steps and deployed it within the LangChain. Other hyperparameters remained unchanged from the default settings. The Actor-Critic network is trained using COMA algorithm. The actor and critic networks share identical structures: an encoder $ENC_{GraphTrans}$ with 8 graph convolutional network (GCN) [20] and 8 transformer [42] layers, and a decoder DEC_{LIN} with 4 linear layers. All layers have a hidden dimension of 128. We use Adam optimizer with learning rates of 0.0001 and 0.0002 for actor and critic networks respectively. Other hyperparameters are listed in Table 4 and network parameter settings are listed in Table 5.

Metrics and Comparisons. We evaluate *NetKeeper* based on intent translation accuracy, configuration quality, load measurement, traffic shift, and adaptability to dynamic networks. Intent translation accuracy refers to the correct invocation of APIs based on natural language intents or anomaly logs. Configuration quality depends on the policy consistency (Eq.1). Load measurement is quantified by maximum link utility, using a routing simulation program described in §5.1. Traffic shift (Eq.8) evaluates the degree of traffic redistribution after configuration update. We also assess network resilience improvements through simulations of dynamic changes and anomalies. For comparison, we use three categories of baselines: SMT-based [8] methods like NetComplete [10], neural network-based methods like GAT [7], and SMT+supervised learning hybrids like NetRen [15], which integrates SMT solvers with neural network predictions, falling back on neural network's outputs when the solver fails (overtime).

Table 6: Intent translation under three different tasks with different conditions.

Task	Feedback	DSL	Precision	Recall	F1
NL2API	✗	✗	83%	79%	0.81
NL2API	✓	✗	88%	83%	0.85
NL2API	✗	✓	92%	90%	0.91
NL2API	✓	✓	94%	93%	0.93
AL2API	✗	✗	90%	88%	0.89
AL2API	✓	✗	95%	94%	0.95
AL2API	✗	✓	97%	96%	0.96
AL2API	✓	✓	100%	100%	1.00
IT2API	✓	✓	**96%**	**94%**	**0.95**

6.2 Intent Translation Accuracy

To evaluate intent translation accuracy for natural language intents (NL) and anomaly logs (AL), we measure precision, recall, and F1 scores across three tasks: `NL2API`, `AL2API`, and `IT2API` (a combined task integrating both NL and AL scenarios), under different conditions, including the presence or absence of feedback and the use of DSL as an intermediary.

Table 6 demonstrates that the introduction of DSL and feedback significantly enhances performance, achieving a precision of 96%, recall of 94%, and an F1 score of 0.95 for `IT2API`. Additionally, 58 participants were involved in the manual feedback process, with each feedback iteration averaging 2s in duration, and erroneous translations were typically resolved with only one round of feedback. Compared to the baseline (without DSL or feedback), `NL2API` exhibits notable improvements: precision increases by 11%, recall by 14%, and the F1 score by 0.12. Similarly, `AL2API` achieves a 10% boost in precision, a 12% increase in recall, and a 0.11 gain in the F1 score. These results demonstrate the critical role of DSL and feedback in enhancing both the accuracy and reliability of intent translation, improving API invocation precision, reducing errors, and supporting diverse intent types in combined tasks.

6.3 Configuration Quality

To evaluate the ability of updating network configuration, we evaluate the policy consistency and time consuming across tasks with varying complexity, including different topology scales, policy categories and policy quantities. We replicate Netcomplele (SMT), GAT and NetRen as benchmarks, following the experimental settings in work [15], and present the average policy consistency. Because SMT solving can be time-consuming on complex tasks, we set a time limit of

Table 7: Comparison of average policy consistency under tasks of varying complexity.

Task	Topology Scale	Forward				Reachable				Isolation				Overall			
		SMT	GAT	NetRen	Ours	SMT	GAT	NetRen	Ours	SMT	GAT	NetRen	Ours	SMT	GAT	NetRen	Ours
3 × 2	S	0.98	1.00	1.00	**1.00**	0.98	0.96	1.00	**1.00**	0.99	0.58	1.00	**1.00**	0.98	0.86	1.00	**1.00**
	M	0.99	1.00	1.00	**1.00**	0.98	0.88	1.00	**1.00**	1.00	0.75	0.94	**1.00**	0.99	0.87	0.98	**1.00**
	L	0.99	1.00	1.00	**1.00**	0.99	1.00	1.00	**1.00**	0.98	0.79	0.94	**1.00**	0.98	0.89	0.98	**1.00**
3 × 8	S	0.98	0.96	1.00	**1.00**	0.98	0.96	0.98	**1.00**	0.99	0.79	1.00	**1.00**	0.98	0.89	0.99	**1.00**
	M	0.99	0.98	1.00	**1.00**	0.99	0.98	0.98	**1.00**	0.99	0.75	1.00	**1.00**	0.99	0.91	0.99	**1.00**
	L	0.98	0.98	1.00	**1.00**	0.99	0.95	1.00	**1.00**	0.98	0.75	0.96	**1.00**	0.99	0.89	0.98	**1.00**
3 × 16	S	0.98	0.94	0.99	**1.00**	0.99	0.98	0.95	**1.00**	0.98	0.78	0.96	**1.00**	0.98	0.91	0.98	**1.00**
	M	0.99	0.98	0.99	**1.00**	0.99	0.95	0.95	**1.00**	0.99	0.73	0.97	**1.00**	0.99	0.91	0.98	**1.00**
	L	0.99	0.97	0.99	**1.00**	0.99	0.96	0.99	**1.00**	0.99	0.79	0.94	**1.00**	0.99	0.92	0.98	**0.99**

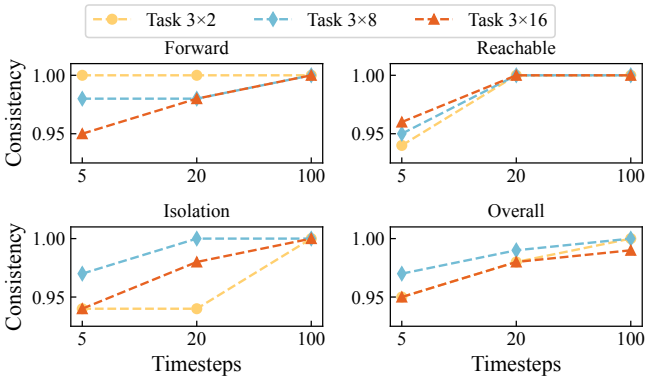


Figure 7: Average policy consistency within limited timesteps.

Table 8: comparison of average time consuming under tasks of varying complexity (1500+ means timeout).

Task	Topology Scale	SMT(s)	GAT(s)	NetRen(s)	Ours(s)
3 × 2	S	7.68	2.62	0.37	**0.31**
	M	98.38	4.37	0.41	**0.34**
	L	1329.32	13.41	1.26	**0.51**
3 × 8	S	58.09	4.26	0.64	**0.42**
	M	148.41	5.89	0.81	**0.47**
	L	1500+	17.35	**2.73**	3.01
3 × 16	S	283.16	5.25	1.13	**1.04**
	M	317.24	6.39	**1.78**	2.08
	L	1500+	20.71	**3.16**	4.07

1500 seconds for all methods and methods that fail to generate a valid configuration within this time are considered failures. Additionally, since *NetKeeper* uses DRL for configuration update, enhancing configuration quality through multiple timesteps and environmental interaction, We evaluate the capability by measuring average policy consistency within a limited timesteps.

Fig. 7 shows the average policy consistency achieved by *NetKeeper* within 5, 20, and 100 timesteps, with 96.2%, 98.7%, and 99.6% respectively, demonstrating a continuous improvement in consistency as timestep (average 0.4s) increases. Moreover, Table 7 demonstrates that *NetKeeper* achieves a policy consistency of 99.6%, outperforming SMT, GAT and NetRen with improvement of 1.3%, 10.4% and 1.2% respectively. Additionally, Table 8 shows that in the most complex network synthesis scenarios, NetRen achieves an average running time of 3.16 seconds with a policy consistency of 98% and a 2% configuration synthesis timeout rate. In comparison, *NetKeeper* achieves a slightly higher average running time of 4.07 seconds, but with a higher policy consistency of 99.6%, a much lower timeout rate of only 0.04%, and superior results in terms of network performance and traffic shift. Notably, *NetKeeper* can exceed NetRen's policy satisfaction rate (98%) within no more than 8 iterations (approximately 3.2 seconds), while maintaining a comparable average running time. These results suggests that *NetKeeper* exhibits superior adaptability across various network scenarios in configuration update compared to both SMT, GAT and NetRen.

6.4 Load Measurement

To test the network performance optimization capability, we evaluate load measurement by computing maximum link utility under different network scenarios with random traffic.

We set three types of network scenarios: `Ideal`: This scenario disregards negative factors, including packet loss and latency, and queue length limitations; `Normal`: This scenario considers all link attributions; `High-load`: This scenario increases the load to three times that of the normal test scenario. The design of these scenarios reflects real-world network conditions: `Ideal` represents an optimized network with minimal interference; `Normal` captures typical operational environments with realistic link attributions; `High-load` emulates peak traffic or network stress conditions. Although the traffic in this study is synthetically generated, it reflects patterns commonly observed in real-world networks.

In addition, under these scenarios, we establish 4 traffic generation ranges reflcting different traffic patterns, including 1∼2, 2∼4, 4∼8, and 8∼16. In high-load scenarios, the maximum range for a single traffic flow reaches 24 to 64. To make the model more focused on network performance optimization, we set only 3×2 network policies for all network topologies in this case. We establish three parameter modes for comparison: (1) Random, (2) OSPF default mode, and (3) NetRen. Due to the occurrence of excessively high utilization, we set maximum link utilization exceeding 1.1 as ≥ 1.1.

Comparing our method with Random, OSPF default mode, and NetRen, Table 9 shows that *NetKeeper* outperforms these three in most network performance optimization tasks. Random and OSPF default mode exceed our critical utilization threshold in most cases. Compared to NetRen, *NetKeeper* achieves average congestion mitigation of 6.3%, 3.9%, and

Table 9: Comparison of maximum link utility under three scenarios and four traffic patterns.

Topology Scale	Traffic Range (bytes)	Ideal				Normal				High-load			
		Random	OSPF	NetRen	Ours	Random	OSPF	NetRen	Ours	Random	OSPF	NetRen	Ours
S	1∼2	0.89	0.49	0.02	**0.01**	0.64	0.76	0.03	**0.02**	>1.1	>1.1	0.06	0.06
	2∼4	0.70	0.96	0.05	0.05	1.04	0.99	0.06	**0.04**	>1.1	>1.1	0.11	**0.10**
	4∼8	>1.1	>1.1	0.09	**0.08**	>1.1	>1.1	0.10	0.10	>1.1	>1.1	0.26	**0.20**
	8∼16	>1.1	>1.1	0.23	**0.22**	>1.1	>1.1	0.25	**0.22**	>1.1	>1.1	0.48	0.48
M	1∼2	0.86	>1.1	0.06	**0.04**	0.60	0.84	0.04	0.04	>1.1	>1.1	0.09	0.09
	2∼4	>1.1	>1.1	0.18	**0.11**	>1.1	>1.1	0.10	**0.08**	>1.1	>1.1	0.16	**0.15**
	4∼8	>1.1	>1.1	0.43	**0.25**	>1.1	>1.1	0.29	**0.21**	>1.1	>1.1	0.37	0.40
	8∼16	>1.1	>1.1	0.40	**0.38**	>1.1	>1.1	0.40	0.40	>1.1	>1.1	0.73	**0.60**
L	1∼2	>1.1	>1.1	0.10	**0.09**	>1.1	>1.1	0.05	0.05	>1.1	>1.1	0.13	**0.07**
	2∼4	>1.1	>1.1	0.18	**0.15**	>1.1	>1.1	0.12	0.12	>1.1	>1.1	0.25	**0.18**
	4∼8	>1.1	>1.1	0.43	**0.32**	>1.1	>1.1	0.29	**0.23**	>1.1	>1.1	0.49	**0.43**
	8∼16	>1.1	>1.1	0.78	**0.66**	>1.1	>1.1	0.68	**0.53**	>1.1	>1.1	1.06	**0.80**

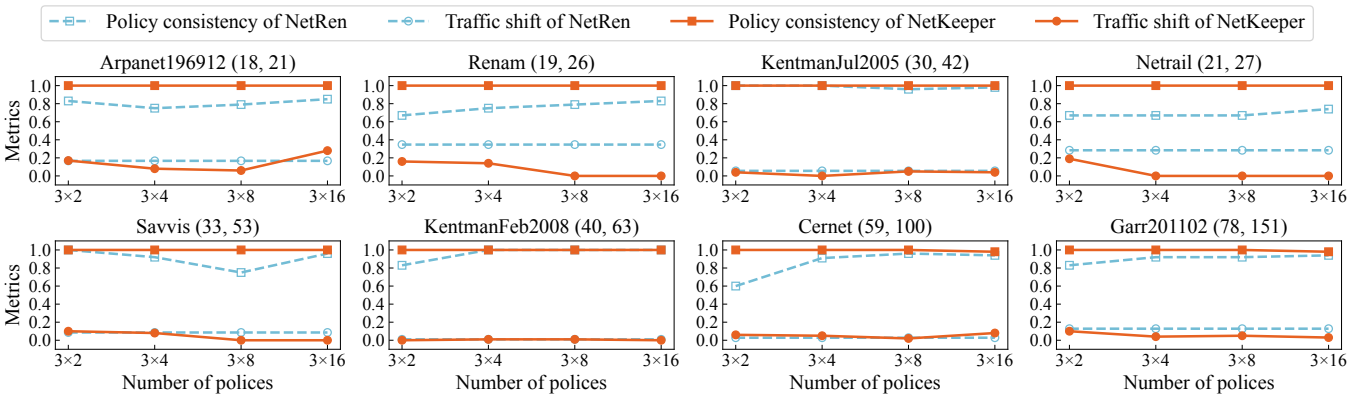


Figure 8: Traffic shift during configuration updates and policy consistency with policy count increases. Each subplot is titled with the topology name, its corresponding number of node and edge.

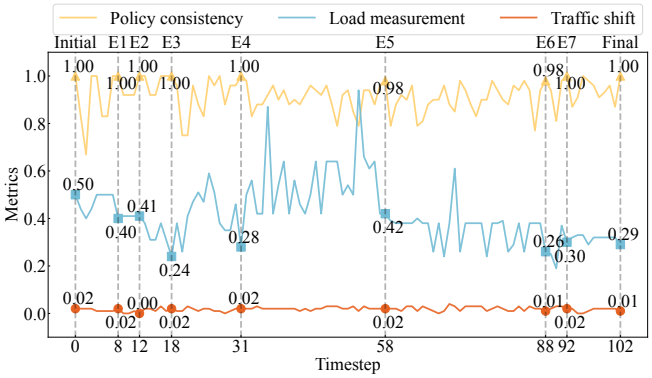


Figure 9: *NetKeeper*'s performance in dynamic network: Policy consistency, Load measurement and Traffic shift.

5.8% in the three test scenarios respectively, with maximum mitigation reaching 18.0%, 15.0%, and 26.0%, resulting in an overall network performance improvement of 5.3%. This indicates that *NetKeeper* is capable of analyzing traffic patterns and performing better network performance optimization based on the current network conditions.

6.5 Traffic Shift

To evaluate traffic shifts during the configuration update process, we sample four groups of networks with cumulatively increasing numbers of policies: 3×2, 3×4, 3×8, and 3×16. Each configuration builds upon the constraints of the previous one, generating topologies with cumulative policies. We select NetRen as the baseline.

Fig. 8 depicts traffic shift and policy consistency after configuration update. *NetKeeper* outperforms NetRen with lower average traffic shift of 5.9% versus 13.9% and higher policy consistency of 99.9% versus 85.8%. Compared to NetRen, *NetKeeper* can reduce traffic shift by an average of 8.7% and a maximum of 34.7%. *NetKeeper* significantly outperforms NetRen in network update, demonstrating superior efficiency through reduced traffic shift and higher policy consistency.

6.6 Adaptability To Dynamic Network

To evaluate *NetKeeper*'s adaptability to dynamic network, we simulate dynamic environments with three types of events reflecting challenges commonly encountered in real-world scenarios: `Physical`: Including device failures, link failures, etc; `Forwarding`: Addition or reduction of forwarding policies, thereby limiting traffic forwarding paths; `Network traffic`: Link congestion or underutilization requiring load balancing.

This experiment is conducted on the Pern network, a real-world topology with 125 nodes, 129 edges, and 3×2 policies. We simulate seven sequential events (E1-E7), with *NetKeeper* updating configurations after each. These events include policy expansions (E1-E4), link load fluctuations (E2 and E5), and component removals (E6 and E7). The changes progressively increase complexity, reduce performance, and weaken

```
2024-10-04 17:05:38 step:9 policy_consistency: 0.77 max_utility: 0.11 traffic_shift: 0.11
2024-10-04 17:05:39 step:10 policy_consistency: 0.91 max_utility: 0.11 traffic_shift: 0.11
2024-10-04 17:05:41 step:11 policy_consistency: 0.79 max_utility: 0.14 traffic_shift: 0.00
2024-10-04 17:05:42 step:12 policy_consistency: 0.89 max_utility: 0.14 traffic_shift: 0.00
2024-10-04 17:05:44 step:13 policy_consistency: 0.91 max_utility: 0.14 traffic_shift: 0.11
2024-10-04 17:05:45 step:14 policy_consistency: 0.72 max_utility: 0.14 traffic_shift: 0.11
2024-10-04 17:05:47 step:15 policy_consistency: 0.96 max_utility: 0.14 traffic_shift: 0.09
2024-10-04 17:05:48 step:16 policy_consistency: 1.00 max_utility: 0.14 traffic_shift: 0.02
```

Figure 10: Network optimization logs in production network.

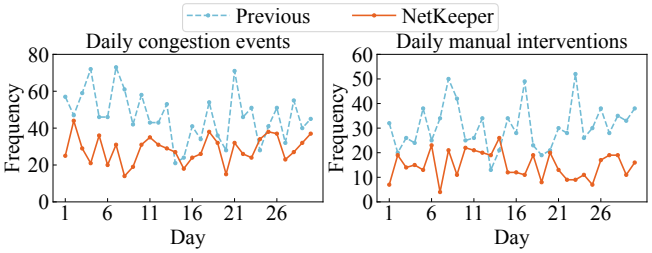


Figure 11: *NetKeeper*'s performance in production network.

connectivity. Each event concludes when consistency reaches 100% or exceeds 30 timesteps (A timestep ranges from 0.5s to 1s).

Fig. 9 shows how *NetKeeper* updates network configurations in response to different events. We observe that *NetKeeper* generally achieves full policy satisfaction alongside network optimization. However, under certain exceptional conditions, this may not hold. In E4, policy conflicts—stemming from natural language inputs that may contain inconsistencies—limit satisfaction to a maximum of 98%. In contrast, in E5, despite link load fluctuations, *NetKeeper* maintains full policy satisfaction while optimizing load distribution. In E6, device failures render associated policies inherently unsatisfiable; these are excluded from reconfiguration, and all remaining conflict-free policies are fully satisfied. This multi-faceted capability showcases *NetKeeper*'s flexibility in dynamic network. Furthermore, *NetKeeper* improves network performance while maintaining stability, demonstrating its practical value in dynamic network management.

6.7 Deployment in Production Network

To evaluate *NetKeeper*'s performance in real world, we deployed it in a large-scale enterprise network with hundreds of routers and switches. The network handles high daily traffic and supports critical applications like video conferencing and cloud services. *NetKeeper* was integrated into the existing network management system. As shown in Fig. 10, part of the deployment logs highlights the optimization of network metrics when facing network congestion.

During deployment, *NetKeeper* achieved 94.8% correct reconfigurations, with 2.1% false negatives and 3.1% false alarms, while safeguards effectively prevented cascading failures during anomaly responses. Fig. 11 illustrates a 39.4% improvement in network resilience (congestion event reduction) and a 51.5% decrease in manual interventions due to the natural language interface and automation. These enhancements enabled operators to focus on strategic tasks like network design and policy optimization, demonstrating significant improvements in network performance and operational efficiency through *NetKeeper*.

7 Limitations

NetKeeper still has limitations, which we will further address as future work.

Limited Autonomy in Policy Generation. *NetKeeper* enables network configuration updates based on operators' intent, yet still requires human involvement in strategic planning and decision-making when handling business requirement changes or network environment shifts. Specifically, network operators must provide granular policies under such dynamic conditions. Currently, the system cannot autonomously generate policy-level configurations directly from business objectives — automated policy generation adapted to specific business-network contexts remains a critical objective for our future work.

Limited Traffic Awareness for Resilience. *NetKeeper* considers traffic patterns when updating configurations to improve resilience, but its current traffic modeling is coarse-grained. It lacks fine-grained analysis of bursty flows or application-specific behaviors, limiting its adaptability to complex or rapidly changing traffic. Enhancing traffic profiling remains an important direction for future improvement.

8 Conclusion

To address the challenges of aligning network intent, optimizing networks based on traffic patterns, and adapting to dynamic network, we propose an autonomous network configuration update framework. Our framework incorporates a bidirectional north-south interface that aligns intents from user inputs and network anomalies. We design a dynamic network configuration update model based on MADRL, enhancing network resilience based on analyzing traffic patterns. *NetKeeper* effectively translates multimodal intents, enabling configuration updates through natural language interactions and facilitating network self-management and update based on anomaly logs. Furthermore, *NetKeeper* achieves a higher degree of automated configuration update, reduces manual intervention and potential human errors.

Acknowledgments

We thank our shepherd, Tieying Zhang, and the anonymous reviewers for their insightful comments. This work was supported in part by the National Key R&D Program of China 2024YFE0200800, the National Natural Science Foundation of China under Grants (62201072, 62321001, 62471055, U23B2001, 62401080, 62101064, 62171057, 62071067), the High-Quality Development Project of the MIIT(2440STCZB2584), the Ministry of Education and China Mobile Joint Fund (MCM20200202, MCM20180101), the Fundamental Research Funds for the Central Universities (2024PTB-004)

References

[1] Anubhavnidhi Abhashkumar, Aaron Gember-Jacobson, and Aditya Akella. AED: incrementally synthesizing policy-compliant and manageable configurations. In *CoNEXT '20: The 16th International Conference on emerging Networking EXperiments and Technologies*, Barcelona, Spain, December 2020.

[2] Kai Arulkumaran, Marc Peter Deisenroth, Miles Brundage, and Anil Anthony Bharath. Deep reinforcement learning: A brief survey. *IEEE Signal Process. Mag.*, 34(6):26–38, June 2017.

[3] Mahmoud Bahnasy, Fenglin Li, Shihan Xiao, and Xiangle Cheng. DeepBGP: A machine learning approach for BGP configuration synthesis. In *Proceedings of the 2020 Workshop on Network Meets AI & ML, NetAI@SIGCOMM*, Virtual Event, USA, August 2020.

[4] Ryan Beckett, Ratul Mahajan, Todd D. Millstein, Jitendra Padhye, and David Walker. Don't mind the gap: Bridging network-wide objectives and device-level configurations. In *Proceedings of the ACM SIGCOMM 2016 Conference*, Florianopolis, Brazil, August 2016.

[5] Ryan Beckett, Ratul Mahajan, Todd D. Millstein, Jitendra Padhye, and David Walker. Network configuration synthesis with abstract topologies. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, Barcelona, Spain, June 2017.

[6] Theophilus Benson, Aditya Akella, and David A. Maltz. Network traffic characteristics of data centers in the wild. In *Internet Measurement Conference*, Melbourne, Australia, November 2010.

[7] Luca Beurer-Kellner, Martin T. Vechev, Laurent Vanbever, and Petar Velickovic. Learning to configure computer networks with neural algorithmic reasoning. In Sanmi Koyejo, S. Mohamed, A. Agarwal, Danielle Belgrave, K. Cho, and A. Oh, editors, *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems*, New Orleans, LA, November 2022.

[8] Leonardo Mendonça de Moura and Nikolaj S. Bjørner. Z3: an efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008*, Budapest, Hungary, March 2008.

[9] Ahmed El-Hassany, Petar Tsankov, Laurent Vanbever, and Martin T. Vechev. Network-wide configuration synthesis. In *Computer Aided Verification - 29th International Conference*, Heidelberg, Germany, July 2017.

[10] Ahmed El-Hassany, Petar Tsankov, Laurent Vanbever, and Martin T. Vechev. Netcomplete: Practical network-wide configuration synthesis with autocompletion. In *NSDI*, Renton, WA, USA, April 2018.

[11] Jakob N. Foerster, Gregory Farquhar, Triantafyllos Afouras, Nantas Nardelli, and Shimon Whiteson. Counterfactual multi-agent policy gradients. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18)*, Louisiana, USA, February 2018.

[12] TM Forum. Autonomous networks technical architecture v1.1.1 (ig1230), Dec. 2022. https://www.tmforum.org/resources/toolkit/autonomous-networks-technical-architecture/, Last accessed on 2024-9-13.

[13] Team GLM, Aohan Zeng, Bin Xu, Bowen Wang, Chenhui Zhang, Da Yin, Diego Rojas, Guanyu Feng, Hanlin Zhao, Hanyu Lai, Hao Yu, Hongning Wang, Jiadai Sun, Jiajie Zhang, Jiale Cheng, Jiayi Gui, Jie Tang, Jing Zhang, Juanzi Li, Lei Zhao, Lindong Wu, Lucen Zhong, Mingdao Liu, Minlie Huang, Peng Zhang, Qinkai Zheng, Rui Lu, Shuaiqi Duan, Shudan Zhang, Shulin Cao, Shuxun Yang, Weng Lam Tam, Wenyi Zhao, Xiao Liu, Xiao Xia, Xiaohan Zhang, Xiaotao Gu, Xin Lv, Xinghan Liu, Xinyi Liu, Xinyue Yang, Xixuan Song, Xunkai Zhang, Yifan An, Yifan Xu, Yilin Niu, Yuantao Yang, Yueyan Li, Yushi Bai, Yuxiao Dong, Zehan Qi, Zhaoyu Wang, Zhen Yang, Zhengxiao Du, Zhenyu Hou, and Zihan Wang. Chatglm: A family of large language models from glm-130b to glm-4 all tools, June 2024.

[14] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *Proceedings of the 35th International Conference on Machine Learning,ICML 2018*, Sweden, July 2018.

[15] Rongxin Han, Jingyu Wang, Qi Qi, Haifeng Sun, Chaowei Xu, Zhaoyang Wan, Zirui Zhuang, Yichuan Yu, and Jianxin Liao. Netren: Service migration-driven network renascence with synthesizing updated configuration. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3, ASPLOS 2024*, La Jolla, CA, USA, April 2024.

[16] Christian E. Hopps. Analysis of an equal-cost multi-path algorithm. *RFC*, 2992:1–8, November 2000.

[17] Arthur Selle Jacobs, Ricardo J. Pfitscher, Rafael Hengen Ribeiro, Ronaldo A. Ferreira, Lisandro Zambenedetti Granville, Walter Willinger, and Sanjay G. Rao. Hey, lumi! using natural language for intent-based network management. In *Proceedings of the 2021 USENIX Annual Technical Conference, USENIX ATC 2021*, April 2021.

[18] Tatsuaki Kimura, Keisuke Ishibashi, Tatsuya Mori, Hiroshi Sawada, Tsuyoshi Toyono, Ken Nishimatsu, Akio Watanabe, Akihiro Shimoda, and Kohei Shiomoto. Spatio-temporal factorization of log data for understanding network events. In *2014 IEEE Conference on Computer Communications, INFOCOM 2014*, Toronto, Canada, April 2014.

[19] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015*, San Diego, CA, USA, May 2015.

[20] Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. In *5th International Conference on Learning Representations, ICLR 2017*, Toulon, France, April 2017.

[21] Simon Knight, Hung X. Nguyen, Nick Falkner, Rhys Alistair Bowden, and Matthew Roughan. The internet topology zoo. *IEEE J. Sel. Areas Commun.*, 29(9):1765–1775, 2011.

[22] Vijaymohan Konda and Vivek S. Borkar. Actor-critic - type learning algorithms for markov decision processes. *SIAM J. Control. Optim.*, 38(1):94–123, February 1999.

[23] Rupa Krishnan, Harsha V. Madhyastha, Sridhar Srinivasan, Sushant Jain, Arvind Krishnamurthy, Thomas E. Anderson, and Jie Gao. Moving beyond end-to-end path information to optimize CDN performance. In *Proceedings of the 9th ACM SIGCOMM Internet Measurement Conference, IMC 2009*, Chicago, Illinois, USA, November 2009.

[24] Xingjian Liao, Haifeng Sun, Jingyu Wang, Qi Qi, Zirui Zhuang, Jianxin Liao, and Guang Yang. Solving distributed ACL policies under complex constraints with graph neural networks. In *31st IEEE International Conference on Network Protocols, ICNP 2023*, Reykjavik, Iceland, October 2023.

[25] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. In *4th International Conference on Learning Representations, ICLR 2016*, San Juan, Puerto Rico, May 2016.

[26] Xiao Liu, Kaixuan Ji, Yicheng Fu, Zhengxiao Du, Zhilin Yang, and Jie Tang. P-tuning v2: Prompt tuning can be comparable to fine-tuning universally across scales and tasks. *CoRR*, abs/2110.07602, December 2021.

[27] Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4):316–344, July 2005.

[28] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin A. Riedmiller, Andreas Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nat.*, 518(7540):529–533, February 2015.

[29] John Moy. OSPF version 2. *RFC*, 2328:1–244, July 1998.

[30] Sanjai Narain, Gary Levin, Sharad Malik, and Vikram Kaul. Declarative infrastructure configuration synthesis and debugging. *J. Netw. Syst. Manag.*, 16(3):235–258, October 2008.

[31] Sivaramakrishnan Ramanathan, Ying Zhang, Mohab Gawish, Yogesh Mundada, Zhaodong Wang, Sangki Yun, Eric Lippert, Walid Taha, Minlan Yu, and Jelena Mirkovic. Practical intent-driven routing configuration synthesis. In *20th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2023*, Boston, MA, April 2023.

[32] Yakov Rekhter, Tony Li, and Susan Hares. A border gateway protocol 4 (BGP-4). *RFC*, 4271:1–104, January 2006.

[33] Ravi S. Sandhu and Pierangela Samarati. Access control: principles and practice. *IEEE Commun. Mag.*, 32(9):40–48, August 1994.

[34] Tibor Schneider, Rüdiger Birkner, and Laurent Vanbever. Snowcap: synthesizing network-wide configuration updates. In *ACM SIGCOMM 2021 Conference*, Virtual Event, USA, August 2021.

[35] Tibor Schneider, Roland Schmid, and Laurent Vanbever. On the complexity of network-wide configuration synthesis. In *30th IEEE International Conference on Network Protocols, ICNP*, Lexington, KY, USA, October 2022.

[36] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodík, Sanjit A. Seshia, and Vijay A. Saraswat. Combinatorial sketching for finite programs. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2006*, San Jose, CA, USA, October 2006.

[37] Robert Soulé, Shrutarshi Basu, Parisa Jalili Marandi, Fernando Pedone, Robert D. Kleinberg, Emin Gün Sirer, and Nate Foster. Merlin: A language for provisioning network resources. In *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies, CoNEXT 2014*, Sydney, Australia, December 2014.

[38] Mohammed H. Sqalli, Sadiq M. Sait, and Mohammed Aijaz Mohiuddin. An enhanced estimator to multi-objective OSPF weight setting problem. In *Management of Integrated End-to-End Communications and Services, 10th IEEE/IFIP Network Operations and Management Symposium, NOMS 2006*, Vancouver, Canada, April 2006.

[39] James PG Sterbenz, David Hutchison, Egemen K Çetinkaya, Abdul Jabbar, Justin P Rohrer, Marcus Schöller, and Paul Smith. Resilience and survivability in communication networks: Strategies, principles, and survey of disciplines. *Computer networks*, 54(8):1245–1265, March 2010.

[40] Kausik Subramanian, Loris D'Antoni, and Aditya Akella. Genesis: synthesizing forwarding tables in multi-tenant networks. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017*, Paris, France, January 2017.

[41] Oguzhan Topsakal and Tahir Cetin Akinci. Creating large language model applications utilizing langchain: A primer on developing llm apps fast. In *International Conference on Applied Engineering and Natural Sciences*, volume 1, pages 1050–1056, July 2023.

[42] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017*, Long Beach, CA, USA, December 2017.

[43] Changjie Wang, Mariano Scazzariello, Alireza Farshin, Simone Ferlin, Dejan Kostic, and Marco Chiesa. Netconfeval: Can llms facilitate network configuration? *Proc. ACM Netw.*, 2(CoNEXT2):1–25, June 2024.

[44] Zhanghao Wu, Paras Jain, Matthew A. Wright, Azalia Mirhoseini, Joseph E. Gonzalez, and Ion Stoica. Representing long-range context for graph neural networks with global attention. In *Advances in Neural Information Processing Systems 34: Annual Conferenceon Neural Information Processing Systems 2021, NeurIPS 2021*, virtual, December 2021.

[45] Kaiqing Zhang, Zhuoran Yang, and Tamer Basar. Multi-agent reinforcement learning: A selective overview of theories and algorithms. *CoRR*, abs/1911.10635:321–384, June 2019.

[46] Jinyu Zhao, Haifeng Sun, Jingyu Wang, Qi Qi, Zirui Zhuang, Shimin Tao, and Jianxin Liao. CONFPILOT: A pilot for faster configuration by learning from device manuals. In *43rd IEEE International Conference on Distributed Computing Systems, ICDCS 2023*, Hong Kong, October 2023.

GREYHOUND: Hunting Fail-Slows in Hybrid-Parallel Training at Scale

Tianyuan Wu[†], Wei Wang[†*], Yinghao Yu[§], Siran Yang[§], Wenchao Wu[§],
Qinkai Duan[†], Guodong Yang[§], Jiamang Wang[§], Lin Qu[§], Liping Zhang[§]
[†]*Hong Kong University of Science and Technology*
[§]*Alibaba Group*

Abstract

Fail-slows, or stragglers, are common problems in large-scale hybrid-parallel training that runs on a large fleet of GPU servers for an extended period of time. Yet, these problems are not well studied. In this paper, we first present a characterization study on a shared production cluster with over 10,000 GPUs. We find that fail-slows manifest as *transient stragglers* caused by slow computations or communications due to contention, device degradation, or network congestion, lasting from sub-minutes to nearly ten hours, and delaying large training jobs by 1.34× on average. The current practice is to manually detect fail-slows and treat them as fail-stops by means of checkpoint-and-restart failover, which is time-consuming. In this paper, we propose GREYHOUND, a system that rapidly identifies slow GPUs and/or communication links, and effectively tackles them with a novel multi-level mitigation mechanism, all without human intervention. GREYHOUND correctly detects fail-slows in a production cluster with over 99% accuracy. Testbed experiment on 256 H800 GPUs further shows it effectively handles (manually injected) stragglers, improving end-to-end throughput by 1.58×.

1 Introduction

Large deep learning models have taken the industry by storm [1, 40, 42, 51, 52, 55, 62]. These large models boast unprecedented sizes, containing billions to trillions of parameters, and are trained over massive datasets in a large cluster. A typical training job often runs on tens of thousands of GPUs for weeks or even several months [1, 12, 62]. At this scale, failures become a norm rather than an exception. Therefore, developing runtime mechanisms that rapidly detect failures and efficiently tackle them is crucial to achieving high reliability in large model training.

Many of these mechanisms are developed to handle *fail-stop failures* that result in a complete halt of training [20, 26, 28, 33, 56], e.g., GPU hangs and runtime crashes. However, fail-stop alone does not cover the full spectrum of failure issues encountered in hyperscale training. Many system components, including CPUs, GPUs, and communication links, may still function but experience occasional performance degradation due to resource contention, thermal throttling, power supply, and network congestion. These failures, known as *fail-slows* or *stragglers*, do not cause a crash stoppage but significantly slow down the training progress [12, 22], as state-of-the-art large model training requires synchronization at each iteration boundary to achieve optimal model quality [50]. Despite their prevalence, fail-slow failures are hard to detect and have not been well studied. Although briefly mentioned in recent reports [12, 22], the overall characteristics of fail-slow failures in hyperscale training remain not well understood.

To shed light on this, in this paper, we first conduct a comprehensive characterization study (§3) in a *shared* production cluster comprising over 10,000 GPUs on 4,000 nodes interconnected through a RoCE network with up to 400 Gbps bandwidth. Our study reveals that fail-slows manifest as *transient failures* in both computation and communication. Specifically, *computation fail-slows* primarily result from CPU contention and GPU performance degradation due to thermal throttling or other issues (§3.2). These fail-slows occur occasionally ($< 2\%$) on individual nodes, with a short mean duration of 10 minutes. In comparison, *communication fail-slows* caused by network congestion on a communication link occur more frequently (40%) and last longer time, with a mean duration of 24 minutes. When it comes to at-scale distributed training, computation and communication fail-slows become even more prevalent, collectively causing more damage than that on small-scale training. We manually inspected large training jobs submitted in July 2024, each requiring at least 512 GPUs. Among all 27 large jobs, 16 experienced fail-slow failures, with a mean straggler duration of 72 minutes. These fail-slows delay the job completion time by an average of 1.34×.

Compared to fail-stops, fail-slows are more elusive to detect and locate [12, 22], especially when advanced *hybrid parallelism* techniques are utilized [35, 50], which combine tensor, data, pipeline, and possibly other parallelism to expedite training [13, 27]. Simply collecting hardware metrics like GPU SM utilization and RNIC's CNP (congestion notification packet) responses through telemetry offers little clue in this regard: given the synchronous nature of training, the presence of a straggler results in *simultaneous* utilization drops of *all* GPUs; likewise, a link could be shared by multiple jobs, and an RNIC's CNP response may not necessarily indicate a job-level performance issue for all. Although ad-

*Corresponding author.

vanced benchmarking tools are available [58, 65], using them to locate fail-slows requires stopping the entire training job and benchmarking all the involved GPUs and links, which is prohibitively expensive.

In this paper, we propose GREYHOUND, a system that rapidly identifies computation and communication fail-slows and reacts with effective mitigation mechanisms. GREYHOUND achieves this through two subsystems. The first subsystem, GREYHOUND-DETECT, employs a non-intrusive, framework-agnostic mechanism for fail-slow detection. It transparently tracks the training iteration time on each worker by hooking to NCCL [38] function calls and identifies prolonged iterations using the Bayesian Online Change-point Detection (BOCD) algorithm [2]. It then initiates lightweight profiling on each worker to obtain a fine-grained execution profile for each parallelization group, without interrupting the ongoing training job. By analyzing these execution profiles, it narrows the search space to a few *suspicious worker groups* where fail-slows may reside. To pinpoint their exact locations, GREYHOUND-DETECT briefly hangs the training job and runs benchmarking tests within suspicious groups to validate the GPU computation and link communication performance. Slow GPUs and links are then identified as straggling components. Notably, our detection system uniquely introduces online anomaly detection and micro-benchmarking to localize fail-slows at runtime, without disrupting training or relying on intrusive, full-job validation. This approach enables accurate, framework-agnostic diagnosis of both computation and communication stragglers at scale.

Once fail-slows are identified, GREYHOUND-MITIGATE takes over with an efficient mitigation mechanism. As fail-slows are usually transient, simply handling them as fail-stops using checkpoint-and-restart is an *overkill*. We explore the design space and identify four strategies to tackle fail-slows: (S1) ignoring them in the hope that they will soon be alleviated, (S2) resharding data parallelism to alleviate the load on slow GPUs, (S3) adjusting the parallelization topology to move congested links to light-traffic groups, and (S4) checkpoint-and-restart. As we move from S1 to S4, the mitigations become more effective, while the cost of action taking also increases. Therefore, the choice of optimal strategy depends on the duration (and severity) of the ongoing fail-slows, which cannot be known *a priori*. This problem resembles the classical *ski-rental problem* [23]. Drawing inspirations from its solution, we propose an effective online heuristic that starts with a low-cost strategy (S1) and progressively switches to a more effective, yet costly one if fail-slow persists and the current strategy proves ineffective. The mechanism falls back to job restart as a last resort. By explicitly modeling it as an online decision process and applying ski-rental theory, our system provides a principled solution for straggler mitigation. This approach systematically balances effectiveness and cost, which is not considered in previous solutions.

We have implemented GREYHOUND-DETECT as a framework-independent detection system and GREYHOUND-MITIGATE as a plugin for Megatron-LM [50]. We use GREYHOUND-DETECT as the primary tool in our characterization study to identify computation and communication fail-slows for 499 probing jobs submitted to the production cluster. Cross validation with human inspection shows that GREYHOUND-DETECT correctly diagnoses 498 jobs (99.8% accuracy), with less than 1% performance overhead. We further evaluate GREYHOUND-MITIGATE with manually injected fail-slows. It reduces the slowdown from computation fail-slows by up to 1.59× and from communication fail-slows by up to 1.23×. Experiments involving a training job on 256 H800 GPUs demonstrate that GREYHOUND accurately identifies the injected fail-slows and improves end-to-end throughput by 1.58×.

Our contributions are summarized as follows:

1. We present the first comprehensive characterization study in a production cluster to understand the overall characteristics and performance impacts of fail-slow failures in hyperscale large model training.
2. We propose GREYHOUND-DETECT, a non-intrusive, framework-agnostic detection system that identifies computation and communication fail-slows at runtime.
3. We propose GREYHOUND-MITIGATE, a system that effectively addresses fail-slow failures through a novel multi-level straggler mitigation mechanism.

2 Background

Parallelism for distributed training. Hyperscale training can require thousands of petaFLOP/s of compute power, necessitating the use of high-performance computing (HPC) clusters [12, 22, 35, 44]. These HPC clusters typically consist of tens of thousands of GPUs interconnected through high-speed fabrics such as InfiniBand [43] and RoCE [24]. To efficiently train large models on HPC clusters, various parallelism strategies have been developed to partition and distribute models across GPUs and nodes.

1) Tensor Parallelism (TP) is a technique that partitions the computation of specific operators, such as MatMul or Attention, along non-batch axes [25, 50, 63]. It enables parallel computation of each partition across multiple devices. TP often incurs significant communication costs due to the need for synchronization of each operator, and is often confined to a single node to minimize latency [22, 35].

2) Data Parallelism (DP) involves creating multiple model replicas and distributing them across multiple GPUs [35, 46, 50]. In each iteration, the global data batch is split into mini-batches, allowing each model replica to handle a portion of the data concurrently. After each iteration, the gradients from all replicas are synchronized. DP communication involves a *moderate* data transfer volume, which can occur either within a single node or across multiple nodes.

3) Pipeline parallelism (PP) partitions the model by placing different groups of layers, called *stages*, on separate

GPUs [19,35,63]. It further divides the mini-batch into micro-batches, allowing for pipelined forward and backward passes across different nodes. PP incurs the smallest communication overhead among all three strategies.

4) Hybrid parallelism. To maximize training efficiency, different parallelism strategies can be combined, allowing the model to be partitioned in multiple dimensions [12, 22, 35]. This technique, known as hybrid parallelism, have demonstrated the ability to train models with over a trillion parameters across thousands of GPUs [12, 22, 35].

Reliability issues. Given the complex nature of distributed training and the sheer scale of resources involved, large model training presents significant reliability challenges, manifested as crash stoppage (***fail-stop***) and still-functioning but slow stragglers (***fail-slow***). Both types of failures stem from software or hardware problems, and their impacts are magnified in large-scale setup: a single component experiencing fail-stop or fail-slow can crash or slow down the entire training process due to the frequent synchronization required in distributed training. Compared to fail-stops, fail-slow problems are hard to detect [12], necessitating sophisticated performance analysis tools [22, 58]. Despite brief reports from recent studies [12,22,58], the overall characteristics of fail-slows remains largely unknown, which motivates our study.

3 Characterization Study

In this section, we intend to answer the question, *how do fail-slows manifest in large model training*? We present a characterization study in a shared production cluster.

3.1 Cluster Setup and Methodology

Cluster setup. Our production cluster consists of over 4,000 nodes and more than 10,000 heterogeneous GPUs, including approximately 1,800 NVIDIA H800 GPUs and 2,600 A100 GPUs. These nodes are connected through a high-performance network employing the popular spine-leaf architecture [44]. The network offers up to $4 \times 200/400$ Gbps RoCE bandwidth for A100/H800 nodes. Within a node, GPUs are interconnected with NVLink/NVSwitch [36]. The cluster is a *shared, multi-tenant system* that runs diverse workloads, including: (1) large-scale model training jobs utilizing over 1,000 GPUs, (2) inference jobs encompassing both online inference and offline batch inference, (3) jobs for recommendation models, such as training embedding tables, and (4) short-running spot jobs for model debugging. Unless a job requests 8×GPUs per node, it may *colocate* with other jobs on the same node, e.g., embedding training collocated with inference. No GPU sharing [57] is allowed.

Methodology. As we are not allowed to instrument production workloads, we use two approaches to characterize fail-slows. (1) *Online probing with repeated sampling.* We repeatedly submitted identical small training jobs as *spot* workloads, which were *randomly scheduled* on available nodes across the cluster, often colocated with other production jobs. These

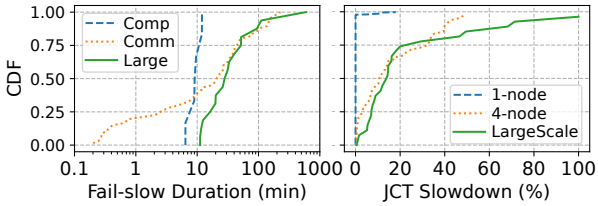


Figure 1: **Left**: CDF of fail-slow duration. **Right**: CDF of fail-slows' impact on job completion time (JCT).

Category	Online Probing		Offline Inspection
	1-Node	4-Node	At Scale ($\geq$ 512 GPUs)
No fail-slow	386	64	11
CPU Contention	4	1	0
GPU Degradation	2	0	0
Network Congestion	0	42	13
Multiple Issues	0	0	3
Total # Jobs	392	107	27
Avg. JCT Slowdown	11.79%	15.45%	34.59%

Table 1: Root causes and JCT slowdown of fail-slow issues in our characterization study.

training jobs are specially designed to act as *probes*, collecting key performance metrics to identify computation and communication fail-slows at runtime using techniques developed in §4. By submitting many of these probing jobs, we can cover a large number of nodes and links, effectively *sampling* a large portion of the cluster to characterize fail-slows at microscale (e.g., individual nodes and links). (2) *Offline inspection with collected traces.* To characterize fail-slows at macroscale, we collected a one-month trace containing numerous large training jobs, each utilizing at least 512 GPUs. We manually inspected the trace to identify fail-slows.[1]

3.2 How Do Comput. Fail-Slows Manifest?

Probing jobs. We start to characterize computation fail-slows occurred on individual nodes. We submitted 400 single-node training jobs to the cluster, of which 392 completed successfully without fail-stop errors. Each job trains a GPT2-11B model on one node using 4×H800 GPUs with a hybrid parallelism strategy of (2TP, 1DP, 2PP) to fully utilize GPU memory. The training framework used is Megatron-LM. Each job runs 10,000 iterations, taking 70 to 90 minutes. These probing jobs were scheduled to run on approximately 500 out of 1,800 H800 GPUs in our cluster (28% coverage).

Frequency and impacts. As summarized in Table 1, among 392 completed probing jobs, six experienced computation fail-slows, of which four are due to CPU contention and two are resulted from GPU performance degradation. Figure 1 (left) further shows the distribution of fail-slow duration. All computation fail-slows (blue curve) are short-lived, with an average duration of 10 minutes, extending the job completion time (JCT) by 11.79%. We next provide two case studies to better understand the root causes of these issues.

[1]We are working on open releasing the trace for public access.

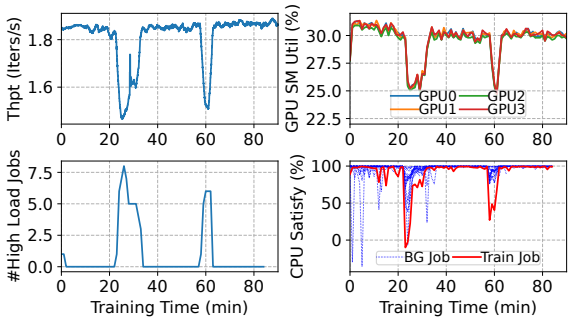


Figure 2: A case of a fail-slow job due to CPU contention. **Upper-left**: Training throughput. **Upper-right**: GPU SM utilization of the four GPUs used by this job. **Bottom-left**: The number of high-CPU jobs running on the same node. **Bottom-right**: CPU satisfaction rate of the training job (red) and other colocated background jobs (blue).

Case-1: CPU contention. As shown in Figure 2 (upper-left), the job under study experienced two fail-slows at 22 and 55 minutes, resulting in a maximum performance drop of 21.6%. Correspondingly, the job measured simultaneous declines in SM utilization across all four GPUs during fail-slow periods (upper-right), suggesting GPU slowdown. To validate this, we paused the job and conducted a matrix computation to assess GPU performance upon fail-slow detection, but found no performance degradation. Further investigation revealed a surge in the number of high-CPU jobs coinciding with the fail-slow occurrence (bottom-left), leading to a decreased CPU satisfaction rate (bottom-right), increased CPU time, and ultimately, a reduction in throughput.

Case-2: GPU performance degradation. Computation fail-slows can also be attributed to GPU performance degradation, often linked to thermal throttling. Figure 3 illustrates a case where the job under study experienced slowdown in the first 10 minutes (upper-left), during which *all four GPUs* measured low SM utilization (upper-right). Further investigation indicated that *only GPU0* was 20% slower than others (bottom-left) because of thermal throttling (bottom-right). Notably, rising temperatures do not always lead to performance issues; this may indicate a hardware problem, with an occurrence rate of about 0.5%, consistent with ByteDance's report [22].

Requirements for mitigation strategies. Given that compute fail-slows are typically transient—often resolving within tens of minutes—the overall impact on training performance is moderate. Thus, mitigation strategies must be lightweight and impose minimal overhead. Costly interventions are unsuitable, as their adjustment costs can exceed the potential performance gains, thereby negating the benefits of mitigation.

3.3 How Do Commun. Fail-Slows Manifest?

Probing jobs. To explore communication fail-slows, we submitted 120 multi-node probing jobs, of which 107 successfully completed without fail-stop. Each job utilizes 8×A100 GPUs across four nodes to train a GPT2-7B model with (2TP, 4DP, 1PP), where TP communications are over NVLink and DP communications are cross-node through a 400 Gbps RoCE link. Each job runs 10,000 iterations, taking approximately five hours. These probing jobs were distributed among 690 out of 2,600 A100 GPUs in our cluster (26.5% coverage).

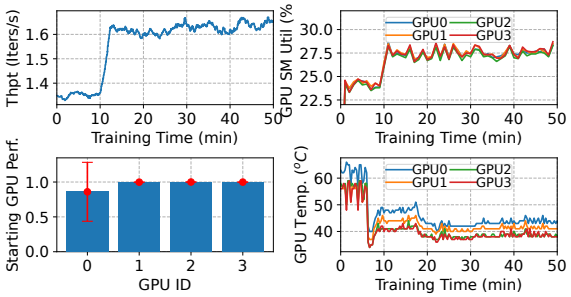


Figure 3: A case of a fail-slow job due to GPU performance degradation. **Upper-left**: Training throughput. **Upper-right**: GPU SM utilization of the four GPUs used by this job. **Bottom-left**: Normalized GPU performance during fail-slow. **Bottom-right**: NVML [39] reported GPU temperature.

Frequency and impacts. As summarized in Table 1, 43 out of 107 jobs experienced fail-slows. Among them, only one job was slowed due to CPU contention, while the other 42 encountered communication fail-slows caused by network congestion. We observed no hardware-induced fail-slows, such as issues with RNICs. Compared to computation slowdowns, communication fail-slows exhibit a wider range of duration, from sub-minutes to over 100 minutes(Figure 1, left). On average, communication fail-slow persists for about 24 minutes, delaying the average JCT by 15.45%.

Network congestion. Compared to computation slowdowns, network congestion emerges as a more significant factor contributing to performance degradation in multi-node training [12,45], with a notably higher frequency. Figure 4 presents a case study on a probing job that experienced two communication fail-slows at t=90 and t=265 minutes. The initial fail-slow resulted in throughput drop from 0.57 to 0.41 iterations/s; shortly thereafter, at t=265, the second slowdown further reduced throughput to merely 0.31 iterations/s (Figure 4, left). The two throughput drops align with the surges of congestion notification packets (CNPs) reported by the RNICs (center), indicating severe network congestion. Upon the onset of communication fail-slows, the SM utilization across *all eight GPUs* reduced simultaneously (right), despite healthy GPUs. We further investigated the spike in CNPs and discovered that other jobs were sharing the same link as our probing job, leading to throttled communication bandwidth.

Although this behavior is expected for network congestion control, such congestion is typically *unavoidable* and *unpredictable* in multi-tenant clusters, leading to reduced training performance [4, 45]. This is primarily due to the spine-leaf topology in modern clusters, where multiple jobs must share spine switches. As a result, isolating network traffic between

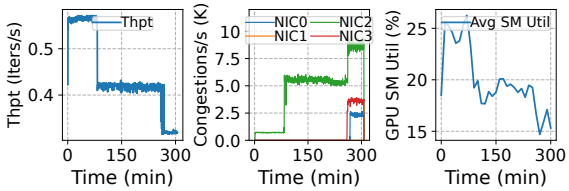


Figure 4: A case of fail-slow jobs caused by network congestion. **Left**: Training throughput. **Center**: The number of congestion notification packets (×1000) sent by RNICs. **Right**: Average GPU SM utilization of the 8 GPUs used by this job.

jobs remains difficult, even with advanced scheduling policies [4, 45] and network-level optimizations [44]—though improved isolation can help reduce the frequency of congestion events. Given its inevitability and impact, we classify such congestion-induced slowdowns as a type of fail-slow.

Requirements for mitigation strategies. Due to their long duration and substantial impact, communication fail-slows can cause severe performance degradation if left unaddressed or mitigated ineffectively. This justifies and motivates the use of more effective, aggressive—and potentially higher-overhead— strategies, as the benefits can ultimately outweigh the associated costs.

3.4 How Do Fail-Slows Manifest at Scale?

Limited by the small scale of each probing job, previous studies can only characterize fail-slows at microscale, i.e., on individual nodes or links. To characterize fail-slows at a larger scale, we collected and manually examined a one-month trace containing 27 large-scale training jobs that ran on our cluster in July 2024, each utilizing 512 to 1024 GPUs.

Frequency and impacts. Among 27 jobs, 16 encountered fail-slows, delaying the average JCT by 34.59%. In particular, 20% of these jobs were delayed more than 50% (green curve in Figure 1, right). The mean fail-slow duration is 72 minutes, significantly longer than that measured in the small sampling jobs (Figure 1, left). Table 1 summarizes the causes of the encountered fail-slows, where 13 slow jobs were due to network congestion, while the remaining were attributed to both network and GPU degradation. We observed no CPU contention for these jobs as they require 8×GPU per node and hence do not collocate with other workloads.

Deep dive. Figure 5 illustrates the throughput of two 1024-GPU jobs, one for LLM training and the other for MoE model training. Both jobs experienced severe network congestion, leading to considerable throughput fluctuations, one at the initial stage (left) and the other throughout training (right). Worse still, at this scale, a job may experience multiple fail-slows at certain times, causing more damage to training. Figure 6 illustrates a real case. Throughout the training process, the observed throughput closely aligns with the GPU SM utilization. The first severe network congestion occurred at t=62 minutes, slashing the training throughput by 80%. This degradation

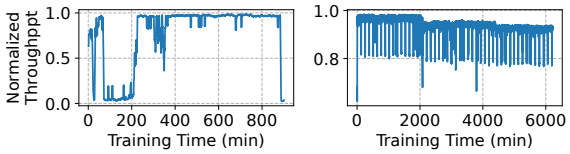


Figure 5: Two 1024-GPU jobs that failed slow due to network congestion. **Left**: An LLM training job. **Right**: An MoE training job with high variance and ladder-shaped fail-slow.

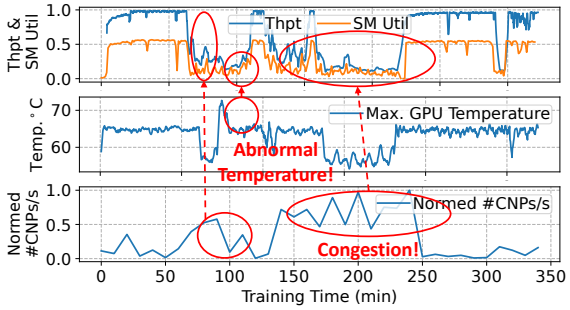


Figure 6: A 1024-GPU training job experiencing multiple performance issues, where fail-slow is caused by a compound of high GPU temperature and congested network. Y-axis of the top and bottom sub-figures are normalized.

was further exacerbated by a GPU thermal throttling event occurred at around t=80 while the network congestion remains unabated, further reducing the throughput to only 10% of the normal performance. Subsequently, from t=120 onward, another severe network congestion persisted for about two hours, cutting the throughput by 85% again. This case highlights the *compounding effects* of multiple performance issues in large-scale training, significantly undermining training efficiency.

Requirements for mitigation strategies. At large scale, training jobs are likely to be affected compound of compute and communication fail-slows. Therefore, mitigation strategies must be adaptive and flexible, able to dynamically handle these complex and compounded performance issues to maintain training efficiency.

3.5 Takeaways and Other Evidences

Our characterization study brings three takeaways:

Takeaway #1. Fail-slows are usually *transient*, primarily caused by degradation in computation and communication; the former typically stem from slow GPUs or CPU contention, while the latter are mainly due to network congestion.

Takeaway #2. Computation fail-slows tend to be short-lived and less frequent, leading to relatively minor performance degradation. In contrast, communication fail-slows due to network congestion are more common and tend to last longer, resulting in more significant training slowdowns.

Takeaway #3. As training scales up, the likelihood of simultaneously encountering multiple performance issues increases. The compounding effects of these issues can lead to significant training slowdowns, potentially exceeding 90%.

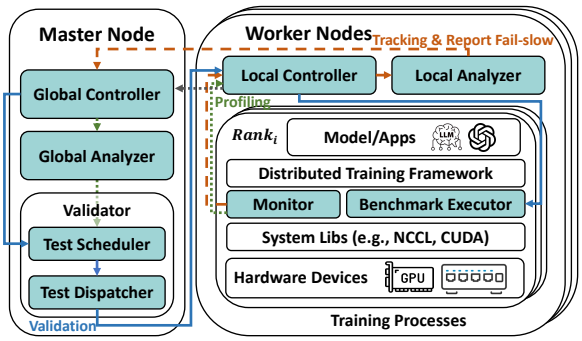


Figure 7: Architecture overview of GREYHOUND-DETECT.

Evidence from other companies. In addition to our study, ByteDance has reported computation fail-slows in its training platform [22]. Also, Meta's Llama team and Alibaba Cloud have reported communication fail-slows in LLM training [4, 12]. Our contacts with other companies bring attention to the similar fail-slow problems in large model training even on a *single-tenant* cluster of a hyperscale. While fail-slow issues are generally aware to practitioners, the general consensus, as noted in [12, 22], is that they are *hard to detect at scale*.

4 GREYHOUND-DETECT

Simply performing telemetry at cluster or node level, such as collecting GPU SM utilization and RNIC's CNP responses using standard tools like `nvidia-smi` or NVML [39], is insufficient to locate fail-slows. As discussed in §3, given the synchronous nature of training, any degraded component results in *simultaneous utilization drops* of *all* GPUs. Also, in a multi-tenant cluster, a network link is commonly shared by multiple jobs, and an increase in CNPs does not mean that all jobs are experiencing communication fail-slows, especially for those transferring light traffics over the link.

In this section, we design GREYHOUND-DETECT, a distributed monitoring system for large-scale training that accurately identifies performance issues in computation and communication at runtime. We have four design requirements.

R1: Non-intrusive and framework-independent. The detection system should not be bound to a specific training framework or require any modifications to the framework.

R2: Rapid and accurate. The system should rapidly identify the onset and resolution of fail-slow degradation while accurately locating the slow GPUs or communication links.

R3: Automated. The detection should be fully automated.

R4: Lightweight. The system should introduce minimal inspection overhead to training, without costly full-job validations that typically require checkpoints and restarts.

4.1 System Overview

GREYHOUND-DETECT is a distributed performance monitoring system deployed together with a large model training framework, such as DeepSpeed [46–48] or Megatron-LM [50]. Figure 7 provides an architecture overview, where components introduced by GREYHOUND-DETECT are highlighted in cyan. GREYHOUND-DETECT employs a master-worker architecture. On each worker node, multiple worker agents are co-deployed with the framework processes to monitor the training performance and report potential degradation to the master for further analysis and handling. Specifically, GREYHOUND-DETECT identifies fail-slows through a three-phase workflow: *tracking*, *profiling*, and *validation*.

1) Tracking. In this phase, each worker keeps track of the training iteration time for all training processes, called *ranks*, and detects slow iterations that indicate the onset of fail-slows. The worker reports these issues to the `GlobalController`, which transitions the system to the profiling phase.[2]

2) Profiling. During this phase, the `GlobalController` instructs each worker to collect the detailed execution profiles of the ongoing training job. These log profiles are sent to the `GlobalAnalyzer`, which identifies *suspicious* worker groups that may contain fail-slows. The `GlobalController` then transitions the system to the validation phase.

3) Validation. In this final phase, the system initiates fail-slow validations within the suspicious worker groups to precisely locate slow GPUs or congested network links.

We next describe the detailed designs in the three phases.

4.2 Tracking

GREYHOUND-DETECT enters the tracking phase upon the execution of a training job, continuously monitoring its performance on each worker node. Note that the monitoring is performed *per-job* on a multi-tenant cluster. To maintain transparency to the training framework (**R1**), GREYHOUND-DETECT inserts a *shim monitoring and benchmarking layer* between the framework and the underlying system libraries, such as NCCL and CUDA (Figure 7). In this shim layer, a `Monitor` intercepts communication operations (e.g., NCCL function calls) from the training framework and logs their *types* and *timestamps*. This is done by hooking to NCCL functions using Linux's `LD_PRELOAD` mechanism. Our system only intercepts top-level communication interfaces that are *uniform* in all communication libraries, such as `AllReduce` or `AllGather`, regardless of the underlying library implementation. Consequently, GREYHOUND-DETECT can seamlessly integrate with other communication libraries like ACCL [10], MSCCL [8], or customized NCCL as long as their top-level interfaces are consistent. The node's `LocalAnalyzer` then retrieves the communication call logs, maintained in shared memory, to infer the iteration time and detect slow iterations using two *time series analysis* techniques as follows.

Iteration time analysis. Throughout iterations, various collective communication functions, such as `ReduceScatter` (RS), `AllGather` (AG), and `AllReduce` (AR), are invoked periodically. Figure 8 illustrates an example, where a training

[2] If a transient fail-slow resolves during subsequent profiling or validation phase, our system identifies no anomaly and resumes normal training.

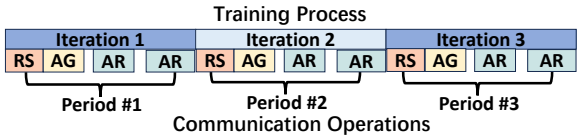


Figure 8: In iterative training, communication operations exhibit a clear periodic pattern, leading to recurring periods.

process exhibits a recurring period containing four communication calls. In practice, the number of communication calls involved in a recurring period and their patterns *vary* depending on the framework and the training model, which cannot be known due to the framework-agnostic requirement (**R1**).

To identify the recurring period from a call sequence, we employ a time series analysis approach based on autocorrelation function (ACF) [5]. Formally, given a call sequence $X = \{x_1, x_2, \ldots\}$, let X_t be a subsequence of X containing L elements starting from x_t. Let k be the *lag*, ranging from 1 to a predefined maximum. We evaluate the likelihood of k being the recurring period of X by calculating the corresponding ACF defined as follows:

$$ACF(X)_k = \frac{Cov(X_t, X_{t+k})}{Var(X_t)} = \frac{\sum_{t=1}^{L-k}(X_t-\mu)(X_{t+k}-\mu)}{\sum_{t=1}^{L}(X_t-\mu)^2},$$

where μ is the mean of X. A higher value of $ACF(X)_k$ indicates a greater likelihood that k is a recurring period. Thus, we can determine the recurring period of X by identifying the first k for which $ACF(X)_k$ exceeds a certain threshold M (set to 0.95 in our experiments), i.e.,

$$\text{Period} = \arg\min_k (ACF(X)_k \geq M).$$

Once the recurring period is identified, the iteration time derives as the time difference between a communication operation and its occurrence in the previous period.

Slow iteration detection. Training performance inherently fluctuates due to factors like periodic Python garbage collection, PyTorch CUDA memory allocator cache misses, and uneven sequence lengths in the dataset [59]. Thus, a robust statistical method is required to reliably identify true performance degradations while filtering out normal fluctuations (**R2**). To achieve this at runtime, we propose to use the Bayesian online change-point detection (BOCD) algorithm [2] followed by a verification checking to differentiate between real fail-slow issues and normal performance jitters.

1) The BOCD algorithm is an efficient time series algorithm that finds *change-points* online in a dynamic sequence in *linear* time. Feeding the algorithm the iteration time sequence, the identified change points usually correspond to the iterations with its duration change of over 10% due to the onset or relief of fail-slows. Specifically, the algorithm defines a run-length r_t for each timestamp t, which equals to $r_{t-1}+1$ if t is not a change point, otherwise it is reset to 0. It then applies Bayesian inference to calculate the likelihood of $r_t = 0$ for each timestamp. If the likelihood exceeds a certain threshold (0.9 in our experiments), it reports t as a change-point.

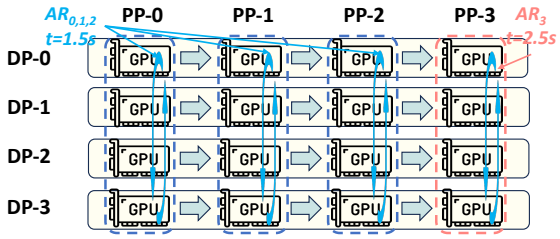


Figure 9: An example of *cross-group comparison*, $AR_{0,1,2,3}$ form a *comparable cluster* and AR_3 is a degraded group.

2) Change-point verification. While the BOCD algorithm identifies potential change-points, applying it directly to fail-slow detection results in numerous false positives, as it misclassifies normal performance jitters as fail-slow incidents. To improve the detection accuracy (**R2**), we propose an additional verification step that compares the average iteration time before and after each identified change-point, treating it as a jitter if the difference is less than 10%.

The ACF-based iteration time analysis, combined with BOCD plus change-point verification, detects slow iterations reliably and in linear time, thereby meeting requirement **R2**. Once slow iterations are detected, the `LocalAnalyzer` reports to the master for further analysis and handling.

4.3 Profiling and Validation

Profiling. The detection of a slow iteration is a clear indicator of stragglers, which must be located rapidly. To avoid benchmarking all GPUs and links (**R4**), which is costly, GREYHOUND-DETECT narrows the search to a few suspicious worker groups. This is achieved through lightweight profiling. On each worker node, the GPU `Monitor` injects CUDA events into intercepted NCCL calls to measure the execution time of each communication group. The measurement results are then collected by the master's `GlobalAnalyzer` to identify the degraded groups via *cross-group comparison*. The basic idea is to categorize communication profiles into *comparable clusters*, where each communication group (e.g., `NCCLGroup`) within a cluster handles an identical data transfer volume. Ideally, communication times within the same comparable cluster should be consistent. However, stragglers manifest as groups with prolonged execution times due to performance degradation. For instance, as illustrated in Figure 9, the four DP all-reduce groups $AR_{0,1,2,3}$ have identical communication volumes under uniform PP stage division, forming a 4-group comparable cluster. Nevertheless, AR_3 measures a markedly longer execution time compared to the other groups, indicating the presence of a straggler within it. In practice, we identify a group as degraded if its execution time exceeds the median by over 10%.

Validation. Once a suspicious group is identified, further validation is needed to locate the degraded components in it. As this requires running benchmark tests, the training job must be temporarily suspended. To avoid expensive checkpoint-

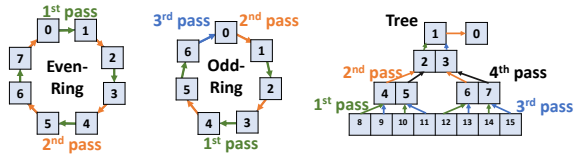


Figure 10: $O(1)$ validation of Ring and Tree communicators. Each cell is a rank, and the lines represent network links.

and-restart, we devise a lightweight training suspension mechanism (**R4**). Since GPU `Monitor` hooks NCCL calls, it can *hang* the training by simultaneously "trapping" those calls into a wait loop and give control back to the hanged training process once validation is done, *without* costly restarting the process or re-initialization.

While the training is being hanged, computation and communication benchmark tests are dispatched automatically to the identified suspicious worker groups to precisely locate the degraded components (**R2** and **R3**).

1) Computation validation. To benchmark computation performance within a group, the master's `TestDispatcher` dispatches standard GEMM [37] tests to all worker GPUs in parallel to identify slow stragglers, if any. We use GEMM as it is a building block training operation and is commonly used as a standard test in advanced benchmarking tools like SuperBench [58]. Customized tests are also supported.

2) Communication validation. Exhaustive evaluation of all possible links within a communication group is time-consuming ($O(N^2)$, where N is the group size). As collective communications can be performed with *ring* or *tree* topologies, we propose to divide the collective topology into non-overlapping peer-to-peer (P2P) operations and evaluate only the links used in training. These operations can be executed efficiently in $O(1)$ time, regardless of the group size (**R2**), as illustrated in Figure 10. Specifically, for a *ring* topology, the algorithm differentiates between even- and odd-rank rings. It divides even-rank rings into P2P send-receive operations that can be covered in *two* passes. In the first pass, data is transferred from even to adjacent odd ranks *simultaneously* (i.e., $0 \rightarrow 1, 2 \rightarrow 3, \ldots$); the second pass sends data from odd ranks to adjacent even ranks simultaneously (i.e., $1 \rightarrow 2, 3 \rightarrow 4, \ldots$, as shown in Figure 10, left). For odd-rank rings, an additional pass is needed to accommodate the remaining link (Figure 10, center). For *tree* topology, the validation requires *four* passes (Figure 10, right). The first pass simultaneously sends from left-child ranks at even levels to their parents, followed by the second pass which sends from right-child ranks at even levels, also simultaneously. The third and fourth passes reverse the roles of the senders, starting from odd levels. Since the transmission sizes are identical, slows link measures longer communication times and can be easily identified. In our implementation, both ring- and tree-topology validation are needed in a worker group as NCCL determines the actual collective communication topology dynamically.

Limitation. Our current design cannot detect fail-slows that occur only when computation and communication kernels co-execute in specific patterns [58]. However, these instances are typically linked to defects in particular hardware batches and are extremely rare in production environments.

5 GREYHOUND-MITIGATE

Unlike fail-slow detection, which is non-intrusive, fail-slow mitigation requires supports from the training framework. Therefore, it cannot be entirely non-intrusive to the training job. In this section, we present GREYHOUND-MITIGATE, a system that effectively addresses fail-slows with a novel adaptive multi-level mitigation mechanism.

5.1 Adaptive Multi-Level Mitigation

Solution space. Simply treating transient fail-slows as fail-stops by means of checkpoint-and-restart can do more harm than good, as dumping and restoring checkpoints for large models is time-consuming. In fact, dumping a GPT2-100B model takes nearly 100 minutes [56], even longer than the mean fail-slow duration in our cluster (§3). We explore the solution space and identify four strategies.

(S1) Do nothing. This approach simply ignores fail-slow problems in the hope that the straggler components will soon be self-recovered. Many existing systems choose to do so due to the lack of an effective detection tool.

(S2) Adjust micro-batch distribution. This strategy is efficient in addressing computation fail-slows, which result in *uneven processing speed* among model replicas (i.e., DP groups). The strategy reacts by redistributing micro-batches across DP groups based on their processing speed, alleviating the load on slow GPUs and rebalancing the computation (§5.2).

(S3) Adjust parallelism topology. This strategy effectively mitigates *both computation and communication stragglers* by: 1) reassigning heavy-traffic communications to less congested links, thereby mitigating network congestion; and 2) consolidating multiple stragglers into the minimal number of PP stages, thus reducing their overall impact (§5.2).

(S4) Checkpoint-and-restart. As a last resort, the system performs checkpointing and restarts training on healthy nodes. While this approach effectively eliminates all fail-slows by replacing slow components, it incurs the highest overhead and may require significant human intervention.

We compare the four strategies in Table 2.[3] As we move from S1 to S4, the mitigation effectiveness improves, but the action overhead also increases. Therefore, the optimal strategy *varies* depending on the severity and the duration of fail-slows. While the severity can be measured, fail-slow duration exhibits a large dynamic range, from tens of seconds to several hours (Figure 1, left), and cannot be predicted accurately.

Ski-rental-like multi-level straggler mitigation. We find that the mitigation planning problem resembles the classical

[3] Adapting TP is ineffective for mitigating fail-slow, as TP operates within a single node, which is not susceptible to communication fail-slow.

Strategy	Effectiveness		Action Overhead
	Slow Comp.	Slow Comm.	
S1: Ignore	No Effect	No Effect	None
S2: Adjust Microbatch	Mitigate	No Effect	Low
S3: Adjust Topology	Mitigate	Mitigate	Medium
S4: Ckpt-N-Restart	Eliminate	Eliminate	High

Table 2: Comparison of mitigation strategies in terms of effectiveness and overhead.

Algorithm 1 Adaptive Multi-level Fail-Slow Mitigation

```
1:  function MitigationPlanner(event)
    Input: The fail-slow event to handle.
2:      ▷ Find available strategies to mitigate this event.
3:      candidates ← FindStrategies(event.root_cause)
4:      ▷ Sort the strategies by their overhead.
5:      candidates.sort(key=strategy.overhead)
6:      id ← 0    ▷ Current mitigation strategy ID.
7:      while event.persist() do
8:          ▷ Get number of iterations that fails slow.
9:          slow_iters ← event.get_slow_iters()
10:         ▷ Calculate the impact of fail-slow.
11:         slow_impact ← slow_iters * (t_slow − t_healthy)
12:         ▷ Apply the current strategy and move forward.
13:         if slow_impact ≥ candidates[id].overhead then
14:             candidate_strategies[id].apply()
15:             id ← id + 1
16:         end if
17:     end while
18: end function
```

ski-rental problem [23], which involves balancing recurring ski-rental costs (akin to experiencing fail-slows) against a one-off ski-buying investment to avoid those costs (akin to taking mitigation action), all without prior knowledge of duration. Inspired by the classical ski-rental algorithm, we design an *adaptive multi-level* fail-slow mitigation mechanism. It begins with a low-cost strategy (**S1**) and progressively switches to more effective—and hence more costly—strategies (**S2** to **S4**) if fail-slow persists and the current approach proves ineffective. To determine when to switch strategy, the algorithm tracks the number of iterations affected by fail-slow and the resulting slowdowns to calculate an *accumulated performance impact*. It switches to the next strategy when the cumulative slowdown *equals* the action overhead of that strategy; that is, the algorithm is better off taking that more aggressive strategy upon the detection of fail-slow should it have known its performance impact (i.e., buy a ski when the cumulative rental cost *equals* the one-off buy cost). Algorithm 1 formally describes this mechanism.

5.2 Micro-batch and Parallelism Adjustment

We now describe the detailed design of the four strategies employed in the multi-level mitigation scheme. Since **S1** and **S4** are straightforward, we focus specifically on the two parallelism adjustment strategies **S2** and **S3**.

S2: Adjust micro-batch distribution. This strategy dynamically adjusts the number of micro-batches allocated to DP groups according to their computation performance, effectively mitigating *computation* fail-slows at a low cost. Specifically, DP shards a large global batch into multiple micro-batches and distributes them evenly among all groups (i.e., model replicas) at initial. When a certain group experiences computation fail-slow, we rebalance the workload by accordingly reducing the number of micro-batches allocated to it.

Formally, let D be the number of DP groups and M be the number of micro-batches in a global batch, where group DP_i is allocated m_i micro-batches. The processing time for a micro-batch in DP_i is denoted as t_i, which is profiled by GREYHOUND-DETECT (§4.3). Our goal is to minimize the processing time of the slowest DP group, which can be formulated as a quadratic programming problem that minimizes the variance in processing times across all DP groups:[4]

$$\min \max_{i=1,\ldots,D} m_i t_i \Leftrightarrow \min \sum_{i=1}^{D} (m_i t_i - \frac{\sum_{i=1}^{D} m_i t_i}{D})^2, \quad \text{Subject to} \quad m_i \in \mathbb{N}^+ \text{ and } \sum_{i=1}^{D} m_i = M. \tag{1}$$

Although the micro-batches are not evenly distributed after this adjustment, the training loss can remain consistent using a weighted gradient aggregation method [6]. This leads to the same training result and hence ensures the correctness. Moreover, this adjustment only modifies the number of micro-batches (m_i) assigned to each DP group, both the global batch size and micro-batch sizes remain unchanged. Given the peak GPU memory usage in memory-efficient pipelines such as 1F1B [34] is independent of m_i, this strategy does not increase memory footprint during training.

Implementation and overhead. Our implementation uses `cvxpy` [9] to solve Equation (1), which typically takes only a few seconds (detailed in Table 5). Before each iteration, each DP group retrieves its assigned m_i from the global controller, allowing the new distribution to take effect immediately in the subsequent iteration, without requiring a job restart.

S3: Adjust parallelism topology. We design this strategy as a *reactive* method to handle communication stragglers. It adjusts the parallelism topology to reduce congestion and minimize PP stages affected by stragglers, more effectively mitigating communication fail-slows with moderate overheads.

Reassign congested links to light-traffic groups. One key characteristic in hybrid-parallel training is that PP communication involves significantly less data transfer than DP [35,50]. Specifically, per-GPU PP traffic is $O(m_i \times \text{activation_size})$, typically in tens to hundreds of MBs per iteration, while DP gradient synchronization can exceed tens of GBs. Therefore, DP groups are much more susceptible to network congestion. To mitigate fail-slow events caused by congestion, we can reassign congested links from heavy-traffic DP groups to light-traffic PP groups, reducing the overall performance impact. For example, as shown in Figure 11, suppose the link between nodes 3 and 4 is congested and originally used for DP communication. By exchanging the DP and PP roles for nodes 2 and 3, we can redirect the traffic from node 3 to node 4 into light-traffic PP communication, effectively alleviating

[4] For a 1F1B pipeline, we add another constraint: $m_i \equiv 0 \bmod PP$.

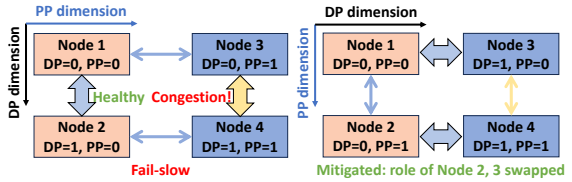


Figure 11: Topology adjustment to mitigate network congestion. After swapping Nodes 2 and 3, the congested link shifts from a heavy-traffic DP group to a lighter-traffic PP group.

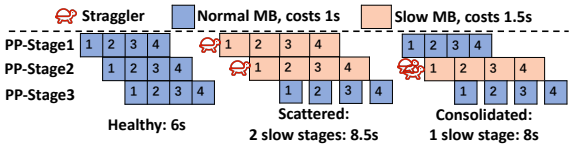


Figure 12: The number of straggling PP stages determines iteration time. With two stragglers are scattered across two stages, the execution time is prolonged to 8.5s, while it can be mitigated to 8s by consolidating two stragglers in one stage.

the impact of network congestion.

Straggler consolidation. When multiple stragglers are present, consolidating them into one PP stage mitigates slowdown. Since workers within the same PP stage operates synchronously, the performance is determined by the slowest straggler, irrespective of the number of stragglers within this stage. In contrast, as shown in Figure 12, having stragglers scatter across multiple PP stages is sub-optimal. Therefore, in case of multiple stragglers, our topology adjustment aims to consolidate them into the minimal PP stages. To achieve this, we calculate the minimal number of PP stages needed to contain stragglers by $\lceil$#Stragglers/# GPUs per PP stage$\rceil$ and consolidate the stragglers accordingly. We also prefer to shift them to interior stages, as the first and last stages typically endure a higher load due to the pre- and post-processing modules (e.g., embedding layers) allocated to them.

Implementation and overhead. Topology adjustment can be implemented in four steps: 1) pausing the ongoing training, 2) temporally dumping parameters to swap into main memory, 3) swapping parameters via peer-to-peer RDMA, and 4) restarting the training process. This adjustment incurs moderate overhead, only determined by the number of parameters per GPU, irrespective of the training scale given that each GPU operates independently. This adjustment is typically done in about one minute (detailed in Figure 19).

6 Implementation

We have implemented GREYHOUND-DETECT in approximately 5.5k lines of code (LOC) of C++ and Python. The worker's GPU `Monitor` hooks NCCL functions using Linux's `LD_PRELOAD` mechanism. Intra-node and inter-node communications between modules are achieved through shared memory and Redis [49], respectively. During validation, the `BenchmarkExecutor` reuses the same CUDA context and NCCL communicators from training, eliminating initialization overhead. For compute benchmarking, we use GEMM kernels with various data types, as GEMM is the primary building block in training and a common benchmarking standard. Specifically, `BenchmarkExecutor` sequentially run FP8, FP16, and FP32 GEMM kernels, with each kernel execution monopolizing all SMs to accurately assess the performance. For communication benchmarking, we perform concurrent NCCL send/receive operations shown in Figure 10, using message sizes of 16, 32, and 64 MB. Each computation and communication test is repeated three times, with the mean execution time used as the final metric. The benchmarking component is designed in a modular fashion, which facilitates the easy integration of additional benchmarks in the future.

GREYHOUND-MITIGATE is implemented in 1.5k LOC of Python. The planner receives straggler IDs from Redis and generates adjustment strategies, which are then executed a lightweight plugin of Megatron-LM [50].

7 Evaluation

In this section, we evaluate GREYHOUND-DETECT and GREYHOUND-MITIGATE to answer the following questions:

1. How accurately does GREYHOUND-DETECT estimate iteration time and identify fail-slow incidents across various models and parallelism configurations? (§7.2)
2. Is GREYHOUND-MITIGATE effective in alleviating various fail-slow failures with different root causes and parallelism configurations? (§7.3)
3. What is the overhead associated with GREYHOUND-DETECT and the various mitigation strategies implemented in GREYHOUND-MITIGATE? (§7.4)
4. How effective is GREYHOUND in enhancing training efficiency and mitigating the impact of fail-slow incidents in large-scale real-world training scenarios? (§7.5)

7.1 Experiment Setup

Testbed configuration. We conduct our evaluation on a high-performance cluster comprising 55 nodes, each equipped with 8 NVIDIA H800 GPUs connected via NVSwitch. The nodes are interconnected through a 400Gbps InfiniBand network in a spine-leaf topology, ensuring symmetric inter-node bandwidth. Our tests utilize Megatron-LM [50], a large-scale distributed training framework built on PyTorch [3], to train a set of GPT-2 models in various sizes and parallel strategies. The testbed runs CUDA version 12.2 and NCCL version 2.18.1.

Fail-slow injection. We evaluate the effectiveness of our mitigation system using deterministic manually injected fail-slows. To simulate computational fail-slows, we employ `nvidia-smi` to lock the GPU SM frequency. To inject communication fail-slows, we initiate side-channel communication jobs that create network bandwidth contention, thereby reducing the available bandwidths on specific network links.

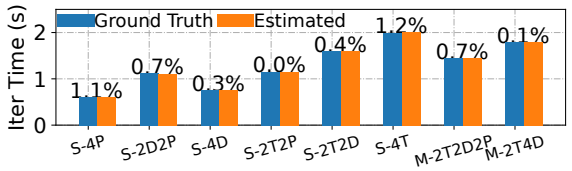


Figure 13: Accuracy of iteration time estimation in single-node (*S*) and multi-node (*M*) settings. The notation *xTyDzP* specifies the TP size *x*, DP size *y*, and PP size *z*.

Algorithm	Accuracy↑ (%)	FPR↓ (%)	FNR↓ (%)
SlideWindow	99.5(390/392)	**0.0(0/386)**	25.0(2/8)
BOCD	77.8(305/392)	18.39(87/473)	**0.0(0/6)**
BOCD+V	**100.0(392/392)**	**0.0(0/386)**	**0.0(0/6)**

Table 3: Detection evaluation for computation fail-slows.

7.2 How Accurate Is Detection?

Iteration time estimation. We first evaluate the accuracy of the ACF-based iteration time estimation across various hybrid-parallel strategies as it is the foundation of fail-slow detection. We deploy GPT2-7B training jobs using different parallelism strategies on 1, 2, and 4 nodes. As illustrated in Figure 13, in single-node experiments with 4 GPUs, the relative error remains below 1.2% compared to the ground truth iteration time, regardless of the parallel strategies employed. In a 2-node experiment with a (2TP, 2DP, 2PP) configuration, the error is 0.7%, while in a 4-node test with a (2TP, 4DP) setup, it remains highly accurate at just 0.1% relative error.

Fail-slow detection. We then assess the effectiveness of our BOCD and verification algorithm (BOCD+V) in detecting computation and communication fail-slows. The baseline methods are sliding window and classical BOCD; the former reports a fail-slow if there's a >10% performance change in the sliding window from the current median, while the latter performs no verification. Using traces from §3, we assess their accuracy against human-labeled ground truth. As shown in Table 3, BOCD+V achieves perfect 100% accuracy with 0% False-Positive Rate (FPR) and 0% False-Negative Rate (FNR) in detecting computation fail-slow. In case of communication fail-slow (as illustrated in Table 4), BOCD+V attains 99.1% accuracy, 0% FPR, and only 2.3% FNR. The FNR primarily results from a rare case containing consecutive <10% degradations. The original BOCD has a lower FNR by reporting all suspicious change-points but suffers from a high FPR. Similarly, the sliding window method is less accurate, missing many fail-slow cases with a higher FNR.

7.3 How Effective Is Mitigation?

In this section, we assess the effectiveness of the mitigation strategies introduced in § 5.2. Since GREYHOUND does not have prior knowledge of stragglers, they must be detected online during training. As a result, these experiments also serve as an implicit end-to-end test of GREYHOUND-DETECT, which achieves 100% accuracy in all cases reported below.

Algorithm	Accuracy↑ (%)	FPR↓ (%)	FNR↓ (%)
SlidingWindow	93.5(100/107)	1.5(1/65)	12.2(6/49)
BOCD	69.2(74/107)	34.0(33/97)	***0.00(0/43)***
BOCD+V	***99.1(106/107)***	***0.00(0/64)***	2.3(1/44)

Table 4: Detection evaluation for communication fail-slows.

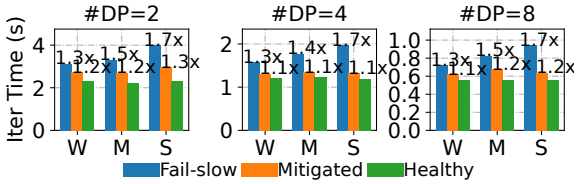


Figure 14: Effectiveness of micro-batch adjustment strategy of mitigating various fail-slow severities and DP settings.

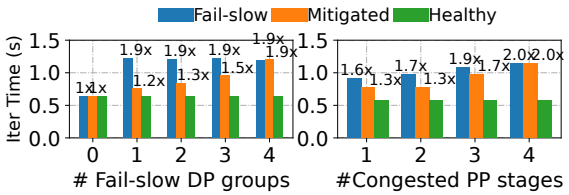


Figure 15: **Left**: Effectiveness of micro-batch adjustment (**S2**) on various number of slow DP groups. **Right**: Effectiveness of straggler consolidation (**S3**) on various slow PP stages.

Micro-batch distribution adjustment (S2). To evaluate the effectiveness of strategy **S2** in mitigating computation fail-slows, we deploy a single-node training with 8 GPUs. We inject weak (W), medium (M), and severe (S) computation fail-slows to training jobs with 2, 4, and 8 DP groups, as illustrated in Figure 14. Our approach reduces the average iteration time from 1.7× the baseline to 1.3/1.1/1.2×, yielding up to 1.52× optimization. This strategy proves effective across various setups and fail-slow severity since it consistently ensures a dynamic load balance across all DP groups.

As shown in Figure 15 (left), we also evaluate **S2** when multiple DP groups experience fail-slow. In a 4-DP training job, we inject medium slow computation into 0 to 4 DP groups. **S2** achieves its best performance with only one slow group, reducing iteration time from 1.31s to 0.83s, yielding 1.59× improvement. While multiple slow DP groups do not further increase iteration time, the room for mitigation decreases as the number of degraded DP groups rises. This is because with multiple DP groups degraded, total computational power decreases, limiting adjustment flexibility, and there is no room for adjustment if all groups are slow.

Topology adjustment (S3). We evaluate strategy **S3** in a 2-node experiment with 16 GPUs. As shown in Figure 16, we inject communication fail-slows into training jobs with 4 or 8 PP stages. The results reveal a reduction in the average iteration time by up to 1.23× for PP=4, and 1.14× for PP=8, both under severe congestion. The strategy is more effective with 4-stage PP due to the increased bubble rate and longer idle times associated with the longer pipeline in the 8-stage

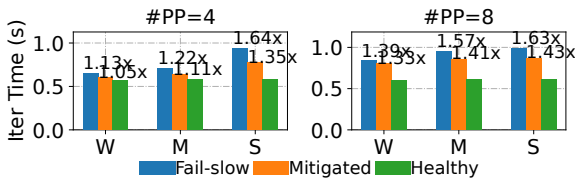


Figure 16: Effectiveness of topology adjustment strategy of mitigating various fail-slow severity and PP settings.

setup, which ultimately understates the effectiveness.

To evaluate the effectiveness of straggler consolidation in topology adjustment, we conduct an experiment with 16 GPUs using (4DP, 4PP) setup. As shown in Figure 15 right, congestion in one or two links raises iteration time to 1.6/1.7×, which can be both mitigated to 1.3× through consolidating them into only one PP stage. With three congested links affecting 6 GPUs, mitigation reduces iteration time from 1.9× to 1.7×, since one stage contains only four GPUs and six stragglers must occur across two PP stages. If all links are slow, there is no room for adjustment.

7.4 How Large Is the Overhead?

Overhead of GREYHOUND-DETECT. As discussed in §4, GREYHOUND-DETECT comprises a three-phase workflow: *tracking, profiling, and validation*, with corresponding overhead evaluated as follows.

Tracking. We conducted training under the same settings as in §7.2. As shown in Figure 18, the average overhead is only 0.39%, with a maximum of 1.1% compared to training without the detector. In some cases, enabling the detector results in an even slightly faster iteration (indicated by 0.0% in green), highlighting that such overhead is caused by inherent fluctuations instead of detector itself. These results prove that the overhead of GREYHOUND-DETECT is negligible. Upon fail-slow events, the BOCD algorithm can detect it in the next 2-3 iterations, costing typically <5 seconds for reaction.

Profiling. Since this phase only collects profiles for offline analysis, it does not introduce any additional overhead.

Validation. The validation phase runs the compute and communication benchmarks described in §6. As shown in Figure 17, compute validation completes within 0.59/0.51/0.24s on NVIDIA A10, A100, and H800 GPUs under normal conditions, and within 1.94/1.23/0.87s under fail-slow scenarios (GPU clock rate locked at 300 MHz). For communication validation, we assess NVLink, PCIe, and RDMA, where RDMA is most susceptible to congestion, but all benchmarks (3 repetitions) still finish within 3.04 seconds. Overall, the total benchmarking overhead remains about 5 seconds even under severe fail-slow scenarios or on lower-end GPUs.

Overhead of GREYHOUND-MITIGATE. We then evaluate the overhead of the strategies in GREYHOUND-MITIGATE.

*Micro-batch adjustment (**S2**).* We evaluate the overhead for adjusting the micro-batch distribution, which primarily arises from solving Equation (1). As shown in Table 5, although

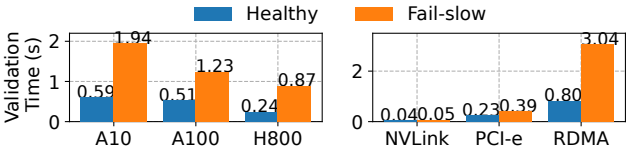


Figure 17: **Left**: overhead of executing compute benchmarks. **Right**: overhead of executing communication benchmarks.

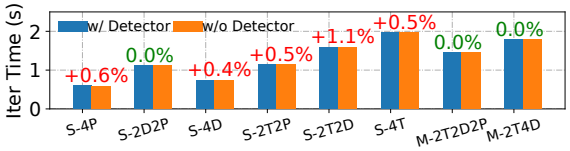


Figure 18: Overhead introduced by GREYHOUND-DETECT across various parallel strategies.

this overhead increases exponentially with the number of DP groups, it remains around 30 seconds even with 512 DP groups, showing its efficiency for hyperscale training.

*Topology adjustment (**S3**).* We evaluate the topology adjustment overhead under various parameter settings. We set the per GPU memory footprint to dump to 10, 22, 38, and 76GB, and conduct training on a single node with 8 GPUs. The adjustment time will no more increase as training scales up, since each node operates independently. As shown in Figure 19, our approach reduces pause time by up to 6.72× compared to the disk-based C/R, primarily by eliminating checkpoint dumping and loading times. The performance gains are more pronounced with more parameters, as the disk operation times increase significantly for large I/O sizes.

7.5 How Does GREYHOUND Perform at Scale?

To evaluate GREYHOUND 's effectiveness at scale, we conduct a GPT2-40B training experiment on 256 NVIDIA H800 GPUs with (8TP, 16DP, 2PP). We manually inject 12 fail-slow events, including two communication congestion events and ten compute slowdowns, each with varying severity levels, as detailed in Table 6. We also simulate compound cases where compute and communication fail-slows occur simultaneously, mirroring the scenarios presented in our characterization (§3.4). The training job is executed twice under identical straggler injection schedule: once with GREYHOUND and once without for A/B testing. Note that the straggler schedule is unknown to GREYHOUND—all fail-slows must be detected first and then mitigated, yielding an end-to-end evaluation.

Detection. First, we evaluate GREYHOUND-DETECT 's detection accuracy and reaction time. As presented in Table 6, it accurately identifies all 12 fail-slow events, achieving a 100% detection rate. For each fail-slow event, we measure the time difference between when GREYHOUND-DETECT

# DPs	16	32	64	128	256	512
Time(s)	0.01	0.01	0.01	0.11	6.78	35.93

Table 5: Time to find the optimal micro-batch distribution.

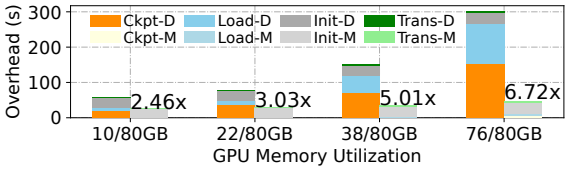


Figure 19: Overhead breakdown of **S3**. *M*: memory-based dump and load (our method), *D*: disk-based C/R.

Comm. Slow	Comp. Slow	Detect Acc.	Avg. Reaction
Medium: 2	*Weak*: 3 *Medium*: 5 *Severe*: 2	100%	*BOCD*: 3.7s *Prof.&Val.*: 6.86s *Total*: 10.56s

Table 6: Detection accuracy and average reaction time of GREYHOUND-DETECT in the 256 H800 GPU experiment.

reports it and the actual injection timestamp, representing the reaction time. The average reaction time is 10.56 seconds in this 256-GPU experiment, indicating low detection overhead at scale. This reaction time consists of two phases: the BOCD algorithm signals a change point, occurring approximately 4 seconds post straggler injection. Subsequently, GREYHOUND-DETECT enters profiling and validation mode, requiring an additional 6 to 10 seconds to identify the root cause.

Mitigation. Accurate and timely detection enables effective mitigation strategies. As shown at the top of Figure 20, without GREYHOUND, computation stragglers cause significant throughput declines. In contrast, with GREYHOUND, throughput quickly recovers to near-optimal levels, demonstrating the effectiveness of micro-batch adjustment (**S2**). During communication congestion periods, GREYHOUND initiates brief pause for topology adjustments (**S3**) at t=850 and t=2100, each lasting under one minute. These adjustments are faster than traditional C/R methods which takes tens of minutes. Once congestion ends, the topology is promptly reverted (at t=1150 and t=2350). Notably, compound fail-slows can reduce training throughput by about 55%, but it can be mitigated to only 25% degradation with GREYHOUND.

End-to-end performance. Combining accurate detection and effective mitigation, GREYHOUND significantly reduces the impact of stragglers. Without GREYHOUND, end-to-end throughput decreases from 37.4 to 18.9 iterations per minute due to injected fail-slow events. In contrast, with GREYHOUND, throughput recovers to 29.8 iterations per minute (including the detection and validation overhead), achieving 1.58× improvement in end-to-end throughput.

8 Related Work

Reliability issues in training. Several studies address the fail-stop issue using checkpoints [28, 33, 56], re-computation [53], and elastic frameworks [11, 20, 53, 56, 64]. For example, Oobleck [20] pre-computes *static* parallelism adjustments for predictable resource changes (e.g., node removal). In contrast, while fail-slow have been acknowledged in various reports [12, 22], only a concurrent work, Holmes [59], addresses

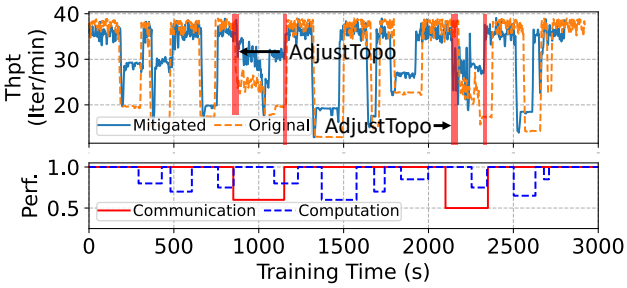


Figure 20: Evaluation of GREYHOUND for a 256 H800 GPU training with computation and communication fail-slows.

this issue but lacks mitigation. However, fail-slow mitigation is orthogonal to fail-stop recovery and more challenging as it requires *dynamic* reactions according to the detected stragglers without altering the number of available resources.

Fail-slow in other fields. Fail-slow also exists in cloud services [7, 14, 15, 17, 18, 29], operating systems [61], and storage [16, 30], but presents unique challenges in large-scale training. In these domains, the main issue is identifying the source of *gradually* propagating [14, 15] or independent [30] fail-slows. In contrast, large-scale training is synchronous, one slow component can *immediately* propagate to the entire cluster. Additionally, replacing degraded components in these fields often does not impact the entire system.

Heterogeneous DL training. Several researches focus on efficient parallel training on heterogeneous hardware with various performance [6, 21, 31, 32, 41, 54, 60, 63]. However, mitigating fail-slow presents distinct challenges. Heterogeneous training is *static*, where performance does not fluctuate over time, thus allowing for higher-cost parallel strategy searches at initial [54, 63]. In contrast, fail-slow handling must be *dynamic*, precluding those high-cost searches.

9 Conclusion

In this paper, we presented the first comprehensive characterization study in a production cluster to understand the overall characteristics and performance impacts of fail-slow failures in large-scale LM training. We proposed GREYHOUND, a framework that swiftly identifies fail-slowed compute or communication components and effectively mitigates them using a novel multi-level mechanism, all without human intervention. GREYHOUND achieves over 99% accuracy in detecting fail-slows and improves the end-to-end performance by 1.58× in large-scale training.

Acknowledgment

We thank our shepherd, Linhai Song, and the anonymous reviewers for their valuable comments that help improve the quality of this work. This work was supported in part by the Alibaba Innovative Research (AIR) Grant, RGC CRF Grant (Ref. #C6015-23G), RGC GRF Grant (Ref. #16217124), and NSFC/RGC CRS Grant (#CRS_HKUST601/24).

References

[1] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

[2] Diego Agudelo-España, Sebastian Gomez-Gonzalez, Stefan Bauer, Bernhard Schölkopf, and Jan Peters. Bayesian online prediction of change points. In *Conference on Uncertainty in Artificial Intelligence*, pages 320–329. PMLR, 2020.

[3] Jason Ansel, Edward Yang, Horace He, Natalia Gimelshein, Animesh Jain, Michael Voznesensky, Bin Bao, Peter Bell, David Berard, Evgeni Burovski, et al. Pytorch 2: Faster machine learning through dynamic python bytecode transformation and graph compilation. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 929–947, 2024.

[4] Jiamin Cao, Yu Guan, Kun Qian, Jiaqi Gao, Wencong Xiao, Jianbo Dong, Binzhang Fu, Dennis Cai, and Ennan Zhai. Crux: Gpu-efficient communication scheduling for deep learning training. In *Proceedings of the ACM SIGCOMM 2024 Conference*, pages 1–15, 2024.

[5] Christopher Chatfield. *The analysis of time series: theory and practice*. Springer, 2013.

[6] Chen Chen, Qizhen Weng, Wei Wang, Baochun Li, and Bo Li. Semi-dynamic load balancing: Efficient distributed learning in non-dedicated environments. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, pages 431–446, 2020.

[7] Mike Chow, Yang Wang, William Wang, Ayichew Hailu, Rohan Bopardikar, Bin Zhang, Jialiang Qu, David Meisner, Santosh Sonawane, Yunqi Zhang, et al. {ServiceLab}: Preventing tiny performance regressions at hyperscale through {Pre-Production} testing. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 545–562, 2024.

[8] Meghan Cowan, Saeed Maleki, Madanlal Musuvathi, Olli Saarikivi, and Yifan Xiong. Msccl: Microsoft collective communication language. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 502–514, 2023.

[9] Steven Diamond and Stephen Boyd. CVXPY: A Python-embedded modeling language for convex optimization. *Journal of Machine Learning Research*, 17(83):1–5, 2016.

[10] Jianbo Dong, Shaochuang Wang, Fei Feng, Zheng Cao, Heng Pan, Lingbo Tang, Pengcheng Li, Hao Li, Qianyuan Ran, Yiqun Guo, et al. Accl: Architecting highly scalable distributed training systems with highly efficient collective communication library. *IEEE micro*, 41(5):85–92, 2021.

[11] Jiangfei Duan, Ziang Song, Xupeng Miao, Xiaoli Xi, Dahua Lin, Harry Xu, Minjia Zhang, and Zhihao Jia. Parcae: Proactive,{Liveput-Optimized}{DNN} training on preemptible instances. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 1121–1139, 2024.

[12] Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

[13] William Fedus, Barret Zoph, and Noam Shazeer. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *Journal of Machine Learning Research*, 23(120):1–39, 2022.

[14] Yu Gan, Mingyu Liang, Sundar Dev, David Lo, and Christina Delimitrou. Sage: Leveraging ml to diagnose unpredictable performance in cloud microservices. *arXiv preprint arXiv:2112.06263*, 2021.

[15] Yu Gan, Yanqi Zhang, Kelvin Hu, Dailun Cheng, Yuan He, Meghna Pancholi, and Christina Delimitrou. Seer: Leveraging big data to navigate the complexity of performance debugging in cloud microservices. In *Proceedings of the twenty-fourth international conference on architectural support for programming languages and operating systems*, pages 19–33, 2019.

[16] Haryadi S Gunawi, Riza O Suminto, Russell Sears, Casey Golliher, Swaminathan Sundararaman, Xing Lin, Tim Emami, Weiguang Sheng, Nematollah Bidokhti, Caitie McCaffrey, et al. Fail-slow at scale: Evidence of hardware performance faults in large production systems. *ACM Transactions on Storage (TOS)*, 14(3):1–26, 2018.

[17] Peng Huang, Chuanxiong Guo, Jacob R Lorch, Lidong Zhou, and Yingnong Dang. Capturing and enhancing in situ system observability for failure detection. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 1–16, 2018.

[18] Peng Huang, Chuanxiong Guo, Lidong Zhou, Jacob R Lorch, Yingnong Dang, Murali Chintalapati, and Randolph Yao. Gray failure: The achilles' heel of cloud-scale systems. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, pages 150–155, 2017.

[19] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, et al. Gpipe: Efficient training of giant neural networks using pipeline parallelism. In *NeurIPS*, 2019.

[20] Insu Jang, Zhenning Yang, Zhen Zhang, Xin Jin, and Mosharaf Chowdhury. Oobleck: Resilient distributed training of large models using pipeline templates. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 382–395, 2023.

[21] Xianyan Jia, Le Jiang, Ang Wang, Wencong Xiao, Ziji Shi, Jie Zhang, Xinyuan Li, Langshi Chen, Yong Li, Zhen Zheng, et al. Whale: Efficient giant model training over heterogeneous {GPUs}. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 673–688, 2022.

[22] Ziheng Jiang, Haibin Lin, Yinmin Zhong, Qi Huang, Yangrui Chen, Zhi Zhang, Yanghua Peng, Xiang Li, Cong Xie, Shibiao Nong, et al. {MegaScale}: Scaling large language model training to more than 10,000 {GPUs}. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 745–760, 2024.

[23] Anna R Karlin, Claire Kenyon, and Dana Randall. Dynamic tcp acknowledgement and other stories about e/(e-1). In *Proceedings of the thirty-third annual ACM symposium on Theory of computing*, pages 502–509, 2001.

[24] Gurkirat Kaur and Manju Bala. Rdma over converged ethernet: A review. *International Journal of Advances in Engineering & Technology*, 6(4):1890, 2013.

[25] Vijay Anand Korthikanti, Jared Casper, Sangkug Lym, Lawrence McAfee, Michael Andersch, Mohammad Shoeybi, and Bryan Catanzaro. Reducing activation recomputation in large transformer models. In *MLSys*, 2023.

[26] Teven Le Scao, Angela Fan, Christopher Akiki, Ellie Pavlick, Suzana Ilić, Daniel Hesslow, Roman Castagné, Alexandra Sasha Luccioni, François Yvon, Matthias Gallé, et al. Bloom: A 176b-parameter open-access multilingual language model. 2023.

[27] Dmitry Lepikhin, HyoukJoong Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Yanping Huang, Maxim Krikun, Noam Shazeer, and Zhifeng Chen. Gshard: Scaling giant models with conditional computation and automatic sharding. *arXiv preprint arXiv:2006.16668*, 2020.

[28] Yuanhao Li, Tianyuan Wu, Guancheng Li, Yanjie Song, and Shu Yin. Portus: Efficient dnn checkpointing to persistent memory with zero-copy. In *2024 IEEE 44th International Conference on Distributed Computing Systems (ICDCS)*, pages 59–70. IEEE, 2024.

[29] Chang Lou, Peng Huang, and Scott Smith. Understanding, detecting and localizing partial failures in large system software. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 559–574, 2020.

[30] Ruiming Lu, Erci Xu, Yiming Zhang, Fengyi Zhu, Zhaosheng Zhu, Mengtian Wang, Zongpeng Zhu, Guangtao Xue, Jiwu Shu, Minglu Li, et al. Perseus: A {Fail-Slow} detection framework for cloud storage systems. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*, pages 49–64, 2023.

[31] Yixuan Mei, Yonghao Zhuang, Xupeng Miao, Juncheng Yang, Zhihao Jia, and Rashmi Vinayak. Helix: Distributed serving of large language models via max-flow on heterogeneous gpus. *arXiv preprint arXiv:2406.01566*, 2024.

[32] Xupeng Miao, Yining Shi, Zhi Yang, Bin Cui, and Zhihao Jia. Sdpipe: A semi-decentralized framework for heterogeneity-aware pipeline-parallel training. *Proceedings of the VLDB Endowment*, 16(9):2354–2363, 2023.

[33] Jayashree Mohan, Amar Phanishayee, and Vijay Chidambaram. {CheckFreq}: Frequent,{Fine-Grained}{DNN} checkpointing. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 203–216, 2021.

[34] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R Devanur, Gregory R Ganger, Phillip B Gibbons, and Matei Zaharia. Pipedream: Generalized pipeline parallelism for dnn training. In *Proceedings of the 27th ACM symposium on operating systems principles*, pages 1–15, 2019.

[35] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, et al. Efficient large-scale language model training on gpu clusters using megatron-lm. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15, 2021.

[36] NVIDIA Corporation. Fabric manager for nvidia nvswitch systems, 2023. Accessed: 2024-09-04.

[37] NVIDIA Corporation. Matrix multiplication background user's guide, 2024. Accessed: 2024-09-17.

[38] NVIDIA Corporation. Nvidia collective communications library (nccl), 2024. Accessed: 2024-09-06.

[39] NVIDIA Corporation. Nvidia management library (nvml), 2024. Accessed: 2025-01-01.

[40] OpenAI. Openai sora, 2024. Accessed: 2024-09-13.

[41] Jay H Park, Gyeongchan Yun, M Yi Chang, Nguyen T Nguyen, Seungmin Lee, Jaesik Choi, Sam H Noh, and Young-ri Choi. {HetPipe}: Enabling large {DNN} training on (whimpy) heterogeneous {GPU} clusters through integration of pipelined model parallelism and data parallelism. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 307–321, 2020.

[42] William Peebles and Saining Xie. Scalable diffusion models with transformers. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 4195–4205, 2023.

[43] Gregory F Pfister. An introduction to the infiniband architecture. *High performance mass storage and parallel I/O*, 42(617-632):10, 2001.

[44] Kun Qian, Yongqing Xi, Jiamin Cao, Jiaqi Gao, Yichi Xu, Yu Guan, Binzhang Fu, Xuemei Shi, Fangbo Zhu, Rui Miao, et al. Alibaba hpn: A data center network for large language model training. In *Proceedings of the ACM SIGCOMM 2024 Conference*, pages 691–706, 2024.

[45] Sudarsanan Rajasekaran, Manya Ghobadi, and Aditya Akella. {CASSINI}:{Network-Aware} job scheduling in machine learning clusters. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 1403–1420, 2024.

[46] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. Zero: Memory optimizations toward training trillion parameter models. In *IEEE/ACM SC*, 2020.

[47] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 3505–3506, 2020.

[48] Jie Ren, Samyam Rajbhandari, Reza Yazdani Aminabadi, Olatunji Ruwase, Shuangyan Yang, Minjia Zhang, Dong Li, and Yuxiong He. {Zero-offload}: Democratizing {billion-scale} model training. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 551–564, 2021.

[49] Salvatore Sanfilippo. Redis - the real-time data platform, 2009. Accessed: 2024-09-08.

[50] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

[51] Shaden Smith, Mostofa Patwary, Brandon Norick, Patrick LeGresley, Samyam Rajbhandari, Jared Casper, Zhun Liu, Shrimai Prabhumoye, George Zerveas, Vijay Korthikanti, et al. Using deepspeed and megatron to train megatron-turing nlg 530b, a large-scale generative language model. *arXiv preprint arXiv:2201.11990*, 2022.

[52] Gemini Team, Rohan Anil, Sebastian Borgeaud, Yonghui Wu, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, Johan Schalkwyk, Andrew M Dai, Anja Hauth, et al. Gemini: a family of highly capable multimodal models. *arXiv preprint arXiv:2312.11805*, 2023.

[53] John Thorpe, Pengzhan Zhao, Jonathan Eyolfson, Yifan Qiao, Zhihao Jia, Minjia Zhang, Ravi Netravali, and Guoqing Harry Xu. Bamboo: Making preemptible instances resilient for affordable training of large {DNNs}. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 497–513, 2023.

[54] Taegeon Um, Byungsoo Oh, Minyoung Kang, Woo-Yeon Lee, Goeun Kim, Dongseob Kim, Youngtaek Kim, Mohd Muzzammil, and Myeongjae Jeon. Metis: Fast automatic distributed training on heterogeneous {GPUs}. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*, pages 563–578, 2024.

[55] Pablo Villalobos, Jaime Sevilla, Tamay Besiroglu, Lennart Heim, Anson Ho, and Marius Hobbhahn. Machine learning model sizes and the parameter gap. *arXiv preprint arXiv:2207.02852*, 2022.

[56] Zhuang Wang, Zhen Jia, Shuai Zheng, Zhen Zhang, Xinwei Fu, TS Eugene Ng, and Yida Wang. Gemini: Fast failure recovery in distributed training with in-memory checkpoints. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 364–381, 2023.

[57] Qizhen Weng, Lingyun Yang, Yinghao Yu, Wei Wang, Xiaochuan Tang, Guodong Yang, and Liping Zhang. Beware of fragmentation: Scheduling gpu-sharing workloads with fragmentation gradient descent. In *2023 USENIX Annual Technical Conference (ATC'23)*, 2023.

[58] Yifan Xiong, Yuting Jiang, Ziyue Yang, Lei Qu, Guoshuai Zhao, Shuguang Liu, Dong Zhong, Boris Pinzur, Jie Zhang, Yang Wang, et al. SuperBench: Improving cloud AI infrastructure reliability with proactive validation. In *2024 USENIX Annual Technical Conference (ATC'24)*, pages 835–850, 2024.

[59] Zhiyi Yao, Pengbo Hu, Congcong Miao, Xuya Jia, Zuning Liang, Yuedong Xu, Chunzhi He, Hao Lu, Mingzhuo Chen, Xiang Li, et al. Holmes: Localizing irregularities in {LLM} training with mega-scale {GPU} clusters. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*, pages 523–540, 2025.

[60] Xiaodong Yi, Shiwei Zhang, Ziyue Luo, Guoping Long, Lansong Diao, Chuan Wu, Zhen Zheng, Jun Yang, and Wei Lin. Optimizing distributed training deployment in heterogeneous gpu clusters. In *Proceedings of the 16th International Conference on emerging Networking EXperiments and Technologies*, pages 93–107, 2020.

[61] Shenglin Zhang, Yongxin Zhao, Xiao Xiong, Yongqian Sun, Xiaohui Nie, Jiacheng Zhang, Fenglai Wang, Xian Zheng, Yuzhi Zhang, and Dan Pei. Illuminating the gray zone: Non-intrusive gray failure localization in server operating systems. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, pages 126–137, 2024.

[62] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, et al. Opt: Open pre-trained transformer language models. *arXiv preprint arXiv:2205.01068*, 2022.

[63] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P Xing, et al. Alpa: Automating inter-and {Intra-Operator} parallelism for distributed deep learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 559–578, 2022.

[64] Yuchen Zhong, Guangming Sheng, Juncheng Liu, Jinhui Yuan, and Chuan Wu. Swift: Expedited failure recovery for large-scale dnn training. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, pages 447–449, 2023.

[65] Keren Zhou, Yueming Hao, John Mellor-Crummey, Xiaozhu Meng, and Xu Liu. Gvprof: A value profiler for gpu-based clusters. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–16. IEEE, 2020.

Crash Consistency in Block-Level Caching Systems: An Open CAS Case Study

Shaohua Duan
Washington State University

Youmin Chen
Shanghai Jiao Tong University

Abstract

Byte-addressable non-volatile memory (NVM) proposes a new opportunity to enable file systems with better performance and durability by adding a new persistent caching layer. However, crash consistency of caching layers and compatibility with diverse reliability features of file systems remains unknown. This paper conducts a crash consistency case study of Open CAS, a popular block-level caching system. Through careful and thorough crash consistency experiments, we show that Open CAS cannot always maintain crash consistency in the persistent caching layer. We also demonstrate some reliability features of file systems are not compatible with Open CAS. Our analysis reveals the importance of a systematic crash consistency test to caching systems and reliability feature co-design with file systems in the construction of a reliable end-to-end file system.

1 Introduction

Byte-addressable non-volatile memory (NVM) opens a new opportunity in designing high-performance block-level caching systems for file systems. NVM provides byte addressability and low-access latency like DRAM and offers data persistence like HDD. Like page cache in traditional file systems, similar hybrid storage is adopted in persistent caching systems, where a fast persistent device is used for a caching layer and a slow but large storage capacity is used for a backend file system to achieve high I/O performance while reducing monetary costs. For example, Open CAS [11], an advanced caching solution recently developed by Intel, allows NVM, SSD, and other high-performance storage devices to be used as a caching layer for improving the I/O performance of Linux file systems.

While the primary goal for introducing non-volatile memory into block-level caching systems is to improve the performance of file systems that are typically installed at the HDDs, i.e., backend devices, it also improves the data durability of file systems. Unlike traditional block-level caching systems (e.g., in-DRAM page cache), block-level caching systems on non-volatile memory guarantee better data durability and enable system designers to leverage new tiered persistent storage devices (e.g., Intel Optane persistent memory [37]) to recover the cached data from system crashes. For example, Open CAS [11] and other persistent caching systems such as bcache [5] and flashcache [28] guarantee the data durability once returning an acknowledgment to users.

Although the advent of fast storage and interconnect technologies present new opportunities to achieve better performance and durability, they also cast new questions on system reliability: How robust are persistent block-level caching systems to system crashes? How compatible are caching systems with diverse reliability features of file systems which were typically designed with in-DRAM page cache? To maximize the performance of block-level caching systems, more and more complicated software architectures, cache strategies and modes, and storage hierarchies are applied which make block-level caching systems tend to be less reliable. Although some block-level caching systems promise data durability and crash consistency in certain crash scenarios by using multiple robustness features, the effectiveness of these reliability designs and the complete crash behavior of the caching systems remain unknown due to a lack of third-party systematical crash consistency tools. For example, Open CAS developers noticed the risk of data durability loss in case of system crashes in certain cache modes, i.e., write-only and write-back modes due to unknown reasons [9]. By contrast, there are multiple third-party domain-specific test libraries and tools [38, 46] focus on crash-consistency tests to file systems, and the crash behaviors of state-of-the-art file systems are well studied. Furthermore, state-of-the-art reliability features of file systems are designed, implemented, and tested with in-DRAM page cache which is modeled as all data in the caching layer are not available after a crash. It is unclear whether these file system reliability features still work well with the new persistent storage hierarchy without suffering compatibility issues. To answer these questions, we analyze Open CAS, a representative block-level caching system, by performing careful and

thorough crash consistency experiments. To the best of our knowledge, this is the first work that specifically focuses on the crash consistency of block-level caching systems.

In this paper, we use Open CAS as an example of block-level caching systems and perform a thorough crash consistency study on persistent block-level caching layers. Our study unveils important, previously undetected crash inconsistency and incompatibility in Open CAS. In our analysis, we find that Open CAS fails to maintain crash consistency and returns bad data to the user due to system crashes. We further find that some reliability features of state-of-the-art file systems are not compatible with Open CAS[1]. Besides Open CAS, we also analyzed a simpler caching system, NVCache [15], and observed similar experiment results. In summary, we make the following contributions:

- The first study (to the best of our knowledge) to empirically analyze the crash consistency behavior of Open CAS, a state-of-the-art block-level caching system.
- A novel workload-oriented test framework (WLOT) to analyze crash consistency behavior of caching systems by leveraging implicit information.
- Results and observations that demonstrate the importance of systematical crash-consistency tests to caching systems and compatibility tests with diverse reliability features of file systems.

The rest of this paper is organized as follows. In Section 2, we motivate our work and discuss the challenges for crash-consistency tests to block-level caching systems. Section 3 provides some background on Open CAS. Section 4 presents the principle of crash-consistency tests to caching systems and our code inspections to Open CAS. In Section 5 and Section 6, we present our analysis of crash consistency experiments on Open CAS. In Section 7, we present the initial results of the crash consistency analysis in NVCache. Section 8 discusses related work and Section 9 concludes our work.

2 Block-Level Caching Systems

Caching systems have long been the most popular design for improving the performance of storage systems [6, 20, 40]. In caching systems, consider a system with two storage layers: a (fast, expensive, small) performance layer and a (slow, cheap, large) capacity layer. With caching, all data resides in the capacity layer, and copies of hot data items are placed, via cache replacement algorithms [7, 34], in the performance layer. With a high-enough fraction of requests going to the fast layer, the overall performance approaches the peak performance of the fast layer. In the context of block-level caching systems, e.g., Open CAS [11], OrthusCAS [50], First Responder [42], Assise [1], and NVCache [15], they are generally implemented as a kernel module in operating systems, and provide caching service for block devices which typically are file systems.

[1]We have reported the issues exposed in our study to the developers and have received acknowledgment.

The advent of fast storage and interconnect technologies, such as persistent memory (PMem) [37], Non-Volatile Memory Express (NVMe) [30], and Compute Express Link (CXL) [13], present new opportunities in boosting the performance of memory hierarchy and caching systems. With new storage technologies like 3D XPoint [43], NVMe SSDs (over the PCIe bus) achieve much higher bandwidth (e.g., 8 GB/s) and lower latency (e.g., 3μs) [58]. Further, byte-addressable persistent memory has also been commercially available [37] and offers an order of magnitude higher capacity than DRAM and within an order of magnitude performance of DRAM [52].

2.1 Why Crash Consistency Does Matter

Although the main goal for introducing nonvolatile memory to caching systems is for performance boosting [12, 19, 24, 25, 49, 51, 57], this technique also poses opportunities and challenges to the reliability of end-to-end file systems, including caching layers and backend file systems. During the past decades, there have been many related works to improve the reliability of file systems [14, 23, 29, 35, 36, 55], and they typically modeled the behavior of in-memory cache, i.e., page cache, as all information in the caching layer is lost upon a crash. However, when nonvolatile memory is introduced into block-level caching systems, this stale crash model doesn't hold, and it provides a new opportunity for caching systems to reuse the cached data for fast system recovery. For example, Open CAS provides a configuration option for the system managers to perform a fast recovery and reuse the cached data from an unplanned system shutdown.

Although a persistent caching layer automatically provides data durability due to the natural behavior of nonvolatile memory, i.e., it retains data even when power is removed, adding a new tiered persistent caching layer also poses critical challenges to the system reliability. First, a persistent caching system is much more complex than traditional in-memory page cache. To meet the various use scenarios and user requirements, a persistent caching system normally provides multiple cache modes to cope with. For example, compared with a single cache mode in page cache, Open CAS provides six cache modes for the different use scenarios. Similarly, First Responder offers seven cache modes for users. Furthermore, to maximally deliver the performance of nonvolatile memory, caching systems normally design complicated caching algorithms and asynchronous data movement mechanisms to fit in nonvolatile memory characteristics. For example, in OrthusCAS [50] the cache eviction strategy not only considers the characteristics of workloads but also is decided by the performance model of caching devices. In general, the more complicated software design and implementation means the systems are less reliable.

Second, testing persistent caching systems poses unique requirements and challenges. In caching systems, the code

logic usually requires a heavy workload to change the caching system status and/or trigger the specific cache operation. For example, cache eviction and cache cleaning operations generally require a heavy workload to be implicitly triggered by cache replacement policies [7,34]. The requirement for heavy workloads makes traditional system testing tools, such as fuzzing [4,17,23,53] and black-box testing [29] less effective which tend to generate lightweight test cases by applying syntax and semantic mutations [22,54]. Furthermore, implicitly triggering through heavy workloads also means there are enormous potential crash points in the heavy-workload test case for a test framework to explore. Therefore, crash-consistency tests to caching systems specifically require a test framework that can generate heavy workloads while precisely identifying crash points for implicit cache operations.

Third, it still remains unknown whether the added persistent caching layer is compatible with the diverse reliability features of file systems. For file systems, the reliability features such as logging [18,39], journaling [48], write-back [47,48], and copy-on-write [21], are designed on a simple crash model that file systems on the persistent device, e.g., HDD, are the only survived data source upon system crashes [3]. When introducing a new persistent caching layer in end-to-end file systems, this stale crash model doesn't hold anymore, and the reliability design based on this model may not be compatible with the newly added persistent caching layer. Actually, in case of a system crash, data movement between the persistent caching layer and the backend file system immediately terminates, which is the same as in-memory page cache. However, during the post-recovery process, the survived data in caching layers and file systems can be inconsistent. If the reliability design between caching layers and file systems is not well coordinated, the end-to-end file system may suffer from durability loss or data inconsistency issues.

In this work, we focus on the crash consistency of block-level caching systems. We specifically take Open CAS as an example and try to answer the system reliability questions in two aspects: does the persistent caching layer maintain crash consistency in case of system crashes? Are state-of-the-art reliability features of file systems compatible with the newly added persistent caching layer?

3 Introduction to Open CAS

Open Cache Acceleration Software, short for Open CAS [11], is a block-level caching solution that is primarily designed to improve the performance of HDDs (backend devices) by caching data on nonvolatile memory (cache devices). Open CAS is widely used and studied in both industrial solutions [31,33] and research projects [16,50]. Open CAS Linux is a kernel module that works under Linux in-kernel file systems. It handles I/O requests from upper-layer file systems through a virtual block device and submits I/O requests to underlying storage devices according to cache modes and policies. In the official document, Open CAS guarantees that data integrity is maintained on the persistent cache device once returning an acknowledgment to users for all cache modes [10]. To achieve this guarantee, Open CAS developers utilize multiple reliability techniques. Besides, a set of built-in test cases is provided to ensure the correct implementation of functionalities and reliability features.

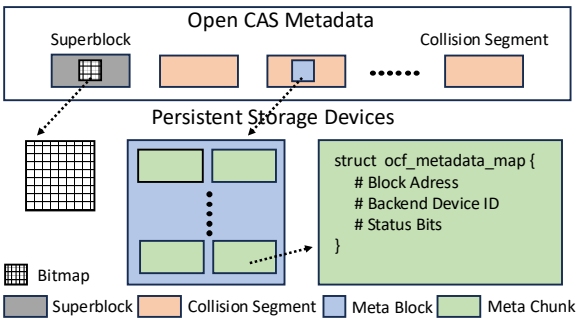


Figure 1: **Metadata Structures in Open CAS**

For the rest of this section, we first introduce the key data structures in Open CAS. We then introduce cache modes and basic cache operations including their reliability features for maintaining crash consistency. After that, We analyze the challenges of performing crash-consistency tests to implicit cache operation. Finally, we describe the post-recovery process of Open CAS.

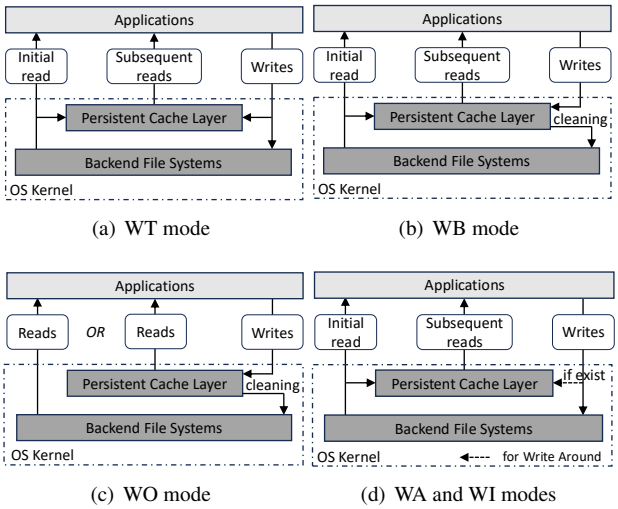
3.1 Data Structures

A `cache line` is the smallest data unit that Open CAS can manage. Every occupied `cache line` is associated with a `core line` on the backend file system and the `cache line` will be eventually written back to its associated `core line`. Open CAS supports different `cache line` sizes ranging from 4KB to 64KB. For each `cache line`, Open CAS maintains a small portion of metadata, which contains the information including core id, core line number, and valid and dirty bits for every sector. The core id determines on which core is the `core line` corresponding to the given `cache line` located, whereas the core line number determines the precise `core line` location on the backend file systems. Valid and dirty bits define the current `cache line` state. When a `cache line` is valid, it means that it's mapped to a `core line` determined by a core id and a core line number. Otherwise, all the other cache line metadata information is invalid and should be ignored. When a `cache line` is in an invalid state, it may be used to map `core line` accessed by I/O requests, and then it becomes valid. The dirty bit determines if the user data stored in the caching layer is in sync with the corresponding data on the backend file systems. If a `cache line` is dirty, then only data on the caching layer is up to date, and it needs to be flushed at some point in the future and be marked as clean by zeroing the dirty bit.

More specifically, as shown in Figure 1, Open CAS has different types of metadata information stored in multiple

Cache modes	Read cache hit	Write cache hit	Read/Write cache miss	Cache eviction	Cache cleaning	Cache flushing
Write-Through (WT)	-	Update	Mapping, Insertion	Eviction, Invalidation	-	-
Write-Back (WB)	-	Update	Mapping, Insertion	Eviction, Invalidation	Cleaning, Invalidation	Flushing, Invalidation
Write-Around (WA)	Invalidation	-	Mapping, Insertion	Eviction, Invalidation	-	-
Write-Invalidation (WI)	-	Update	Mapping, Insertion	Eviction, Invalidation	-	-
Write-Only (WO)	-	Update	Mapping, Insertion	Eviction, Invalidation	Cleaning, Invalidation	Flushing, Invalidation
Pass-Through (PT)	-	-	-	-	-	-

Table 1: **The involved cache operations in different cache modes.** Blank cells mean that none of the cache operations is applicable.



(a) WT mode (b) WB mode

(c) WO mode (d) WA and WI modes

Figure 2: **Cache modes of Open CAS.** Cache eviction and flushing are not shown in all figures.

segments, among which the "collision" segment is the key data structure for storing Open CAS `cache line` metadata. The collision section includes a set of 4KB metadata blocks in DRAM, each of which consists of multiple 12-byte chunks (12, 18, and 42 bytes for `cache line` sizes = 4KB, 16KB, and 64KB respectively) storing the mapping between a `cache line` and a `core line` and status bits (valid and dirty). In addition, the superblock segment includes the available `cache line` bitmap block and the recovery block which stores the option of post-recovery process.

3.2 Cache Modes

Open CAS accelerates Linux file systems by caching active (hot) data to caching layers, i.e., performance devices, thus reducing storage latency. Open CAS provides six cache modes, and based on performance improvements to read and/or write I/O requests, we generally classify them into two categories. In the read cache, initial read data is retrieved from backend file systems and copied to the caching layer. Subsequent reads are returned the copied data from the caching layer at the speed of performance devices. In the write cache, all data is written to the caching layer and eventually flushed to the backend file systems through the cache eviction and cleaning operations. For all cache modes, when the caching layer is full, newly identified active data evicts stale data from the caching layer by utilizing the cache eviction operation.

Write-Through mode: as shown in Figure 2(a), Open CAS writes data to the caching layer and simultaneously writes the same data "through" to the backend file systems. Write-through ensures the data written to the caching layer is always in sync with data on the backend file system. This mode only accelerates read requests, as writes need to be performed on both the backend file system and caching layer.

Write-Back mode: as shown in Figure 2(b), Open CAS writes the data first to the caching layer and acknowledges to the application that the write request is completed before the data is actually written to the backend file systems. Based on the cleaning policy, Open CAS writes back those data to the backend file system opportunistically. Write-back mode improves both write-intensive and read-intensive requests.

Write-Only mode: as shown in Figure 2(c), Open CAS writes user data exactly like in Write-Back mode so the data is written to the caching layer without writing it to backend file systems immediately. Read requests do not promote data to the caching layer.

Write-Around mode: as shown in Figure 2(d), Open CAS writes data to the caching layer if and only if that cache line is already mapped into the caching layer that it can be mapped during read requests. Simultaneously, Open CAS writes the same data "through" to the backend file systems. This cache mode only accelerates read-intensive requests.

Write-Invalidate mode: as shown in Figure 2(d), Open CAS handles write requests by writing data directly to backend file systems, and if the written cache line is already mapped into the cache, then it becomes invalid. Write-invalidate mode only improves read-intensive requests. It also reduces the number of eviction operations for workloads where written data is not often subsequently read.

Pass-Through mode: Open CAS bypasses the caching layer for all I/O requests. This mode is used for dynamically enabling/disabling caching layers without requiring to manually alter the I/O path in the application.

3.3 Cache Operations

There are seven cache operations upon which cache modes are constructed. To guarantee crash consistency of cache operations, Open CAS utilizes multiple techniques like atomic metadata update, I/O request ordering, and flapped sections. Getting familiar with them is essential to understanding how the test procedure is designed to cover all possible crash sce-

narios. It also helps us to explain our crash consistency experiment results.

Mapping is a cache operation of linking up a `core line` with a `cache line` in the `cache line` metadata. During mapping, a single `cache line` is assigned to the `core line`, and during cache operations, this area can be used for storing `core line` data, until the `cache line` is evicted.

In Open CAS, `cache line` metadata is stored in a small and constant block, i.e., 4K. Cache operations such as `insertion`, `invalidation`, etc. can atomically update `cache line` metadata to the caching layer by requiring a single I/O request and guarantees `cache line` metadata transactional consistency during performing I/O operations.

In addition, to ensure atomic update of the `cache line` bitmap, the flapped sections are designed. Specifically, the flapped sections in Open CAS maintain two copies of the `cache line` bitmap and information about which copy is valid. On the bitmap update, the bitmap flush is performed to an invalid copy, and then only after this operation succeeds the updated copy is marked as valid in the superblock, which is flushed afterward.

Insertion is a cache operation of writing `core line` data to a `cache line`. During insertion, the data of the I/O request is written to the `cache line` corresponding to `core line` being accessed. The valid and dirty bits of the cache line in the metadata are being updated accordingly during this operation.

To guarantee `cache line` data integrity, `insertion` operation is designed by ordering the I/O requests of `cache line` data and its metadata. Specifically, `insertion` first flushes the user data to `cache line`. When it is done, it then changes the corresponding validation bit from invalid to valid and updates the corresponding `cache line` metadata in the caching layer. Due to the initial invalid state of `cache line`, the corrupted data, which are caused by a system crash during flushing them to the caching layer, aren't accessed by users or flushed to the backend file system.

Update is a cache operation of overwriting `cache line` data in the caching layer. It's performed when the `core line` accessed during a write request is mapped into the caching layer and the accessed sectors are marked as valid.

Invalidation is a cache operation of marking one or more `cache line` sectors as invalid. It can be done during the handling of discard requests, during purge operations, or as a result of some internal cache operations.

Flushing is a cache operation of flushing all dirty data to the backend file system. `flushing` is a cache management operation and has to be triggered by users or Open CAS Udev rule. It is typically done for safely detaching the backend file system from the caching system or recovering the backend file system without risking data loss.

Eviction is a cache operation of removing `cache line` mappings. It's performed when there is not sufficient space in the caching layer for mapping data of incoming I/O requests, i.e., when cache occupancy is close to 100%. The `cache lines` to be evicted are selected by an eviction policy algorithm. Open CAS supports multiple eviction policies including the Least Recently Used (LRU) which selects `cache lines` that have not been accessed for the longest period of time.

Cleaning is a cache operation of synchronizing data in the caching layer and the backend file system. It involves reading sectors marked as dirty from the caching layer, writing them to the backend file systems, and setting dirty bits in `cache line` metadata to zero. It's performed by the cache automatically as a background task. The process of cleaning is controlled by a cleaning policy. Open CAS supports multiple cleaning policies including Approximately Least Recently Used (ALRU), Aggressive Cleaning Policy (ACP), and No Operation (NOP).

Finally, it is important to note that Open CAS doesn't record an operation log and persist it in the caching layer for the debug or failure recovery due to maximizing the caching performance. Table 1 summarizes the involved cache operations in different cache modes.

3.3.1 Implicit Cache Operation

Block-level caching systems like Open CAS are typically integrated into Linux operating system as a kernel module. The advantage of the integration is that the cache solution is transparent to the user applications and existing storage infrastructures, and no storage migration or application changes are required. However, due to restrictions by the existing Linux user interfaces, i.e., the POSIX standard [32], some cache operations cannot be explicitly invoked by the user interfaces. It also means that it is hard for a test framework to test these operations by explicitly invoking while performing system crashes. Furthermore, as a portion of Linux operation system, kernel modules are unexpected to change for software testing, e.g., adding the stub code. Actually, these cache operations are only invoked by internal cache policies, e.g., Least Recently Used (LRU) [7, 34, 50], which are typically decided by a heavy workload. We call these cache operations an implicit cache operation. For example, in Open CAS, `eviction` and `cleaning` are implicit cache operations. In summary, due to the implicit invocation and heavy workload requirement, an implicit cache operation is hard to test, and we will introduce our solution, workload-oriented test framework (WLOT) in Section 5.

3.4 Post-Recovery Process

Open CAS post-recovery process provides two options to recover a caching layer from a crash. The first option is to deal with a caching layer as page cache in volatile memory (e.g., DRAM). Specifically, Open CAS reinitializes the caching layer with new metadata, which clears the cache device, resulting in the loss of all cache data that is currently residing in the cache device.

The second option, a more sophisticated one, is to restart a caching layer and utilize the previous cache data, including metadata, in the cache device for a fast recovery. Specifically, Open CAS first performs CRC checksums for all metadata sections in the cache device and then compares them with ones stored in the recovery section to guarantee the metadata data integrity. To guarantee the data integrity of the recovery section, the recovery size is kept under 4K, which makes Open CAS update the recovery section atomically with a single 4K I/O. Then, Open CAS mounts the core devices, i.e., backend file systems, and the corresponding file system recovery process is activated according to the file system's reliability configuration. If the cache device and core device are successfully recovered, the recovered data in the caching layer can be accessed and reused by the backend file system and the applications. Open CAS recovery process is performed automatically in accordance with dynamic device management `Udev` [26]. Open CAS administrator can set the post-recovery process option and enable the caching layer to automatically flush the dirty data to the recovered file systems by configuring the in-built `Udev` rules in `60-persistent-storage-cas.rules`. Note that although Open CAS developers also mention the potential risk of data durability loss under this recovery option [9], there is no clear description or guidance to help users to evaluate the tradeoff between performance benefit and reliability risk for choosing the suitable recovery options. Therefore, we believe a thorough study is much needed.

4 Crash-Consistency Tests to Caching Systems

In this section, we first investigate the general principles for crash-consistency tests to a block-level caching system. Then, we take the in-build test suite in Open CAS as an example and specifically inspect its crash-consistency tests. Our aim is to find whether Open CAS in-build crash-consistency tests can guarantee data consistency in case of system crashes.

4.1 Principle of Reliability Test

Principle 1: *Crash-consistency tests should explore the crash behaviors of a caching system for crashes at any time.* Although crash consistency and data durability may not be guaranteed in any crash scenario, a system crash, such as a power outlet and system corruption, can happen at any time. It's necessary to understand complete crash behavior rather than solely crash scenarios where crash consistency is promised by caching system developers. By contrast, the crash consistency behaviors of file systems under any time system crashes were well studied [36]. More importantly, it helps system administrators to identify the root cause of crash inconsistency issues in the end-to-end file system. For end-to-end file systems, it can include multiple persistent caching layers, local file systems, and remote storage components. Each persistent layer and component may exhibit various crash consistency characteristics. Finding the root cause of crash inconsistency issues to such systems can be a daunt task. Understanding the complete crash consistency characteristics of individual storage layers eases system administrators to identify the root cause of crash inconsistency issues.

Principle 2: *Crash-consistency tests should cover all crash-vulnerable cache operations.* Like crash-consistency tests to other storage systems, it's necessary for test cases to cover all crash-vulnerable operations that potentially cause crash inconsistency. For example, many file system test tools [23, 29] more focus on testing the write-related operations since they change the internal state and the persistent user data. For caching systems, if a cache operation updates user data and/or caching system metadata in the caching layer, it will change the caching system state and be potentially vulnerable to system crashes. This cache operation can be seen as a crash-vulnerable cache operation. For example, in Open CAS, all cache operations can be seen as crash-vulnerable cache operations. By contrast, the cache operation for responding to a read cache hit request is not a crash-vulnerable cache operation since it doesn't change user data or caching system metadata.

Principle 3: *Crash-consistency tests should include diverse reliability features of file systems to guarantee compatibility with them.* The state-of-the-art reliability features of file systems, such as data journaling, copy-on-write, etc., are designed under the assumption that cached data is not available after crashes. During the post-recovery process, reusing the cached data in the persistent caching layer for a fast recovery may cause compatibility issues such as durability loss and crash inconsistency.

4.2 Crash-Consistency Tests in Open CAS

Open CAS provides a set of test cases, including unit tests and integration tests to guarantee correct implementation. Regarding system reliability, Open CAS offers crash-consistency tests in the integration test suite. We find that the test cases provided by Open CAS aren't enough to guarantee crash consistency and compatibility.

Code Observation 1: *Open CAS didn't provide crash-consistency tests to implicit cache operations.* Open CAS provides the test cases to all cache operations for functionality testing under normal cache scenarios. However, in crash-consistency tests, the test cases didn't include implicit cache operations. One possible reason for crash-consistency tests without including implicit cache operations is that it is difficult for a test framework to identify the running state of such cache operations for precisely injecting synthetic crashes.

Code Observation 2: *Open CAS didn't explore crash behaviors when crashes happen during cache operations on the fly.* In Open CAS, the in-built crash-consistency tests only cover the crash scenarios in which system crashes happen right after an acknowledgment of user APIs, which are the guarantees

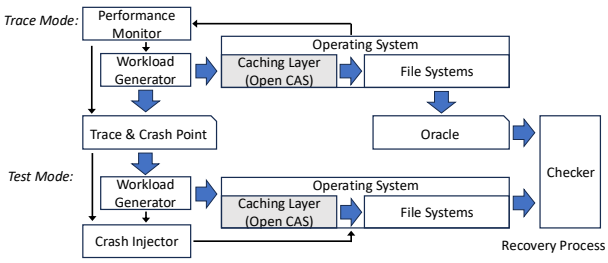


Figure 3: **Overview of crash-consistency test framework.**

of data durability and consistency.

Code Observation 3: *There are no test cases testing the compatibility with diverse reliability features of file systems.* In Open CAS, crash-consistency tests only perform synthetic crashes to the caching layer and takes one specific file system, i.e., ext-4, with the default configuration as the backend file system. The back logic of such test cases is based on the following assumption. If the newly added caching layer itself can keep crash consistency successfully, it should naturally work well with other file systems that include diverse reliability features.

In summary, crash consistency of Open CAS remains unknown in some crash scenarios. Also, extra test cases are required to guarantee the compatibility with diverse reliability features of file systems. We will present our analysis, including test methodology and results in later sections.

5 Crash-Consistency Tests to Caching Layer

Our crash-consistency tests to Open CAS includes two parts. One is crash-consistency tests to caching layers. The other is compatibility tests with backend file systems. In this section, we mainly focus on crash consistency of caching layers and perform system crashes solely on caching layers and try to answer the following research question: Can Open CAS keep the internal metadata and cached data crash consistency under all cache operations, cache modes, and crash scenarios? We will present compatibility tests to file systems in section 6, in which we explore Open CAS compatibility with diverse reliability features of state-of-the-art file systems.

This section is organized as follows. We first introduce our test framework, procedure, and workload for crash-consistency tests. We then discuss the results of our crash-consistency tests.

5.1 Test Framework

In this section, we first show the overview of our test framework. We then specifically present the workload-oriented test (WLOT) for testing implicit cache operations.

5.1.1 Overview of Test Framework

Our test framework is specifically performed on an HPE ProLiant DL380 Gen10 Plus server with 1 × Intel Xeon Silver 4314 processor, 1 × 4GB Optane Pmem as a cache device, and 1 × 256GB Samsung SATA SSD PM9A3 as a backend storage device. Above the hardware, we install Red Hat enterprise server 7.9 with 5.4.0-144-generic Linux kernel. We install Open CAS (version 22.03.2) as a caching system and ext-4 [47] as a backend file system. In addition, we set ext-4 with the default configuration and enabled direct I/O (–direct=1) to bypass page cache. WLOT has two modes of operation: trace mode and test mode. Figure 3 shows the main components and workflow in these two modes.

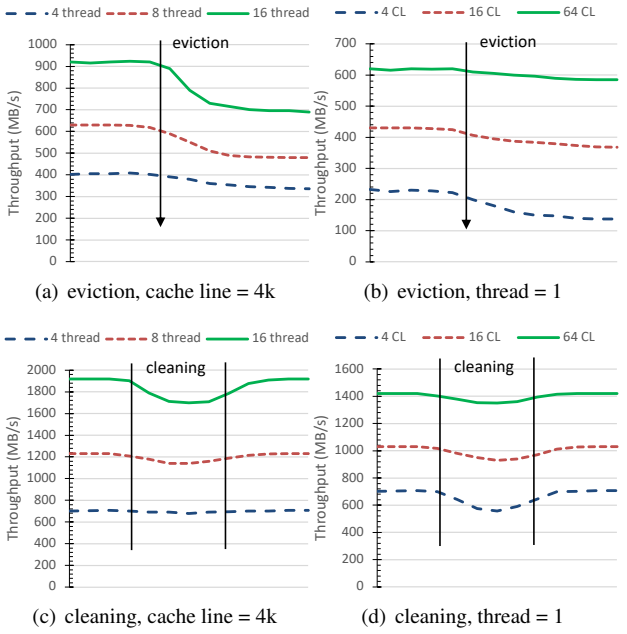
In trace mode, the framework first generates a specific workload and tracks write and/or read requests. While generating I/O requests to the caching layer, the framework utilizes the Open CAS command `casctl static` to monitor the I/O performance of the caching layer and sets crashing points which is based on the performance degradation caused by implicit cache operations. Right after setting crashing points, the framework stops the I/O requests and performs a flush operation to flush all cached data to a file to generate the test oracle. Except for the performance monitoring, we do similar steps for the trace generation and crash point setting to other cache operations.

In test mode, the framework first feeds the workload trace and injects a preset synthetic crash to the caching layer, and then triggers the post-recovery process of the caching systems. Finally, the checker checks the correctness by comparing the reads from the recovered caching system with the oracle. Two modes of operation are specifically important for testing implicit cache operations because the framework cannot identify the crash "window" for implicit cache operations without running the workload to the caching layer.

5.1.2 Workload-Oriented Test

The key challenge for performing crash-consistency tests to caching layers is to precisely inject a synthetic crash while executing an implicit cache operation, i.e., eviction and cleaning operation. As we discussed in section 3, an implicit cache operation can only be activated by cache policies rather than being explicitly invoked by user APIs. To address this challenge, we utilize the concept of implicit information [2, 41] in the operating system design and present the workload-oriented test (WLOT), a generic crash-consistency test framework for caching systems. Implicit information is a powerful tool to understand the running state of a gray box system, such as the operating system kernel, which knows the internal mechanism and external behavior but is ineligible to change the internal design including the interfaces. In crash-consistency tests to caching systems, WLOT detects the performance degradation of a caching system and leverages the performance degradation as implicit information to identify cache operation running states for a crash injection.

To better understand the performance degradation caused by implicit cache operations, we demonstrate an experiment for cache `eviction` and `cleaning` operations. We apply 4GB



(a) eviction, cache line = 4k (b) eviction, thread = 1

(c) cleaning, cache line = 4k (d) cleaning, thread = 1

Figure 4: **Performance Degradation by eviction and cleaning operations.** We measure the throughput degradation of Open CAS while performing eviction and cleaning operations under different cache line sizes and the number of threads.

Optane Pmem and 256GB PM9A3 as a cache device and a backend storage device separately. For the `eviction` operation, we continuously write data to the cache device, until it is full, to force the cache eviction policy to trigger the `eviction` operation. For the `cleaning` operation, we initialize the cache device with a 2GB cached data and repeatedly write a small subset of cached data (2MB) to the cache device, and wait for the cache cleaning policy to trigger the `cleaning` operation. To understand impacts of various factors, we test different `cache line` sizes = 4KB, 16KB, and 64KB, and various intention of workloads (# threads = 2, 4, 8, and 16). Figure 4 shows the I/O throughput degradation when performing `eviction` and `cleaning` operations. The results indicate that throughput degradation becomes significant when applying an intensive workload and a small size `cache line` size. Based on the above observations, we apply the following settings for our test procedure to help the test framework to identify the running state of `eviction` and `cleaning` operations more precisely. We use FIO (version 3.34-13) in 16 threads as the workload setting and 4K `cache line` size as the caching layer setting.

To confirm the accuracy of WLOT further, we specifically implement user APIs `casctl cleaning` and `casctl eviction` as testing drivers on Open CAS (version 22.03.2) for cache implicit operations `cleaning` and `eviction` separately. We then applied the same workloads above and explicitly invoked `cleaning` and `eviction` operations, and saw the same performance degradation pattern shown in Figure 4. We believe WOLT provides high accuracy for testing implicit cache operation with better generality and less human effort involved.

5.2 Test Procedure and Workloads

Since we specifically focus on the crash consistency of caching layers, we only perform synthetic crashes on the caching layer and leave the backend file system alive throughout our crash-consistency tests. Specifically, we use `unmount` and Open CAS's command `casctl stop` to perform a synthetic crash on the caching layer. By doing so, all in-DRAM user data and metadata that are related to the caching system are not available after a synthetic crash. We utilize `mount` and Open CAS's command `casctl start` to only activate the caching layer post-recovery process. We perform synthetic crashes during performing a cache operation and right after receiving an API return. Also, for each cache mode, we perform crashes for each cache operation multiple times to cover all possible crash scenarios.

The workloads are carefully crafted to cover all cache modes and cache operations including implicit cache operations. We set the capacity of the cache device as 4GB while utilizing FIO to write and read an oversized file with 6GB size to the Linux virtual block device. We specifically list workloads for each cache mode and cache operation below.

Write-Through mode: We apply sequential write (SW) and sequential read (SR) workloads, in which we sequentially write/read data into/from an oversize file, to test cache operations `mapping`, `insertion`, and `eviction` in the read cache miss, write cache miss, and cache eviction states separately. To test cache operation `update` in the write cache hit, we also apply repeat write (RW) workload, in which we repeatably write a small portion of the cached data to the caching layer.

Write-Back mode: We apply sequential write (SW) and sequential read (SR) workloads to test cache operations `mapping`, `insertion`, and `eviction` in the read cache miss, write cache miss, and cache eviction states separately. To test cache operations `update`, `cleaning`, and `flushing` in the write cache hit, cache cleaning, and cache flushing, we also apply repeat write (RW) workload.

Write-Around mode: We apply sequential read (SR) workload to test cache operations `mapping`, `insertion`, and `eviction` in read cache miss and cache eviction states separately. We also apply an update-after-read (UR) workload, in which we sequentially read and then write the same user data from an oversized file to test cache operation `update` in the write cache hit.

Write-Invalidate mode: We apply sequential read (SR) workload to test cache operations `mapping`, `insertion`, and `eviction` in read cache miss and cache eviction states separately. In addition, we apply repeat write (RW) workload to test cache operation `invalidate` in a write cache hit.

Write-Only mode: We apply sequential write (SW) workload to test cache operations `mapping`, `insertion`, and `eviction`

Cache mode	Workload	Crash point	Result-1	Result-2
Write-Through	sequential write	write cache miss	R	C
		cache eviction	R	C
	sequential read	read cache miss	R	C
		cache eviction	R	C
	repeat write	write cache hit	R	B
Write-Back	sequential write	write cache miss	R	C
		cache eviction	R	C
	sequential read	read cache miss	R	C
		cache eviction	R	C
	repeat write	cache cleaning	-	C
		cache flushing	R	C
		write cache hit	R	B
Write-Around	sequential read	read cache miss	R	C
		cache eviction	R	C
	update after read	read cache miss	R	C
		cache eviction	R	C
		read cache hit	R	B
Write-Invalidate	sequential read	read cache miss	R	C
		cache eviction	R	C
	repeat write	write cache hit	R	B
Write-Only	sequential write	write cache miss	R	C
		cache eviction	R	C
	repeat write	cache cleaning	-	C
		cache flushing	R	C
		write cache hit	R	B

Table 2: **Crash-consistency test results of caching layers.** The table shows the results of crash-consistency experiments on caching layers. Result 1 and result 2 represent crashes that happen right after acknowledgments and during performing cache operations separately. Each cell indicates whether Open CAS was able to recover data from a crash (R), whether Open CAS maintains crash consistency but did not successfully persist user data (C), and whether Open CAS returned bad data to the user (B). Blank cells mean that the result was not applicable.

in the write cache miss and cache eviction states separately. In addition, we apply repeat write (RW) workload to test cache operations `update`, `cleaning`, and `flushing` in a write cache hit, cache cleaning, and cache flushing. Table 2 summarizes the test workloads with the inserted crash points for all cache operations and modes.

For the check criteria, we check Open CAS log files and API returns for error messages and states. We also compare the cached data with the oracle, which is recorded during generating I/O workload, to ensure user data correctly persists in the caching layer when an acknowledgment returns to applications.

5.3 Results and Observations

The results of our crash-consistency tests are shown in Table 2. We now explain the results in detail in terms of the observations we made from our crash-consistency tests.

Observation 1: *Open CAS gracefully maintains crash consistency and data durability to caching layers in all cache operations and modes once it returns an acknowledgment to users.* This behavior is shown by the results (R) in the result-1 column of Table 2. Just like what Open CAS designers guarantee, we found that Open CAS gracefully recovers from a crash and maintains data consistency and durability, including internal metadata, after returning acknowledgment to users. There are no asynchronous I/O operations to flush data from DRAM to the persistent caching layer. Before returning an acknowledgment, all data and metadata have safely persisted in the caching layer.

Observation 2: *Open CAS maintains crash consistency to caching layers if a crash happens during cache miss and cache eviction.* This behavior is shown by the results (C) in the result-2 column of Table 2. Write/Read cache miss is activated by an initial write/read request and involves cache operations `mapping` and `insertion` which include three key steps:

1. Mapping `cache line` to `core line` and initialize valid bit of `cache line` to be invalid (atomic operation).
2. Write new user data to the cache device.
3. Write new metadata to the cache device (atomic operation).

Since the initial valid bit of `cache line` for caching the new data is invalid. Any corrupted data due to a crash in step 2 will not be returned to users or flushed to backend file systems. The corresponding `cache line` will be reused for caching other data in the future. Similarly, cache eviction involves cache operations `cleaning` and `invalidate` which include five key steps:

1. Read old data from the cache device to DRAM.
2. Write old data to the backend storage device.
3. Update status bits to be invalid and write it to the cache device (atomic operation).
4. Write new user data to the cache device.
5. Write new metadata to the cache device (atomic operation).

Notice again, since the valid bit of the evicted `cache line` for the new user data is updated to invalidate, any corrupted data later in that `cache line` will not be returned to the user or flushed to backend file systems.

Observation 3: *Except for write-around mode, Open CAS returns bad data to users if a crash happens during a write cache hit.* This behavior is shown by the results (B) in the result-2 column of Table 2. Unlike a write cache miss, in the write cache hit, there is no step 1 that initializes the corresponding `cache line` metadata as invalid when updating user data. The corrupted user data due to a crash can be cached to the caching layer and finally flushed to the backend file system and returned to users.

Observation 2 and Observation 3 also lead to an interesting conclusion that cold data is likely to tolerate more serious crashes than hot data which frequently update or access.

Observation 4: *Open CAS returns bad data to users if a crash happens during a write request to the cached data in write-around mode.* This behavior is shown by the results (B) in the result-2 column of Table 2. In write-around mode, processing a write request is mainly processed by the cache operation `invalidate` and involves two key steps:

1. Write the updated data to the backend file system.

2. Update status bits of metadata to invalid and flush it to cache device (atomic operation).

If Open CAS successfully writes the new data to the backend file system but fails to flush the corresponding metadata to the caching layer, the stale user data in the caching layer can return to users. Notice again, since the read cache hit doesn't involve cache operations in other cache modes, we only inject synthetic crashes during a read cache hit in write-around mode.

In summary, for all cache operations and modes, Open CAS successfully maintains crash consistency and persists the cached data once returns an acknowledgment to users. Meanwhile, when a system crash happens during cache operations on the fly, some cache operations cannot maintain crash consistency or data durability. The corrupted data may return to users and flush to the backend file system.

6 Compatibility Tests with File Systems

In Section 5, we showed crash-consistency tests to the caching layer. In this section, we try to explore how compatible Open CAS is with diverse reliability features of state-of-the-art file systems, i.e., whether post-recovery processes of file systems can coordinately work well with persistent caching layers. Our compatibility tests with file systems indicates that some reliability features of file systems are not compatible with Open CAS: incompatibility issues are various from minor issues, e.g., performance degradation to serious ones, e.g., durability loss.

This section is organized as follows. First, we discuss our test framework, procedure, and workloads we used to conduct compatibility tests. Then, we present the results and our observations.

6.1 Test Framework

Our test framework for compatibility tests is similar to crash-consistency tests to caching layers. The difference is the backend file system setting. We choose five state-of-the-art file systems with diverse reliability features as backend file systems. The target file systems roughly fall into three categories: 1) file systems with no or simple reliability features, i.e., ext-2 [8]; 2) file systems with journaling approaches, i.e., ext-3 [48] and ext-4 [47] in writeback, ordered, journal, and nodealloc configurations, and xfs [45]; 3) copy-on-write file systems, i.e., btrfs [27]. Table 3 lists backend file systems and their reliability features, i.e., configuration settings, we choose for our compatibility tests.

6.2 Test Procedure and Workloads

We study the compatibility of Open CAS under the more realistic crash-recovery procedures. During the crash procedure, we perform a crash as a clean power fault. Specifically, at the time of crashes, all I/O operations between the caching layer and the backend file system terminate immediately, and the caching layer cannot receive I/O errors from the backend device. In addition, all in-DRAM data and metadata that are related to the caching system and backend file system are not available after a crash. For the recovery procedure, we perform the post-recovery process on both the caching layer and the backend file system. Also, we reuse the cached data and metadata in the caching layer for a fast recovery. The specifics of our crash-recovery procedure in our test framework are as follows: we use `unmount` and Open CAS's command `casctl stop` to perform synthetic crashes on the both caching layer and backend file system. We utilize `mount` and Open CAS's command `casctl start` to perform the caching layer post-recovery process. After that, we utilize `mount` to automatically invoke the post-recovery process of the backend file system.

Since we are only interested in the post-recovery process of caching layers and backend file systems, we apply with the simpler workload and cache mode settings than Section 5. We only choose the repeat write workload and two write cache modes, i.e., write-back mode and write-only mode for compatibility tests. The reason for choosing WB and WO modes is that these cache modes can cache the latest user data, i.e., dirty data, in the caching layer and leave the stale user data in the backend file system, which enables us to expose more interesting results such as crash inconsistency or durability loss during the post-recovery process. Specifically, We first restore a file on the backend file system and then apply repeat write (RW) workload in the write-back mode and write-around mode separately, in which we repeatably write a small portion of the cached data to the caching layer.

For the crash point setting, we simultaneously inject synthetic crashes on both the caching layer and the backend file system. In addition, since we attempt to activate the post-recovery process of backend file systems, we always inject synthetic crashes during the backend file system performing I/O operation. We utilize the cache operations `cleaning` to raise I/O requests to the backend file system.

For the check criteria, we flush the cached data to the backend file system and compare the recovered file system image with oracle, which is generated by performing the same workload without a crash. We also check log files and API returns in both Open CAS and the backend file system for error messages.

6.3 Results and Observations

The results of our compatibility tests are shown in Table 3. The table reports the results of compatibility tests with state-of-the-art reliability features of file systems. We now explain the results in detail in terms of the observations we made from our compatibility tests.

Observation 1: *For ext-2 with all configurations and ext-3 with the default configuration, performing the file system post-recovery process and flushing dirty data to the file systems keep crash consistency and durability.* In Table 3, this behavior is shown by the results (SR). Since the recovered caching

File system	Configuration	Cache mode	Result
ext-2	-	Write-Back	SR
		Write-Only	SR
	sync	Write-Back	SR
		Write-Only	SR
ext-3	-	Write-Back	SR
		Write-Only	SR
	writeback	Write-Back	FE
		Write-Only	FE
	orderd	Write-Back	FE
		Write-Only	FE
	journal	Write-Back	FE, LE
		Write-Only	FE, LE
ext-4	nodelalloc	Write-Back	FE
		Write-Only	FE
	writeback	Write-Back	FE
		Write-Only	FE
	orderd	Write-Back	FE
		Write-Only	FE
	journal	Write-Back	FE, LE
		Write-Only	FE, LE
btrfs	-	Write-Back	FE, LS
		Write-Only	FE, LS
xfs	-	Write-Back	FE, LS
		Write-Only	FE, LS
	wsync	Write-Back	FE, LS
		Write-Only	FE, LS

Table 3: **Compatibility test results of caching layers.** The table shows the results of caching layer compatibility experiments with state-of-the-art file systems and their reliability configurations. SR represents end-to-end file systems fully recovered with a slow post-recovery process; FE represents flushing dirty data to file systems incurred an I/O error; LE indicates end-to-end file systems suffered from durability loss with an I/O error; LS indicates end-to-end file system suffered from durability loss silently. Blank cells mean the default configuration settings.

layer maintains the complete and up-to-date user data, including the related metadata, flushing dirty data to the recovered file systems can help file systems to keep up-to-date and maintain crash consistency. One minor issue is that during the file system post-recovery process, the time-consuming checking task, i.e. fsck, is performed and incur a slow recovery process. It finally makes Open CAS "fast" recovery guarantee impossible.

It seems the solution for this incompatibility issue is simple which just requires file systems to degrade their reliability features and omit to check entire file systems. However, there is a dilemma to do so. Actuarily, not all cache modes cache the complete and up-to-date user data so flushing dirty data to backend file systems are not always helpful for keeping file system crash consistency. For example, in write-around mode and write-invalidate mode, the caching system may or may not cache the up-to-date user data in the caching layer, which depends on workload and data access pattern. For these cache modes, degrading the reliability features and omitting internal data consistency checking can potentially cause the file system crash inconsistency.

Observation 2: *For ext-3 and ext-4 in journaling-related configurations, xfs, and btrfs, performing the post-recovery process and flushing the dirty data to the file systems incur I/O error.* In Table 3, this behavior is shown by the results (FE). For the file systems with journaling-related configurations, if a crash causes an incomplete journal transaction, the file systems will automatically switch to read-only mode in post-recovery process. As a result, flushing the dirty data to the recovered file systems incurs I/O error. Naively forcing the file systems to the read-write mode incurs a similar dilemma situation which we discussed in Observation 1.

Observation 3: *For ext-3 and ext-4 in journal configuration and xfs, performing the post-recovery process and flushing the dirty data to the file systems incur durability loss with an error.* Similar to Observation 2, ext-3 and ext-4 in journal configuration and xfs automatically switched into read-only mode when they were remounted to the caching system due to the incomplete journal transaction. However, forcing these file systems to read-write mode still cannot solve all compatibility issues. Different from Observation 2, ext-3 and ext-4 in the journal configuration and xfs write both data block and the related metadata, which are normally larger than 4K `caching line` size, to the journal. During the post-recovery process, if the journal transaction is complete, the data block and metadata in the journal are checkpointed to its final in-place block location. Otherwise, the incomplete transaction is discarded. On the other hand, Open CAS only guarantees data transaction at the grain of `caching line` size. By setting the dirty bits for each `cache line` , Open CAS can identify whether a `cache line` is successfully synchronized to the backend file system or not. Furthermore, during the post-recovery process Open CAS doesn't redo the last failed cache operation, i.e., `cleaning` in our case. Since the flushed dirty `cache line` didn't include the complete journal transaction, the file systems discarded the flushed data and raised an I/O error. Although Open CAS persisted the complete journal transaction data in the caching layer, part of them had been marked as a clean state i.e., setting the dirty bit to zero, and cannot be flushed to the backend file system again but be evicted by `eviction` cache operation which finally cause durability loss.

Observation 4: *For btrfs, performing the post-recovery process and flushing the dirty data to the file system incur durability loss without an error.* Btrfs is a copy-on-write file system that writes to the same block only once except for the superblock which contains root-node information. Specifically, btrfs first writes to a log tree to record changes and then writes the data block and any related metadata to new locations and updates the roots in its superblock. In case of a crash, btrfs effectively reverts the contents of the data block and metadata back to its old state. Similar to Observation 3, btrfs simply discards the incomplete user data and metadata during the post-recovery process. Flushing the rest data blocks and metadata cannot help btrfs to maintain crash consistency and keep user data up-to-date. However, unlike Observation 3, flushing the dirty data to btrfs does not return an error which finally causes a silent durability loss.

In summary, Open CAS fails to be compatible with the

diverse reliability features of state-of-the-art file systems. The incompatibility issues are various from minor issues, e.g., performance degradation to serious ones, e.g., durability loss. The results also indicate the more rigid reliability features may suffer from the more serious incompatibility issues.

7 Beyond Open CAS

In addition to Open CAS, we have applied the same test framework used in Section 5 to a simpler caching system, NVCache [15]. NVCache is an approach that leverages a non-volatile main memory (NVMM) as a write cache to improve the write performance of legacy applications. Similar to Open CAS, NVCache intercepts and redirects in user-space I/O function calls to achieve this transparency. Also, NVCache guarantees data durability once returning an acknowledgment to users. Unlike Open CAS, NVCache uses log structures for persisting `cache line` and its metadata. In each write request, NVCache appends the new user data at the end of `cache line` area and directly embeds a commit flag within each `cache line`. The commit flag indicates whether `cache line` data is successfully committed to the caching layer or not. In order to handle non-committed writes left in the caching layer after a crash, NVCache firstly saves user data to `cache line` and then updates the corresponding commit flag from non-committed to committed.

Our initial results indicate that NVCache is also vulnerable to system crashes. As shown in table 4, NVCache gracefully recovers data from system crashes for the write cache after returning an acknowledgment. Moreover, thanks to the log structure to `cache line`, NVCache maintains crash consistency to the caching layer if a crash happens during cache miss and hit which is better than Open CAS. However, similar to Open CAS, NVCache returns bad data to users if a crash happens during the cache operation `cleaning`, the only implicit cache operation in NVCache. In addition, we see the significant throughput collapses when performing the cache operation `cleaning` which indicates the generalization of the workload-oriented test for identifying the running state of implicit cache operations.

In summary, so far we have studied two examples: Open CAS, a complex block-level caching system with techniques to maintain crash consistency, and NVCache, a simpler block-level caching system with few mechanisms to provide extra reliability. Both are vulnerable to system crashes. In addition, since OrthusCAS [50] was integrated with Open CAS with minor modifications for performance optimization, we believe most of our observations of Open CAS should also apply to OrthusCAS. It seems that regardless of the complexity of the caching system and the amount of machinery used for crash consistency, implicit cache operations are still a bottleneck for crash consistency, and systematical crash-consistency tests are necessary for various caching systems. In addition, reliability feature co-design on caching layers and backend file system is necessary for keeping crash consistency.

Cache mode	Workload	Crash point	Result 1	Result 2
Write-Cache	sequential write	cache miss	R	C
		cache cleaning	-	B
	repeat write	cache hit	R	C

Table 4: **Crash-consistency test result to NVCache.** Each cell indicates whether NVCache was able to recover data from a crash (R), whether NVCache maintains crash consistency but did not successfully persist user data (C), and whether NVCache returned bad data to the user (B). Blank cells mean that the result was not applicable.

8 Related Work

There is a lack of related work on crash-consistency tests to block-level caching systems. In addition, previous work on the crash-consistency tests to end-to-end file systems simply models the crash behavior of caching layers as an in-DRAM page cache in which all cached data is lost in case of a crash. We compare our work with ones on file systems and databases. Crash-consistency test frameworks and fault injection techniques have been widely used to analyze the robustness and find bugs in file systems and databases [23, 29, 36, 44, 55, 56] For example, Crashmonkey [29] exhaustively performs file system crash-consistency tests within a bounded space, a heuristic pruning rule based on the bug study of real file systems. Hydra [23] mutates both file systems image and operation sequences when performing feedback-oriented fuzzing to identify crash consistency bugs. Block Order Breaker (BOB) [36] creates crash states from recorded I/O to show that different file systems persist file-system operations in significantly different ways. The Application-Level Intelligent Crash Explorer (ALICE), explores application-level crash vulnerabilities in databases, key-value stores, etc. The logging and replay framework from Zheng et al. [56] is focused on verifying ACID guarantees of databases, works on iSCSI disks, and uses four dedicated manual workloads.

However, our work on Open CAS is the first comprehensive crash consistency analysis of the block-level caching system that covers carefully controlled crash-consistency tests to caching layers and compatibility tests with backend file systems.

9 Conclusion and Discussion

In this paper, we analyzed a representative block-level caching system, Open CAS, to study the reliability of the persistent caching layer in case of system crashes. We introduce a novel approach the workload-oriented test to systematically perform crash consistency experiments to the caching layer and selectively perform integration test to the diverse reliability features of file systems. We presented our observations about Open CAS crash consistency and compatibility with the diverse reliability features of file systems.

Although these are always the tradeoffs one has to make for crash consistency, as one tier of the end-to-end file system, the block-level caching layer should provide strong crash consistency against crashes. We conclude by discussing some techniques that caching systems can use to increase system

reliability and compatibility.

Checksums in cache lines: Caching systems could protect the user data in the `cache line` by using checksums. For example, Open CAS could use a checksum inside `cache line` in the caching layer, update it on caching operation `update`, and verify the checksum on read-related caching operations or during the post-recovery process (Observation 3 in Section 5.3). However, this does incur an overhead in computation as well as some complexity in implementation.

Data synchronization during post-recovery process: To deal with the stale user data in the caching layer due to system crashes (Observation 4 in Section 5.3), a caching system could synchronize user data with backend file systems during post-recovery process.

Reliability-feature-aware post-recovery process: A caching system could be aware of the reliability features of the backed file system, and take a well-co-designed post-recovery process with them (Observation 1 - 4 in Section 6.3).

Acknowledgments

We thank the anonymous reviewers and our shepherd, Mai Zheng, for their insightful comments and feedback. We thank Kan Wu for the invaluable discussions. This work is supported by the National Key R&D Program of China (Grant No. 2022YFB4500302) and the National Natural Science Foundation of China (Grant No. 62202255). Youmin Chen is the corresponding author.

References

[1] Thomas E. Anderson, Marco Canini, Jongyul Kim, Dejan Kostić, Youngjin Kwon, Simon Peter, Waleed Reda, Henry N. Schuh, and Emmett Witchel. Assise: Performance and availability via client-local NVM in a distributed file system. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 1011–1027. USENIX Association, November 2020.

[2] Andrea C. Arpaci-Dusseau and Remzi H. Arpaci-Dusseau. Information and control in gray-box systems. In *Proceedings of the 18th Symposium on Operating Systems Principles (SOSP'01)*, pages 43–56, Banff, Canada, October 2001.

[3] Remzi H. Arpaci-Dusseau and Arpaci-Dusseau Andrea C. *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books, LLC, 1.00 edition, 2015. http://pages.cs.wisc.edu/ remzi/OSTEP/.

[4] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, , and Daniel Teuchert. Nautilus: Fishing for deep bugs with grammars. In *Symposium on Network and Distributed System Security (NDSS)*, 2019.

[5] Bcache. A block layer cache. https://bcache.evilpiepirate.org.

[6] Randal E Bryant and O'Hallaron David Richard. *Computer systems: a programmer's perspective*, volume 281. Prentice Hall Upper Saddle River, 2003.

[7] Cache (computing). https://en.wikipedia.org/wiki/Cache_(computing).

[8] Remy Card, Theodore Ts'o, and Stephen Tweedie. Design and implementation of the second extended filesystem. In *First Dutch International Symposium on Linux*, Amsterdam, Netherlands, December 1994.

[9] Open CAS. Cache configuration. https://open-cas.com/cache_configuration.html.

[10] Open CAS. Data integrity test to a power failure scenario. https://github.com/Open-CAS/open-cas-linux/blob/master/test/functional/tests/data_integrity/test_data_integrity_unplug.py.

[11] Open CAS. Open cache acceleration software. https://open-cas.github.io/.

[12] Youmin Chen, Youyou Lu, Bohong Zhu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Jiwu Shu. Scalable persistent memory file system with Kernel-Userspace collaboration. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 81–95. USENIX Association, February 2021.

[13] Compute express link: The breakthrough cpu-to-device interconnect. https://www.computeexpresslink.org/download-the-specification.

[14] David Domingo and Sudarsun Kannan. pFSCK: Accelerating file system checking and repair for modern storage. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 113–126. USENIX Association, February 2021.

[15] Remi Dulong, Rafael Piresz, Andreia Correia, Valerio Schiavoni, Pedro Ramalhetex, Pascal Felber, and Gael Thomas. Nvcache: A plug-and-play nvmm-based i/o booster for legacy systems. In *Proceedings of the 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 21)*, page 186–198. IEEE, September 2021.

[16] Alexandra Fedorova, Keith A. Smith, Keith Bostic, Susan LoVerso, Michael Cahill, and Alex Gorrod. Writes hurt: lessons in cache design for optane nvram. In *the*

13th Symposium on Cloud Computing (SoCC 22), pages 110–125, 2022.

[17] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. AFL++ : Combining incremental steps of fuzzing research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*, August 2020.

[18] Robert Hagmann. Reimplementing the cedar file system using logging and group commit. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles (SOSP '87)*, Austin, Texas, November 1987.

[19] Jun He, Kan Wu, Sudarsun Kannan, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. Read as needed: Building wiser, a flash-optimized search engine. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 59–73, February 2020.

[20] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.

[21] Dave Hitz, James Lau, and Michael Malcolm. File system design for an nfs file server appliance. In *Proceedings of the USENIX Winter Technical Conference (USENIX Winter '94)*, San Francisco, California, January 1994.

[22] Shengtuo Hu. A grammar-based custom mutator for afl++. https://github.com/AFLplusplus/Grammar-Mutator.

[23] Seulbae Kim, Meng Xu, Sanidhya Kashyap, Jungyeon Yoon, Wen Xu, and Taesoo Kim. Finding semantic bugs in file systems with an extensible fuzzing framework. In *Proceedings of the 27th Symposium on Operating Systems Principles (SOSP'19)*, pages 147–161, October 2019.

[24] Youngjin Kwon, Henrique Fingler, Tyler Hunt, Simon Peter, Emmett Witchel, and Thomas Anderson. Strata: A cross media file system. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP'17)*, pages 460–477, October 2017.

[25] Zhen Lin, Lianjie Cao, Faraz Ahmed, Hui Lu, and Puneet Sharma. When caching systems meet emerging storage devices: A case study. In *15th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 23)*, Boston, MA, July 2023.

[26] Linux. Udev. https://www.man7.org/linux/man-pages/man7/udev.7.html.

[27] Chris Mason. The btrfs filesystem, September 2007. https://www.oss.oracle.com/projects/btrfs/dist/documentation/btrfs-ukuug.pdf.

[28] Meta. A general purpose, write-back block cache for linux. https://github.com/facebookarchive/flashcache.

[29] Jayashree Mohan, Ashlie Martinez, Soujanya Ponnapalli, Pandian Raju, and Vijay Chidambaram. Finding crash-consistency bugs with bounded black-box crash testing. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation (OSDI'18)*, pages 33–50, 2018.

[30] Intel optane dc ssd series. https://www.intel.com/content/www/us/en/products/details/memory-storage/data-center-ssds/optane-dcssd-series.html.

[31] Open cas - who already uses ocf? https://opencas.github.io/ocf_intro.html#who-already-uses-ocf.

[32] The open group. The open group base specifications issue 7, 2018. http://pubs.opengroup.org/onlinepubs/9699919799/.

[33] Research on performance tuning of hdd-based ceph cluster using open cas. https://01.org/blogs/tingjie/2020/performance-tuning-ceph-cluster-using-open-cas.

[34] Elizabeth J O'neil, Patrick E O'neil, and Gerhard Weikum. The lru-k page replacement algorithm for database disk buffering. *Acm Sigmod Record*, 22(2):297–306, 1993.

[35] Thanumalayan Sankaranarayana Pillai, Ramnatthan Alagappan, Lanyue Lu, Vijay Chidambaram, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Application crash consistency and performance with CCFS. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 181–196, Santa Clara, CA, February 2017. USENIX Association.

[36] Thanumalayan Sankaranarayana Pillai, Vijay Chidambaram, Ramnatthan Alagappan, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. All file systems are not created equal: On the complexity of crafting Crash-Consistent applications. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 433–448, Broomfield, CO, October 2014. USENIX Association.

[37] Intel optane dimm. https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html.

[38] Tom Ridge, David Sheets, Thomas Tuerk, Andrea Giugliano, Anil Madhavapeddy, and Peter Sewell. Sibylfs: Formal specification and oracle-based testing for posix and real-world file systems. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP'15)*, pages 38–53, October 2015.

[39] Mendel Rosenblum and John Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, 10(1):26–52, February 1992.

[40] Abraham Silberschatz, Greg Gagne, and Peter B Galvin. *Operating system concepts*. Wiley, 2018.

[41] Muthian Sivathanu, Vijayan Prabhakaran, Florentina I. Popovici, Timothy E. Denehy, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Semantically-Smart disk systems. In *2nd USENIX Conference on File and Storage Technologies (FAST 03)*, pages 73–88, San Francisco, CA, March 2003. USENIX Association.

[42] Hyunsub Song, Shean Kim, J. Hyun Kim, Ethan JH Park, and Sam H. Noh. First responder: Persistent memory simultaneously as high performance buffer cache and storage. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 839–853. USENIX Association, July 2021.

[43] Intel optane ssd p5800x series. https://www.intel.com/content/www/us/en/products/docs/memory-storage/solid-state-drives/datacenter-ssds/optane-ssd-p5800x-p5801x-brief.html.

[44] Sriram Subramanian, Yupu Zhang, Rajiv Vaidyanathan, Haryadi S. Gunawi, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Jeffrey F. Naughton. Impact of disk corruption on open-source dbms. In *Proceedings of the 26th International Conference on Data Engineering (ICDE'10)*, pages 509–520, March 2010.

[45] Adan Sweeney, Doug Doucette, Wei Hu, Curtis Anderson, Mike Nishimoto, and Geoff Peck. Scalability in the xfs file system. In *Proceedings of the USENIX Annual Technical Conference (USENIX '96),*, San Diego, California, January 1996.

[46] Theodore Ts'o. Ext2/3/4 file system utilities. https://github.com/tytso/e2fsprogs., 2018.

[47] Theodore Ts'o and Stephen Tweedie. Future directions for the ext2/3 filesystem. In *Proceedings of the USENIX Annual Technical Conference (FREENIX Track)*, Monterey, California, June 2002.

[48] Stephen C. Tweedie. Journaling the linux ext2fs file system. In *The Fourth Annual Linux Expo*, Durham, North Carolina, May 1998.

[49] Kan Wu, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. Towards an unwritten contract of intel optane SSD. In *11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 19)*, Renton, WA, July 2019. USENIX Association.

[50] Kan Wu, Zhihan Guo, Guanzhou Hu, Kaiwei Tu, Ramnatthan Alagappan, Rathijit Sen, Kwanghyun Park, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. The storage hierarchy is not a hierarchy: Optimizing caching on modern storage devices with orthus. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 307–323, San Francisco, CA, February 2021. USENIX Association.

[51] Kan Wu, Kaiwei Tu, Yuvraj Patel, Rathijit Sen, Kwanghyun Park, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. NyxCache: Flexible and efficient multi-tenant persistent memory caching. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*, pages 1–16, Santa Clara, CA, February 2022. USENIX Association.

[52] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. An empirical guide to the behavior and use of scalable persistent memory. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 169–182, Santa Clara, CA, February 2020. USENIX Association.

[53] Michał Zalewski. American fuzzy lop, 2017. https://github.com/google/AFL.

[54] Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler. *The Fuzzing Book*. CISPA Helmholtz Center for Information Security, 2021. Retrieved 2021-10-26 15:30:20+02:00.

[55] Yupu Zhang, Abhishek Rajimwale, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. End-to-end data integrity for file systems: A ZFS case study. In *8th USENIX Conference on File and Storage Technologies (FAST 10)*, San Jose, CA, February 2010. USENIX Association.

[56] Mai Zheng, Joseph Tucek, Dachuan Huang, Feng Qin, Mark Lillibridge, Elizabeth S. Yang, Bill W Zhao, and Shashank Singh. Torturing databases for fun and profit. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 449–464, Broomfield, CO, October 2014. USENIX Association.

[57] Shawn Zhong, Chenhao Ye, Guanzhou Hu, Suyan Qu, Andrea Arpaci-Dusseau, Remzi Arpaci-Dusseau, and Michael Swift. MadFS: Per-File virtualization for userspace persistent memory filesystems. In *21st USENIX Conference on File and Storage Technologies*

(FAST 23), pages 265–280, Santa Clara, CA, February 2023. USENIX Association.

[58] Yuhong Zhong, Haoyu Li, Yu Jian Wu, Ioannis Zarkadas, Jeffrey Tao, Evan Mesterhazy, Michael Makris, Junfeng Yang, Amy Tai, Ryan Stutsman, and Asaf Cidon. XRP: In-Kernel storage functions with eBPF. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 375–393, Carlsbad, CA, July 2022. USENIX Association.

FiDe: Reliable and Fast Crash Failure Detection to Boost Datacenter Coordination

Davide Rovelli
Università della Svizzera Italiana & SAP SE

Pavel Chuprikov
Télécom Paris & Institut Polytechnique de Paris

Philipp Berdesinski
turba

Ali Pahlevan
SAP SE

Patrick Jahnke
turba

Patrick Eugster
Università della Svizzera Italiana

Abstract

Failure detection is one of the most fundamental primitives on which distributed fault tolerant services and applications rely to achieve liveness. Typical crash failure detectors resort to using timeouts that have to take into account the unpredictability in interaction times among remote processes, caused by resource contention in the network and in endhost processors. While modern (gray) failure detectors have improved in detecting a wide range of failures, the problem of prohibitively large and unreliable timeouts for crash failures still persists, hampering performance of both the failure detector themselves and modern µs-scale services sitting on top.

We propose a novel fully reliable failure-detector (FiDe) that can report the crash of a remote process in a datacenter within less than 30µs (7.2× faster than the current state of the art) with *extremely high reliability*, thanks to a ground-up design which provides stable end-to-end process interactions. By reliably lowering worst-case crash detection time, FiDe enables a class of algorithms that can be used to boost coordination services even in the absence of failures. We devise two novel, FiDe-based, highly efficient consensus protocols and integrate them into a key-value store and a synchronization service, improving throughput by up to 2.23× and reducing latency down to 0.46×.

1 Introduction

An increasing number of user-facing services are being developed as distributed applications, many running 24/7 in cloud datacenters leveraging cheap hardware parallelism.

Failure detection in datacenters. Alas failures are a common issue in all distributed systems, especially in cloud datacenters. Chiefly, processes can fail due to a variety of reasons, including hardware faults. Process failure detectors (FDs) are thus a common building block in fault-tolerant distributed systems, which include highly available core services underlying user-facing applications. FDs allow the cross-cutting concern of detecting and notifying process failures to be factored out into one module, encapsulating timeouts and more, used by various components and systems. Timeouts are at the center of the prominent *heartbeat* crash FD design which has distributed processes periodically send "I am alive" messages to each other, interpreting the absence thereof beyond a certain time limit as indication of failure.

Limits of current timeout-based detectors. FDs are fundamentally limited by the systems they are built on. Distributed systems, including datacenters, are commonly viewed as being completely asynchronous, which makes it in theory impossible to achieve reliable failure detection, i.e., failure detection without any false suspicions. In practice, any FD must eventually make some synchrony assumptions and resort to heartbeats/timeouts to detect the unresponsiveness of monitored processes. Timeouts can be easily set too short, leading to premature false suspicion of processes. Inversely, setting timeouts too large can hamper system performance by delaying reactions to actual failures which is unaffordable for the growing µs-scale application market [2, 46]. With several processes failing in short time without their timely removal or replacement, common systems relying on majority quorum voting can block as the number of correct processes drops below the necessary quorum size. Tradeoffs between safety and liveness can not be navigated reliably by applications using an FD built on top of standard communication and OS abstractions. A common mitigation strategy is to resort to error-prone manual deployment and resource reservation [35, 38, 83].

Multi-level and gray failure detection. Previous fast, reliable FDs described in literature [7, 24, 44] build on the concept, spearheaded by Falcon [45], of closely monitoring multiple layers of a system (e.g., application process, OS, VM) through a network of probes. Increasingly popular gray failure detectors [27, 49, 58] use similar techniques to very effectively detect subtle failures (e.g. deadlocks) which harm availability without processes crashing. While these approaches excel at addressing a large variety of failure scenarios, they increase

software and system complexity. In the context of crash failures this leads to additional observation overhead, a conundrum of having to "monitor the monitor", besides requiring the programmer to implement failure detection logic. These design side-effects can decrease reliability by increasing performance non-determinism in applications, often compensated for by actively killing monitored application/host to back up a false failure suspicion [20, 45].

FiDe. This paper proposes a different novel approach specialized for crash failure detection, based on the observation that existing crash FDs are limited by being realized as modules, not as components, and do not fully exploit precision and programmability of modern OSes and networking hardware. More precisely, our approach is *system-driven*: leveraging custom lean system support from the ground-up, our approach, dubbed FiDe[1], achieves ultra-low and ultra-stable latency process interactions (communication and reaction) in a subset of the system custom built for failure detection enabling reliable heartbeats, as opposed to prior approaches which are at best adapted to a given system (*system-tailored*). In short, this is achieved by a combination of techniques including highly stable OS-supported send/receive pipelines and traffic engineering (TE) to reserve communication resources on switches via a centralized controller upon pre-registration by applications. FiDe also caters for network failures, making the probability of a false positive (i.e., a late or missed heartbeat with no process crash) to be once in years of uptime. To be clear, the correctness and efficiency of FiDe/FiDe-based protocols depend on assuming synchrony. While all-encompassing synchrony is unfeasible, the probability of synchrony violations in FiDe's controlled environment is negligible for practical purposes thanks to its reliable system substrate enforcing the assumption. Moreover, our approach is *externally observed*: application processes are monitored by dedicated, uninterrupted FD processes which leverage insights already provided by the OS as opposed to burdening programmers to modify applications to make them *internally observed* and in the process making them perform non-deterministically.

The extremely high reliability and promptness of FiDe serve two main purposes: (1) provide a foundation for developing novel, more efficient and simpler distributed coordination primitives and (2) improve crash detection accuracy and performance of core services with high availability requirements as standalone or combined with a more comprehensive (e.g. gray) failure detector. In summary, this paper presents:

- the design of FiDe which is system-driven and externally observed, allowing it to be very reliable and timely, while remaining non-invasive.
- FiDe's implementation using a high-performance system substrate based on custom process and network isolation, traffic engineering, and network processing pipelines.
- the design and implementation of novel FiDe-based broadcast and consensus primitives for simple fast replication.
- the empirical evaluation of our prototype in a production datacenter of SAP. FiDe consistently detects failure within less than 30µs outperforming the fastest state of the art crash failure detector by 7.2×.
- the integration of FiDe-based algorithms into the Redis key-value store [62] and Zookeeper synchronization service [29] to increase throughput by 1.7× and 2.23× and reduce latency down to 0.46× and 0.57× respectively, in failure-free execution.

The paper is structured as follows. § 2 presents model and assumptions, and § 3 FiDe's design. § 4 outlines FiDe-based algorithms. § 5 presents implementation details. § 6 evaluates FiDe by comparison to prior art. § 7 contrasts FiDe with related work. § 8 concludes with final remarks. App. A details FiDe-based algorithms, and App. B the TE algorithm. in detail. App. C provides details on network failure probability.

2 Model and Scope

2.1 System and failure model

FiDe is designed primarily to detect *crash-stop process failures* in networked distributed systems, and for use in applications dealing with such failures. When not specified further, a "failure" refers to when a process drops from the process table of the underlying OS.

Gray failures [49], fail-slow [52], and partial software/process failures [50] are not directly targeted by FiDe. However, one of FiDe's intended use-cases is to be used in combination with more comprehensive failure detectors aiming to improve their remote failure detection.

Due to the need for communication in failure detection, and the possibility of communication failures being falsely interpreted as process failures, FiDe has to cater for *network failures*, manifesting by packet drops (i.e. generalization of switch and link faults). We address this by exploiting 2-fold physical redundancy in the network configuration and introducing a network failure recovery mechanism (see § 3.3). FiDe tolerates multiple network failures unless they occur in very quick succession within the recovery time (we denote such set of failures as *critical compound failure*). We do not consider Byzantine failures but design FiDe to serve as a high-performance component of FDs providing broader coverage, i.e., gray/Byzantine failure detectors. We assume that software bugs, which are omitted even by several works on Byzantine fault tolerance (BFT) as corresponding bugs would be triggered across replicas, are handled by formally verifying FiDe-based algorithms (cf. [12]).

[1] ("Fee-deh") Latin for "with trust", a declination of *fides* = "trust", "faith".

2.2 Properties

We revisited the FD primitive and designed FiDe from the ground up aiming for three salient properties:

Reliability: FiDe's primary goal is to approximate (*in practice*) the strong completeness and strong accuracy properties of Chandra&Toueg's failure detector classification [14]. In short, if a process fails all interested correct processes are notified (completeness), and only then (accuracy).

Timeliness: FiDe explicitly aims for µs range failure detection, going beyond simple "eventual" notification of actual failures to correct processes.

Non-intrusiveness: Unlike existing failure detectors [2, 24, 45, 76] that require application-specific code injection even for crash failures to guarantee prompt detection, FiDe embraces the separation of concerns principle. Applications should be able use it immediately out-of-the-box through a lean API.

FiDe ensures reliability and timeliness by establishing *stable*, *end-to-end timely interaction* among its processes. The key to achieving this is minimizing interference at both process and network level: FiDe reduces maximum jitter to a point where it becomes consistently so small relative to the latency, that, in practice, it can be assumed to be bounded. We acknowledge that, in general, all-encompassing synchrony is unfeasible and that there is a probability that a series of failures (i.e. critical compound failure) and delays could break FiDe's model assumptions and reliability property. FiDe is explicitly designed and system-driven to make such probability negligible for a "slice" of the datacenter (FiDe domain). We show an estimate of this probability to be once in years of uptime, orders of magnitude less likely than failures in commonly-believed reliable services such as TCP.

2.3 Targeted services

We target core services with high availability requirements, like replicated storage or databases and increasingly relevant µs-scale services [2, 10, 24, 38]. FiDe's primary goal is to boost common fault-tolerant coordination protocols, e.g., consensus, underlying such services, through reliable and fast crash failure detection (§ 4). While we could use the reliable system substrate provided by our design to support more complex services, we choose to specifically support only failure detection in order to preserve modularity, allowing FiDe to be tailored to custom coordination protocols and failover mechanisms. Our solution requires a minimal setup time, which is negligible compared to its uptime, and acceptable for such services. We assume that a number of resources can be dedicated to FiDe, which is the case for modern over-provisioned datacenters and is commonly enforced *manually* in current practice when deploying critical services [19, 83]. Our primarily targeted services are in practice limited in scale, with replication clusters typically around 3 to 5 nodes (e.g. Zookeeper advises 5 nodes for exceptional maintenance [83]), almost never above 9 as shown in previous work [2, 6, 24, 34, 39]. While FiDe's design does not prevent it from larger scale deployment as in blockchains, this paper does not explore that angle.

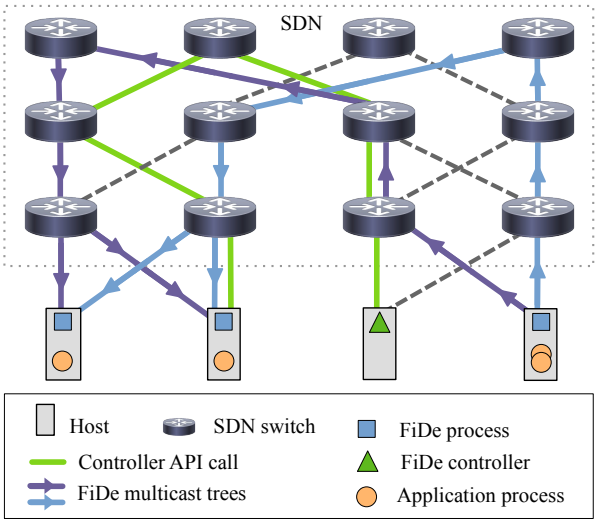


Figure 1: Simplified datacenter fat-tree topology showing FiDe's architecture and process interactions.

3 System Design

3.1 Architecture

Components. Fig. 1 illustrates the architecture of a typical FiDe deployment. It consists of a set of client processes (simply called *FiDe processes*), deployed alongside a distributed application, and a *FiDe controller* which dynamically configures the switches of a software-defined network (SDN) using our custom traffic engineering (TE) techniques (§ 3.3). FiDe processes are at most one per host, where they monitor one or more local *application processes* (§ 3.4). At the same time, each FiDe process can monitor and/or be monitored by other FiDe processes. Monitoring in a FiDe *cluster* need not be symmetrical, allowing for more flexibility and efficiency, covering application use cases where not all hosts play the same role (e.g. clients monitoring servers but not vice-versa).

Communication. FiDe processes communicate via a set of redundant directed multicast trees (cf. blue and purple lines in Fig. 1). The FiDe controller adjusts trees dynamically with cluster join request by allocating resources to each FiDe process according to the network topology, capacity, and number of monitoring processes (cf. § 3.3). The controller provisions and monitors such resources with TE techniques that cap jitter

```
struct resources {
  int pbsize;    // max piggyback size
  int timeout;   // min wrt request
}
// Downcalls from app to FiDe
↓ resources monitor(int appid, resources r);
↓ void unmonitor(int appid);
↓ bool join(appid);
↓ void quit();
↓ bool piggyback(char[] message);
// Upcalls from FiDe to monitoring apps
↑ void on_failure(int appid);
↑ void on_piggyback_recv(char[] message);
↑ void on_timeout_changed(int timeout);
// Upcalls from FiDe to monitored apps
↑ void on_monitor_req(int appid, int pbsize);
↑ void on_pbsize_changed(int pbsize);
```

Listing 1: Summary of FiDe's C API exposed to applications.

introduced by concurrent traffic. We use standard Ethernet links while prioritizing FiDe traffic and exploiting a custom TE-driven periodic IP-level multicast protocol, thus enabling reliable communication (see § 3.3).

Monitoring, allocation and piggybacking. Applications interact with FiDe through the API shown in List. 1. At a high level, applications can `join`, `quit` and `monitor` another application supplying a desired timeout value and a minimum piggyback message size. The FiDe controller the proceeds with the TE-driven allocation (§ 3.3), possibly relinquishing resources for processes already in the cluster (through callbacks `on_timeout_changed` and `on_pbsize_changed`) in case the current network capacity is not sufficient. After registration, a FiDe process uses the `on_failure` upcall to notify of a failure, which might be followed by a re-allocation find a more optimal resource assignment for remaining applications. Monitored applications can `piggyback` a limited amount of messages on FiDe's monitoring payload. The size of the piggyback payload (`pbsize`) is determined by the TE-driven allocation process to a maximum of 1418B. FiDe declines `piggyback` calls that try to exceed `pbsize` as it would break the guarantees provided by TE. Piggybacking enables seamless integration with gray/multi-level failure detectors, improving the propagation time of custom failures as we exemplify later. When applications and FiDe processes use distinct network paths, it could occur that the communication between applications is interrupted while respective FiDe processes monitoring such applications can still communicate, hence resulting in a "false negative". Piggybacking can prevent such scenarios since it ensures that the traffic of both application and FiDe processes goes through the same network path. Moreover, piggybacking exposes FiDe's reliability property to the monitored processes, bridging the gap between modularity (i.e., using FiDe as a general-purpose reliable crash FD module) and protocol-specific optimizations. We exploit piggybacking to define an efficient consensus primitive in § 4.

3.2 Reactive, uninterrupted processing

Task preemption is managed by the OS kernel scheduler which, generally, distributes running processes in a fair manner across all CPU cores. This is detrimental for time-sensitive tasks, in which preemption overhead is often the predominant cause of high processing tail-latency, as shown in previous work [21, 37, 53]. FiDe minimizes jitter through selective isolation and custom reactive network processing pipelines. We pin a FiDe process to a *dedicated CPU core*, set it to the highest active power state and isolate it from OS interrupts to provide uninterrupted execution. FiDe's Linux kernel module (LKM) exploits kernel space privileges to provide prompt monitoring of application processes (§ 3.4). It also uses an uninterrupted dedicated loop for ultra-stable periodic sending of heartbeat packets, bypassing the jitter-prone network stack. On the receiving side, we optimize packet delivery latency by reserving one specific queue in the network interface controller (NIC) to FiDe's traffic and throwing an interrupt request (IRQ) to FiDe's core at packet reception. This allows the separation of FiDe traffic and regular traffic both for reception and processing, allowing to maintain timeliness and low jitter even under heavy load. Finally, FiDe uses eXpress Data Path (XDP) [78] to intercept its packets as early as possible in the network stack and quickly deliver them, reducing network stack overhead on the receiving side as well. We experimentally found that using XDP and core isolation techniques account for the great majority (> 90%) of the jitter reduction, while other techniques are used to mitigate low, sporadic latency spikes. We fine tune low-jitter processing optimizations based on existing interaction latency breakdown studies [51, 82] and leave in-depth analysis contributions of each optimization for future work.

3.3 Fast-track, redundant networking

Fast-track multicast trees. FiDe minimizes network interference by establishing "fast-track" communication between FiDe processes, using TE techniques to manage and assign resources of links and switches within the limits of availability in order to avoid congestion upfront. This is made possible through an software-defined network (SDN)-assisted *periodic multicast protocol*. The FiDe controller enables the use of the protocol without making any explicit bandwidth reservations in the network, relying instead on reserving highest-priority queues and rate limiting at endhosts. The latter is a part of a contract between a sending FiDe process p_i^*, a set of receiving FiDe processes $\{p_j^*\}_{j \in J}$, and the network, where we use p_i^* instead of p_i to denote that p_i^* is a FiDe process and not an application process. Concretely, if p_i^* sends multicast

messages of maximum size $\sigma_{\max}(i,J)$ to the receiving processes with the minimum period $\pi_{\min}(i,J)$, then for every $j \in J$ the messages experience minimum latency $\lambda_{\min}(i,j)$ with maximum jitter $\delta_{\max}(i,j)$ when delivered to p_j^*. The prioritization ensures that the guarantees hold under arbitrarily heavy network load. TE techniques (see App. B) are used to compute and assign such parameters to a set of directed (single-source) *multicast trees*, each tree representing one-to-many network connections from a single sending process to a set of receiving processes. The trees in the set are vertex-disjoint (in internal nodes) providing redundancy. The TE algorithm takes as input i, J, $\sigma_{\max}(i,J)$, and the lower bound on $\pi_{\min}(i,J)$, and then constructs the new multicast tree set in such a way as to minimize the sum of maximum latencies among all trees, taking into account and possibly changing protocol parameters of already existing trees. The algorithm makes a crucial use of regularity of datacenter networks in tackling the challenge of vertex-disjoint tree set construction. We use the periodic multicast properties given by TE to determine the timeouts. FiDe exploits this within a simple heartbeat mechanism, where the heartbeat interval HB_i of every monitored FiDe process p_i^* is larger than or equal to the period $\pi_{\min}(i,J)$ assigned by TE to the corresponding set of multicast trees. Given a monitored *application* process p_i, an application process p_j monitoring p_i can set a timeout $TO(i,j)$ to $TO(i,j) = HB_{i'} + \lambda_{\min}(i',j') + 2 \cdot \delta_{\max}(i',j')$, where $p_{i'}^*$ is a FiDe process at p_i's host, and $p_{j'}^*$ is a FiDe process at p_j's host. Note, this ensures that all heartbeats sent by p_i are delivered to p_j before p_j's timeout expires. Our TE can be configured to use at most a given fraction of network bandwidth and queue capacity, or to restrict FiDe interaction to a desired subset of network links.

Redundancy and tree recovery. Network failures may cause a monitoring process to miss a heartbeat message, leading to a false positive. FiDe uses twofold physical redundancy with a tree recovery mechanism to increase robustness to such failures. First, the FiDe controller builds a pair of redundant trees for every cluster. A monitoring process detects a network failure (including slowdowns or suboptimal trees) when it receives a single heartbeat. It then informs the FiDe controller which computes an alternative tree to substitute the faulty one, contacts relevant switches to reserve priority queues and change routing tables. FiDe can be configured for the case when no alternative tree is available. One option is to simply notify the application and carry on, which might lead to compromising FiDe's properties (cf. §2.2) upon (additional) network failures. FiDe can be configured to also terminate gracefully without compromising reliability guarantees, by discontuing operation, which can for instance further involve the option of terminating protocols relying on FiDe before FiDe processes. This mechanism prevents any number of network failures from affecting reliability unless two or more affect both trees within the recovery time, i.e., in case of a critical compound failure. We give an estimated probability bound on such event occurring in §6.5.

Deployment notes. The simplest and most efficient deployment option is having all FiDe processes *as close as possible* while still maintaining redundancy (e.g. nodes linked to 2 ToR switches via redundant multicast trees of depth 1). This allows for minimal latency, network utilization/reservation as well as maximal network fault tolerance since the number of links and devices that can fail is minimal. FiDe can however be deployed over a larger topology if desirable, e.g., to follow application constraints, differentiate power supply or resources. In such deployments, FiDe continues to rely on the tree recovery mechanism to mitigate the probability of additional network failures. We evaluate the impact of deployments over larger networks in §6.5. Moreover, since the number of priority queues at the switches is limited, it can be desirable to share them with other services. FiDe TE can also account for these services and adjust periodic multicast properties accordingly, making use of the fact that high-priority queues often correspond to near real-time or control traffic, both of which are likely rate-limited already.

3.4 Failure detection

Domains. Distributed applications relying on a failure detector (e.g., replicated key-value stores or relational databases) are typically deployed as ordinary processes and therefore use system resources in the usual best-effort manner. We make no changes to this behavior and only focus on shielding FiDe from jitter sources. The resulting system, illustrated in Fig. 2, is split into two domains. We call "*best-effort domain*" the application environment where network and endhost resources are shared with other processes and "*FiDe domain*" FiDe's privileged system substrate. This system-driven design allows FiDe to achieve unprecedented timing consistency and differentiates it from previous performance-enhancing approaches (e.g. systems based on data plane development kit (DPDK), remote direct memory access (RDMA) and kernel-bypass [2, 17, 24, 80]).

Workflow. Fig. 2 shows the flow of events following different types of failures in our model. In *passive* failure detection (used to detect all failures that render the system unusable), a host fails causing both the FiDe process and the application process p_1 to fail as well ①. The periodic sending of heartbeat messages containing monotonically increasing values is therefore interrupted ②. Remote FiDe processes poll an extended Berkeley packet filter (eBPF) map containing the heartbeat value of each monitored process at every `timeout` interval and triggers a failure notification upcall ③ when they read the same value twice. Note that the scenario, where a FiDe process fails but its monitored application process does not, is prevented by the fact that FiDe runs in the kernel and is

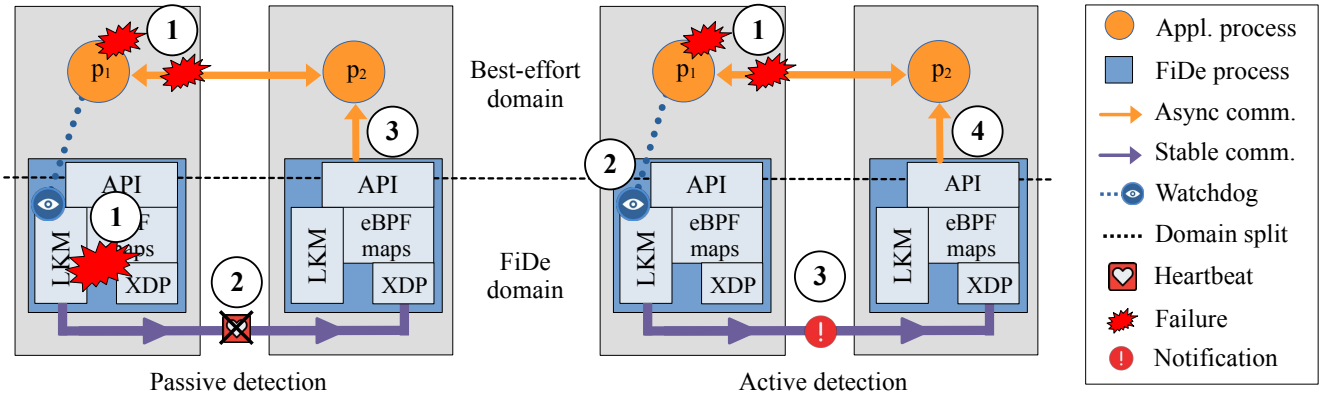


Figure 2: Failure detection flow in different scenarios. FiDe uses passive detection (left) when a FiDe process (i.e. the whole host) fails and active detection (right) when only an application process fails.

designed to not be subject to partial OS failures. In *active* failure detection (right scenario), an application process p_1 fails ① and becomes unresponsive to other applications. FiDe's LKM detects this promptly ② using a *watchdog* and immediately sends a failure notification ③ to all monitoring FiDe processes. Remote XDP hooks trigger the failure notification upcall ④ to the application.

Prompt detection with OS watchdog. FiDe uses an OS watchdog to monitor applications. The watchdog is implemented inside the LKM and uses a `kprobe` [47] registered on Linux's `do_exit` function. Every time the process id of the failed (i.e. exiting) process matches one of the application process ids monitored by FiDe, the OS watchdog logic in the `kprobe` pre-handler is executed just *before* the process actually fails. This design grants accurate detection, as the OS watchdog only reports a failure when it is officially "registered" by the system, prompt detection (technically negative latency) and solely relies on a single external observer.

External vs internal observation. Reliability and timeliness of a failure detector are often associated with its fine granularity: monitoring of as many as possible system components at multiple levels, from network nodes to application threads. This has been a common design principle in several systems [4, 50, 55], among which we find Falcon [45] of particular relevance, as it constitutes the base of several other systems [24, 27, 43, 44, 58, 76]. Such multi-level, system-tailored failure detectors are implemented by placing several probes (also called hooks or spies) in the software stack, a technique that we refer to as *internal observation*. This approach provides higher specificity: programmers can tailor detection probes to detect custom failures - including gray and partial failures - inside the application. However, this comes at the cost of increased complexity and intrusiveness for crash failures. system-tailored FDs have to take into account the failure of the probes themselves through a network of probes-monitoring-probes, introducing observation overhead which is often compensated for by selective killing and/or unreliable timeouts. Additionally, internal observation requires hundreds of lines of application-specific code for every monitoring probe [27, 43, 45, 49]. Thanks to its *external observation* approach which uses a single, external, timely observer (§3.4) on top of stable interactions, FiDe instead achieves reliable and non-intrusive detection of crash failures, trading off coverage for improved performance in order to boost services running on top (including gray FDs).

4 FiDe-based Novel Fast Services

4.1 Boosting consensus

A core contribution of this work is exploiting the extremely high reliability and speed of FiDe to improve the efficiency and simplicity of distributed services in *failure-free* executions. We use FiDe as a practical approximation of Chandra and Toueg's perfect failure detector $\mathcal{P}$ [14] to devise three novel algorithms: one reliable broadcast and two uniform consensus primitives which achieve consistent replication more efficiently than classical, quorum-based alternatives (Tab. 1). We give a concise overview of the algorithms below, please refer to App. A for in-depth specification and description. The algorithms were specified with TLA+ [74] and their correctness verified with TLC [79]. Algorithm TLA specifications and Redis implementation will be submitted to artifact evaluation can be found at: `https://github.com/swystems/fide-fd`.

Optimistic stabilizing reliable broadcast (OSRB). We start by introducing OSRB: a novel broadcast primitive which we later use in HSUC. OSRB optimistically implements BROADCAST using a multi-send, DELIVERS and buffers received messages, which it may need to retransmit later upon detecting a process failure. Classical lazy reliable broadcast

primitives require messages to be buffered infinitely because they might need to be re-transmitted eventually [13]. Since buffering in long-running applications can result in a prohibitively large memory footprint and possibly buffer overflow, OSRB introduces a novel performance improvement via an additional downcall called STABILIZE, which takes a message *id* as an argument. STABILIZE gives the possibility to apply a stabilization criterion to limit the messages buffered for lazy message relay upon failure. Ensuring that the stabilization criterion is satisfied is responsibility of a OSRB user, i.e., the protocol/application using OSRB must call STABILIZE only when every correct process has RECEIVEd a message (cf. App. A); otherwise, the *agreement* property of RB can be violated.

Hierarchical stabilizing uniform consensus. The first algorithm, hierarchical stabilizing uniform consensus (HSUC), solves uniform consensus (see [13] for specification) using a hierarchy among the processes and FiDe as a reliable failure detector. Processes that participate in consensus (e.g., key-value store replicas) are given an *id* to establish a hierarchy, in which the process with the highest ID corresponds to the first leader. During normal execution, a client sends a write request to the leader which starts the consensus instance, adopts the request as its proposal and multi-sends it to all other processes. Processes which receive the proposal from the leader adopt it as theirs and send an acknowledgment (ACK) to the leader. Once the leader collects ACKs from all processes, it disseminates its decision using OSRB and DELIVERs itself. Replicas also DELIVER once they receive the leader's decision. OSRB guarantees *Termination*, i.e., that the decision is delivered by all processes. Upon leader failure, FiDe notifies every process so that the next leader in the hierarchy can proceed with the algorithm. FiDe's accuracy enables an important algorithmic optimization: unlike quorum-based asynchronous algorithms which tolerate only a minority of crash process failures, HSUC can tolerate N-1 failures with same message complexity (cf. Tab. 1). The reception of the next round's decision defines the stabilization criterion of OSRB. Once this phase of the algorithm is reached, HSUC can safely call STABILIZE so that the message from the previous consensus will not be retransmitted by OSRB upon future failure detections, limiting the number of messages to be relayed to 2.

Fail-consistent uniform reliable broadcast We abstract FiDe's piggyback feature, i.e., the mechanism that allows applications to add a message into FiDe's heartbeats, into a primitive called fail-consistent uniform reliable broadcast (FCURB). FiDe's heartbeats have two important properties. First, every correct process either receives a heartbeat from a given process or no process does since multicast, redundancy and tree recovery guarantee uniformity (unless an extremely rare critical compound failure occurs, cf. §3.3, §6.5). Second, for any two FiDe processes p_i, p_j in a cluster, all heartbeats sent by p_i are delivered to p_j before p_j detects the failure of p_i. This is due to FiDe's upper-bounded interaction latency and the choice of large-enough timeouts (see $TO(i, j)$ in §3.3). Note that this order applies for both (1) passive failure detection since a crashed FiDe process is unable to send further heartbeats and (2) active failure detection since a FiDe process stops sending heartbeats after detecting a failed application (note that messages are not piggybacked into active notifications). Hence, applications using FCURB are guaranteed that all processes deliver a message before failure notifications. We call this property *fail-consistency* (refer to App. A for a formal specification).

Table 1: Complexity of FiDe-based algorithms in comparison with Raft and Zab in the failure-free (best) case. Note that Raft and Zab also tolerate process slowdowns.

	Message delays	Message complexity	Tolerated process failures
Raft	2	$O(N)$	$\lfloor \frac{N-1}{2} \rfloor$ (crash/slow)
Zab	3	$O(N)$	$\lfloor \frac{N-1}{2} \rfloor$ (crash/slow)
HSUC	3	$O(N)$	$N-1$ (crash only)
HUC	2	$O(N)$	$N-1$ (crash only)

Heartbeat uniform consensus. The second algorithm, heartbeat uniform consensus (HUC), is shown in Alg. 1. Here, all processes send proposal to a designated leader (line 5-7). The leader waits to collect proposals from all correct processes, i.e., all processes which are not signaled by FiDe, DETerministically chooses one of the proposals (line 12) and then broadcasts the decision using FCURB (line 11). Failures are detected by FiDe (line 15) causing HUC to update the *correct* set (line 16) and possibly triggering a new leader election in case the failed process is a leader (lines 18-21). FiDe's accuracy (cf. §2.2) guarantees that all processes consistently detect the failure of a leader, i.e., no two processes can have different leaders at one time. Unlike HSUC, HUC does not need an explicit ACK-ing round before calling DECIDE as the moment a proposal is sent to a new leader (line 21 or 7) it is already known that no decision from an earlier leader will ever be delivered. This is due to FCURB. To send a proposal to a newly elected leader, a process must first receive crash notifications from all previous leaders without receiving any decisions. Hence due to uniformity (all processes receive a message or no process does) and fail-consistency (messages are delivered before failure notifications) of FCURB, no other process will ever receive a decision from a previous leader. The lack of the ACK-ing round gives HUC an advantage over HSUC, Raft and other asynchronous consensus algorithms, as we will discuss shortly. Note that the throughput is limited by FiDe's periodic multicast protocol and the limit in message size imposed by the piggyback buffer, but as we show shortly, these limits are ample in practice.

Algorithm 1: Heartbeat uniform consensus (HUC). Uses fail-consistent uniform reliable broadcast (FCURB). Executed by every process p_i

```
1  correct ← {1,2...N}
2  leader ← MIN(correct)
3  proposals ← [⊥]^N
4  acks ← ∅
5  to PROPOSE(val)
6      proposals[i] ← val
7      SEND(val) to p_leader
8  upon RECEIVE(val) from p_j
9      proposals[j] ← val
10     acks ← acks ∪ {j}
11 upon acks ⊇ correct
12     val ←DET(proposals)
13     acks ← ∅
14     BROADCAST(val)
15 upon CRASH(j) from FiDe
16     correct ← correct \ {j}
17     proposals[j] ← ⊥
18     if j = leader then
19         leader ←MIN(correct)
20         if proposals[i] ≠ ⊥ then
21             SEND(proposals[i]) to p_leader
22 upon DELIVER(val)
23     DECIDE(val)
```

4.2 Advantages over traditional approaches

Simple and efficient crash-tolerant services. The advantages of using FiDe as a building block for fault-tolerant services are both improved performance and improved simplicity, a byproduct of FiDe's reliability and timeliness (§2.2). Tab. 1 compares complexity of the FiDe-based algorithms with Raft in the failure free case. Our algorithms can tolerate more crash failures with less resources and have same or comparable performance to Raft and Zab (Zookeeper Atomic Broadcast) [36], as we showcase in the evaluation shortly (§6). Traditional consensus algorithms are often so complex that they are hard to implement and verify (e.g. Raft was conceived as a simpler version of Paxos [41]). Such complexity often comes at the cost of safety which can be harmed by incorrect implementations and optimizations. FiDe provides a way of building and reasoning about simpler algorithms without compromising safety and performance.

Consensus and slow processes. By default, asynchronous consensus algorithms like Raft and Zab treat process slowdowns and crashes interchangeably. For instance, followers might time out on a leader process whether it is crashed or is simply delayed in computation, elect a new one, and carry on with the consensus execution given a majority of active processes. In comparison, HSUC and HUC do not cater for slowdowns and require participation of every non-crashed process, meaning that consensus execution will stall until a process recovers or crashes (and is detected by FiDe). The integration of FiDe with gray failure detectors (cf. §6.6) is an effective strategy to improve HSUC and HUC reactiveness to slow processes in such scenarios, while maintaining their performance and simplicity improvements. It is important to note that, while Raft and Zab-like algorithms tolerate slowdowns, they do not actively evict or replace faulty processes (whether slow or crashed) since they assume that process interactions can be arbitrarily delayed. This can degrade performance as the number of undetected faulty processes increases, resulting in a stall when a majority fails. Therefore, employing a gray failure detector is essential for improving the availability of both traditional and FiDe-based consensus algorithms. In addition, the use of formal verification is a good practice to effectively prevent software bugs such as deadlocks when developing critical consensus-based services.

5 Implementation Details

FiDe was implemented in 4032 lines of C code, split into LKM, XDP (eBPF) hooks and userspace API. We use the `libbpf` library (v0.5.0) and Linux kernel eBPF support as of v5.9.8 for XDP (Mellanox NIC `mlx4` driver). HSUC and HUC are implemented as Redis modules using the API version 1 in 910 and 810 lines of code respectively. Zookeeper variants were tested modifying configuration parameters according to the algorithmic complexity, without requiring internal modification. Both algorithms use a batch optimization to send multiple messages for a single consensus instance.

Fine-tuning is crucial to ensure stable communication; below we list some of the most relevant measures and settings. We isolate FiDe's core by assigning it to `isolcpus` kernel boot parameter and shielding the core from standard IRQ by setting their `smp_affinity` on other cores only, and from inter-processor interrupt IRQs by using `local_irq_save()` [51]. We maximize the power state of FiDe cores to avoid costly sleeping periods by setting them to the maximum C-state (C0). During testing, we found paging to be another source of jitter, which we addressed by using hugepages for our buffers. We also disable detrimental read-copy-update (RCU) stall detector warnings from kernel boot options to prevent the OS from interfering with the FiDe core which loop with interrupts disabled [51]. FiDe uses custom network processing pipelines to accelerate and stabilize packet transmission and delivery. The LKM manages its own socket kernel buffers (SKBs) [68] and uses an active pacemaker loop bound to FiDe's uninterrupted core to send heartbeat messages (and active notifications) at as precise time as possible. On the receiving side,

FiDe uses XDP hooks to bypass the network stack. These use eBPF maps of type `BPF_MAP_TYPE_ARRAY` for device `mmap`ing, allowing for efficient communication between kernel and user space. XDP was chosen over other kernel bypass methods for its superior stability in tests. We measure time using POSIX's `clock_gettime()`, with `CLOCK_MONOTONIC` parameter.

6 Evaluation

We evaluate FiDe by comparison with state-of-the-art services and applications, addressing four research questions:

RQ1: How stable is FiDe's underlying remote process interaction (cf. Reliability)?

RQ2: How quickly can FiDe detect failures (cf. Timeliness)?

RQ3: How well does FiDe scale in a real-world application (cf. Reliability, Timeliness)?

RQ4: By how much can FiDe accelerate replication (cf. Timeliness)?

RQ5: How do network faults and deployment impact FiDe (cf. Reliability, Timeliness)?

Finally, we discuss the impact of FiDe on the underlying system in terms of integration cost (cf. Non-intrusiveness), and overheads of processing and network.

6.1 Benchmark setup

Datacenter. We ran all our evaluation on 6 servers of a *production datacenter* of SAP hosting Arista 7280CR-48 [8] switches and servers with Intel Xeon E5-2680 v4 at 2.40GHz (28 cores, 56 threads), 1 TB RAM, Mellanox ConnectX-4 4x10 GbE [54] and Intel XL710 4x10 GbE [32] as commodity NIC, Ethernet interconnects. All the servers are connected via two redundant physical paths organized in a mini fat-tree topology. Every node runs our own customized version of Ubuntu 20.04 [75]. Our tests compare against a service/application that uses RDMA due to the relevance of the technology in modern systems and literature. For such applications we use RDMA over converged Ethernet (RoCE) [65]. While RDMA performs better on Infiniband networks we compare against RoCE as the majority of existing datacenters still relies on Ethernet. For multi-switch evaluation we use an additional cluster of 2 Cloudlab's [18] xl170 server, connecting its ConnectX-5 NICs to a number of Dell-s4048 type switches.

Comparison. We compare against three services chosen for their relevance in research and practice, and diversity:

Falcon [45], the seminal multi-level reliable FD (cf. §3.4);

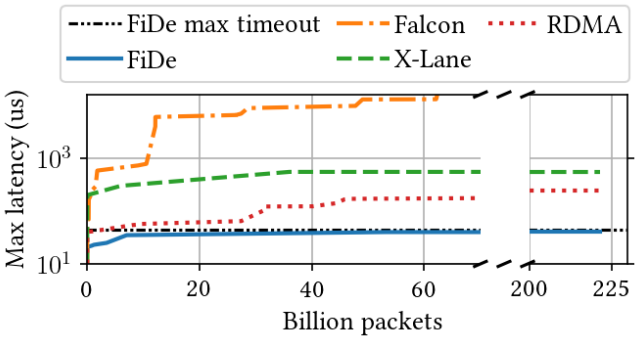


Figure 3: Maximum peer-to-peer latency evolution of the compared approached over 2.3 trillion packets

X-Lane [35], a state-of-the-art system designed to achieve ultra-latency and ultra-low relative jitter in datacenter interaction, used to build a passive FD;

uKharon [24], a state-of-the-art group membership protocol using RDMA and a Falcon-like multi-level FD. We compare against its FD only, referred to as uKharon-FD.

uKharon-FD can be seen as a re-implementation of Falcon on RDMA; to our knowledge, it reports the fastest failure detection in literature, as "*within few tens of microseconds*" [24] (without exact quantification). We compare FiDe primarily against uKharon-FD and X-Lane for performance, and use a re-implemented version of Falcon on commodity hardware as baseline. Note that, since Falcon can be customized to detect more complex types of (gray) failures whose detection is inherently slower, we compare only against Falcon's crash detection mechanism. We compare integrations of FiDe into Redis and Zookeeper against the originals. For Redis, we also compare against RedisRaft [64], an official module developed by RedisLabs using Raft for state machine replication (SMR).

6.2 Interaction stability (RQ1)

Our first experiments evaluate the possibility of achieving stable, bounded interaction latency (§3.2) in practice.

Methodology. We test FiDe's heartbeat mechanism by running a simple ping-pong protocol *over several weeks* with concurrent real-life traffic and logging the peer-to-peer latency for every packet. The dataset size amounts to 2.3 trillion consecutive packets which is 10× the amount used in previous work evaluating stable interactions [35] and over 1000× the amount used in works focusing on performance (vs. stability), e.g. [30, 60]. We found 20µs to be the lowest possible interval that provides stability with maximal throughput. All other works were tested with $>$ 100µs sending interval, favoring them over FiDe in terms of expected jitter. We use `stress-ng` [70] and `iPerf` [33] to generate periodic spikes of maximum CPU and network utilization. Priority queues are

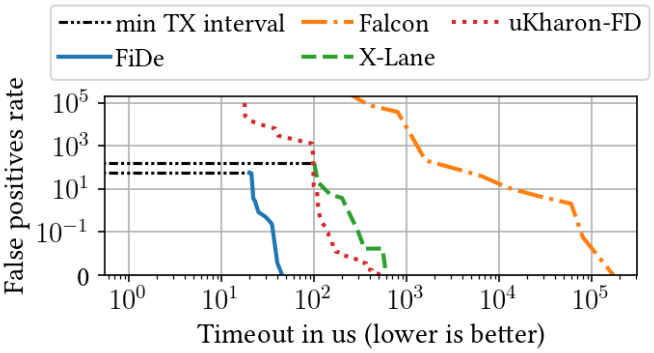


Figure 4: Failure detection accuracy for application crashes: false positives per minute (lower is better) at increasing timeout values. Timeout is meant as maximum time to wait before declaring a remote process unresponsive: a suitable value should yield no false positives (i.e., strong accuracy)

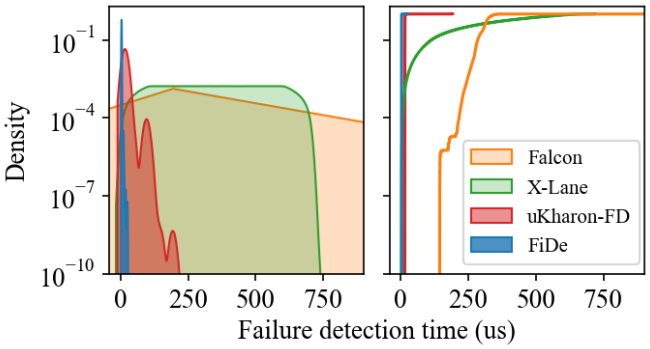


Figure 5: Failure detection empirical density distributions. The left plot shows the PDF (thinner is better) while the right plot shows the CDF. Each approach uses 10^7 datapoints taken from the respective empirical distributions.

reserved for FiDe. In this benchmark, we tested Falcon's communication between its client library and the spies in isolation (i.e., without failure detection logic). The same was done for uKharon-FD, that we refer to as "RDMA" since we consider only the communication and the packet processing layer.

Results. Fig. 3 shows that FiDe achieves upper-bounded interaction latency at $<$ 45µs (black dotted line), outperforming other approaches by over 5.4×. RDMA's latency is the closest to FiDe in terms of magnitude quickly reaching $\approx$ 50µs but peaking at 243µs after 70 billion packets. Regular "jumps" in latency including this last one suggest that despite having excellent average performance, RDMA by itself is not sufficient for stable interactions. We attribute the cause of this instability to network and process interference which FiDe actively aims to minimize. X-Lane shows better stability with latency ranging from 100µs to 550µs which is above the values reported [35], most likely because it was tested at much higher throughputs. Lastly, Falcon shows the most unstable latency, showcasing the importance of reactive processing (§ 3.2) which Falcon does not optimize. In fact, jumps in latency often correspond to an increase in the CPU load of the involved nodes. The graph line was cropped for readability.

6.3 Application failure detection (RQ2)

We define failure detection latency as the time between (1) an application process crash and (2) a failure notification delivery on a remote monitoring node. As (1) are (2) occur on different nodes, we divide latency into segments which are relative to each node to account for clock mismatch.

Methodology. We use optimal sending intervals and same settings as § 6.2. Since multi-level FDs such as Falcon and uKharon-FD have different detection "modes", we compare against the fastest of each.

Failure detection timeouts. Fig. 4 shows the failure detection accuracy in terms of number of false positives per minute. Every FD monitors an application and is set to send regular heartbeats at the smallest interval possible. False positives are obtained by counting how many times a local FD instance incorrectly signals a remote application process failure (i.e., no heartbeat) at a given timeout which is strictly larger than the heartbeat interval. Results show that FiDe can use timeouts larger than 48µs which is ca. 16× faster than uKharon-FD and X-Lane which reach perfect accuracy only at around 800µs.

Average and maximum failure detection times. The second requirement set for our design is low failure detection time (§ 2.2) which is required for fast process recovery in critical scenarios in production systems. Since we do not have access to a global clock in our system, we measure crash failure detection time based on the empirical joint distribution of the latency of the 3 steps involved in a successful failure notification delivery: (1) time from the manually injected crash to when it is picked up by the local FD instance; (2) communication latency from the local FD instance to the remote node; (3) latency for "delivery" from message reception at the remote FD instance to the remote monitoring application. We measure the *inspector-enforcer* crash detection latency for Falcon which consists in the time between the detection of an application crash and its communication to the local monitoring process trough inter-process communication (IPC).

Fig. 5 shows the empirical density distributions. The CDF in the right plot shows two almost vertical lines which flatten at the very top for FiDe and uKharon-FD, indicating that the vast majority of failures will be detected close to the average latency, with increases only around the 99.999th percentile. The PDF plot on the left of the figure provides a smoothed view of the same data which further highlights the differences between the two approaches. FiDe's failure detection is more deterministic and reliable, showing a low number of modest outliers contained in the narrow base of the curve. Tab. 2

Table 2: Summary of evaluated + state-of-the-art failure detectors, considering crash failures of A = application, OS = operating system, C = catastrophic (worst-case timeout). All values are in µs. *Slanted* numbers are taken from respective publications and blank values were not addressed.

Approach	A_{avg}	A_{max}	OS	$C_{timeout}$
FiDe	4.58	26.54	45.00	45.00
uKharon	17.39	193.56	*30.00*	*1000.00*
X-Lane	354.75	718.54	600.00	600.00
Falcon	496.29	169000.00	*204000.00*	3.00×10^8
Zookeeper	*3000.00*	*12000.00*		
Panorama	*2000.00*	*8000.00*		

summarizes average and maximum failure detection latencies of the compared approaches. FiDe achieves µs-range average failure detection outperforming the state-of-the-art uKharon-FD by 3.8× on average and by 7.2× at maximum (worst-case) latency, and X-Lane by 77× and 27× respectively. The table reports timeouts used to detect OS and catastrophic failures which uKharon-FD and Falcon address using specific probes. FiDe can use a general, worst-case timeout as low as 45µs, orders of magnitude below other approaches apart from uKharon-FD's OS timeout, reported as 30µs [24]. We believe that a much higher timeout would be needed in our (non-Infiniband) setup as suggested by previous measurements. Tab. 2 includes statistics from Panorama [27], a state-of-the-art gray failure detector, and Zookeeper's FD for reference.

6.4 FiDe for key-value stores and distributed synchronization (RQ3, RQ4)

FiDe's main goal is to be used as a reliable building block for efficient and simpler coordination protocols. To showcase the benefit of using FiDe beyond failure detection, we evaluate our novel algorithms proposed in § 4 integrated into Redis and Zookeeper. We refer to our variants as RedisFiDe (RedisFiDe)-HSUC/HUC and Zookeeper-HSUC/HUC respectively. We evaluate the SET request latency and throughput performance to address **RQ3** and **RQ4**. We do not evaluate GET requests since all approaches would simply return the requested value, adding no overhead to native performance.

Methodology. We use the official redis-benchmark [63] utility for Redis and custom Zookeeper native client [83] with the following settings: 50 concurrent clients, for a total of 1 million SET requests. For every approach, all requests are sent to the leader which is the only possible option for both RedisFiDe and RedisRaft. The latter uses batch optimization. In Zookeeper, we use Oracle quorum setting [83] and configure the ensemble to reproduce the complexity of our algorithms substituting Zab's atomic broadcast with consensus instances ordered by the leader. In HUC's implementations, the leader

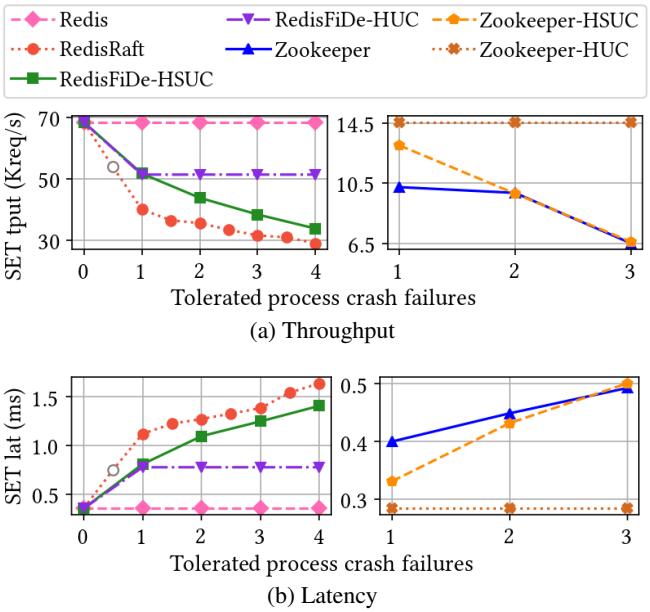


Figure 6: SET request performance of Redis and Zookeeper with different replication algorithms for given numbers of tolerated process failures (except for "Redis" reported as baseline). The values in between points on the x axis for RedisRaft are due to its use of a larger number of processes for tolerating the same number of failures cf. Tab. 1.

orders and batches concurrent requests which fit in the piggyback payload in a given FiDe heartbeat interval (at least multicast period and in the benchmarks $HB_i = 100\,\mu s$ for every p_i). We scale up to five and three tolerated failures, respectively 9 Redis instances and 7 Zookeeper instances, large sizes for a replication cluster (§ 2.3). Note that no method benefits from traffic prioritization: HUC-based services immediately return after piggybacking the decision message, hence only exploiting FiDe's accuracy and not latency enhancements. We evaluate HSUC-based services without traffic prioritization.

Results. Fig. 6a and Fig. 6b show throughput and latency of the compared approaches respectively. The metrics are plotted against the number of tolerated process failures corresponding to Tab. 1. The number of points equals to the number of processes. Since RedisRaft requires at least 3 processes to run, we put a "null" point as a placeholder for two processes (empty circle). All FiDe implementations almost always outperform the native performance, except for Redis which does not provide consistency guarantees. HUC has the highest throughput, 1.22× on average and 1.7× at maximum better than RedisRaft, and 1.71× on average and 2.23× at maximum for Zookeeper. It also has the lowest latency, respectively, reduced by 0.79× on average, 0.46× at maximum against RedisRaft, and 0.64× on average, and maximum 0.57× against Zookeeper. This is expected: the leader in HUC

uses only one heartbeat multicast to piggyback its decision directly, making FiDe's heartbeat interval (HB_i) and piggyback size limit (σ) the major factors affecting throughput. Considering $HB_i = 100\,\mu s$ for all p_i and $\sigma \approx 400\,B$, we can estimate the max throughput to be around $32\,Mbit/s$, largely sufficient for this benchmark. HSUC implementations show typical downtrend in both latency and throughput when the number of involved processes increases.

6.5 Network faults and deployment (RQ5)

We evaluate the network fault tolerance mechanism by estimating the probability of a critical compound failure, i.e., both redundant multicast trees failing withing a short time (cf. § 3.3), at different scales.

Methodology. We assume the widely-deployed multi-rooted fat tree topology [48] as defined by Al-Fares et al. [5], which uses multiple rooted trees to approximate a classical fat-tree (e.g., see Fig. 8 in App. C). We consider trees with $height = 1,2,3$ which we define as half the diameter of the tree. Note that multi-rooted fat trees always have an even number of nodes. We deploy FiDe in the multi-switch Cloudlab setup to analyze the effect of increased latency on previously measured application and OS failure detection time as well as RedisFiDe SET request latency. We apply the network and CPU load of § 6.2. We estimate the probability of *critical compound failure* by taking empirical failure probabilities from a popular study [22]; see App. C for a detailed breakdown.

Results. As expected (see Tab. 3), we observe an increase on the average application failure (A_{avg}) and OS failure (OS) detection times due to the added network distance, with latency overheads roughly equivalent to the additional switch forwarding latency of 5-10µs for larger fat trees. The impact on RedisFiDe HSUC and HSUC variations is instead minimal, due to the latency being dominated by Redis' request processing. Note that results for 1 switch vary compared to previous tests due to the different hardware setup (cf. Fig. 6b). The expected frequency of 1 critical compound failure every 22.7 years in the ideal deployment shows that FiDe is extremely likely to guarantee reliability for any realistic uptime. A back-of-the-envelope estimation of the probability of packet corruption in Ethernet + TCP (cf. App. C and [69]), widely regarded as reliable and used as such [29], shows that we could observe one packet corruption every 2.3 days at best, making FiDe more than 3 orders of magnitude more reliable.

6.6 FiDe in the bigger picture

Implementation costs. FiDe meets Non-intrusiveness: no application-specific failure detection code is required for the implementation of RedisFiDe modules apart from the necessary API calls, which amount to 6 lines of code including the imports. FDs using internal observation (§ 3.4) can have a much larger impact in similar contexts: applications based on Pidgeon [43] require 68 and 414 integration lines of code, while Falcon requires up to 159 for replication protocols [45].

Table 3: Consequences of deployment at larger scale. $freq_{\text{CCF}}$ is the expected frequency of a critical compound failure, A_{avg} and OS respectively denote average application and operating system failures, RF= average latency of RedisFiDe deployed on 2 processes (1 tolerated process failure).

Tree height	A_{avg} (µs)	OS (µs)	RF_{HSUC} (µs)	RF_{HUC} (µs)	$freq_{\text{CCF}}$
1 (ideal)	6.34	45	531	518	1 / 22.7 years
2	13.62	65	580	530	1 / 11.3 years
3	20.89	80	621	541	1 / 6 years

Combination with other FDs. FiDe can be combined out-of-the-box with other, more comprehensive (i.e. multi-level/gray) FDs through its API (§ 3.1). This would allow for performance gains and wider coverage, e.g., terminating processes that are out of control such as through deadlocks, getting the best of both worlds. For example, integrating FiDe into Panorama would improve Panorama's average crash failure detection by ca. 400×. Moreover, piggybacking probes' observations on FiDe's fast communication substrate could be exploited to accelerate remote detection of other types of failures, e.g., by more than 2 orders of magnitude for Panorama's reported propagation delay of 776.3µs [27] and more than 3 orders of magnitude for Falcon.

Overhead. FiDe's impact on the system consists in process and network resources reserved to enable reliable interactions (§ 5). We reserve one CPU core fully utilized during peak performance. This might impact performance differently depending on the processor's core count; in our case, it constitutes only $1/56$ (1.79%) of overall server processing power. The overhead introduced by the kernel watchdog, specifically `kprobes`, is in the order of ns thus negligible. In the network, we reserve highest priority queues (where needed) and consume bandwidth proportional to the periodic multicast protocol parameters (§ 3.3). In our setup, bandwidth consumption amounts to $\approx 50\,Mbit/s$ for links adjacent to endhosts, and *at most* $k \times 50\,Mbit/s$ inside the network, for k FiDe processes.

Mitigation and future work. Highly available core services targeted by FiDe are usually placed in a way to guarantee performance in practice, avoiding co-locating too many other compute intensive processes, achieving the effects of FiDe resource reservation by deployment [19, 83]. Furthermore, both processing and networking costs are mitigated by the efficiency of FiDe-based services, namely the smaller

number of nodes to tolerate same amount of failures as traditional quorum-based fault-tolerant services. We propose the following methods to mitigate energy consumption. Increasing FiDe's heartbeat interval, e.g. to hundreds of µs, would greatly reduce core utilization and only compromise failure detection time for passive failure detection. While we did not test kernel/system upgrades, FiDe simply treats them as failures and requires no particular upgrade strategy. We assume such updates are infrequent and also always provide a degree of backward compatibility. Combining FiDe with one of the state-of-the-art interference-aware CPU schedulers [21, 57], could provide stable processing without reserving a full core. Priority queues can be shared with other applications, as long as the resources needed are accounted by FiDe controller.

7 Related Work

FDs. Being able to detect failures promptly and reliably can tremendously improve distributed systems performance [4]. One of the most prominent works is Falcon [45], which aims to implement a reliable FD using *system-tailored* design with a web of layer-specific probes (spies). Falcon allows programmers to define failures but pays with intrusiveness, complexity, and observation overhead. Several FDs use a system-tailored design [7, 24, 43, 44], including gray/partial failure detectors [27, 49, 50, 52, 58] which enhance system availability by detecting subtle failures. FiDe uses a ground-up *system-driven* approach focusing on crash-stop process failures, gaining in reliability, usability, and detection timeliness.

Accelerating µs services. Many real-world distributed applications require operations at the µs scale [9, 10, 67], for which they rely on fast fault-tolerant distributed services built on top of FDs. State-of-the-art works here include Mu [2], Hovercraft [39], and DARE [60] for SMR, and uKharon [24] for group membership. Mu and DARE employ heartbeat-based FDs and assume stable interaction but fail to guarantee reliability respectively by not considering jitter sources. uKharon puts a stronger focus on reliability and implements a multi-level FD, achieving the fastest performance to our knowledge with a 50 µs full membership change. However, it fails to tackle the unpredictability of the network and optimizes heavily for RDMA on Infiniband, strongly limiting deployment. FiDe proposes a different approach with strong focus on reliability and timeliness, improving over uKharon by 7.2× on worst-case failure detection, on generic Ethernet.

Stable communication in datacenters. An ever-growing number of systems optimizes tail-latency for datacenter remote procedure calls through specialized networking stacks in hardware/software co-design [30, 71] and software only [11, 16, 21, 37, 38, 46, 57, 59, 61, 81]. These works achieve ns [30] and µs tail-latency through optimal endhost packet processing but do not consider sources of jitter in the network, hampering stability at the tail end. QJump [23] leads the way to achieve minimal, stable tail-latency in networks but does not consider jitter at the endhost, leading to the same issue. To our knowledge, the only comprehensive approach for end-to-end stable latency is given by X-Lane [35]. FiDe leverages network and process execution à-la X-Lane, but introduces more efficient and reproducible packet processing, critical novel design features including TE tailored for redundancy, reliability towards network failures and a ground-up, robust design tailored to failure detection (non-intrusive, prompt monitoring, active detection) offering dramatic improvements in throughput (50× from the values reported [35]) and reliability. Seminal work on deterministic distributed processes was introduced by DDOS [28]. However DDOS has significant overhead in remote process interaction, which FiDe's communication substrate could significantly reduce. Several prior works simply take stable interactions as given for, e.g., optimal weaker FDs [3], coordination primitives like leader election [66], or even synchronous BFT and blockchain protocols [1, 26] on top of a commodity software stack, without any concrete implementations to enforce such assumptions.

Clock synchronization protocols such as PTP [31] and TrueTime in Google Spanner [15] provide a very accurate global clock to support synchronous systems. Since gains in clock drift accuracy (ns-scale) are so small compared to interaction/communication latency (µs-scale), FiDe focuses on the latter leaving possible integration with accurate clock synchronization mechanisms for future work.

8 Conclusions

An increasing number of highly available, distributed applications are being deployed in datacenters. A key challenge in building such applications is detecting failures quickly and reliably, to ensure liveness and safety. Yet, existing failure detectors (FDs) fail to provide reliability at the µs scale, leading to inefficient, quorum-based coordination services built on top. Such FDs and are also often intrusive, making it hard for programmers to use them. FiDe fills this gap by providing reliable failure detection accelerated with respect to the state of the art by up to 7.2×, and 3.8× on average. To this end FiDe introduces a novel split system design, built ground-up, for timely interactions (communication and processing). Beyond failure detection, FiDe can be used as building block for simpler, faster algorithms. We propose 3 novel coordination primitives, accelerating replication in Redis and Zookeeper.

Acknowledgments

This work was supported by Swiss National Science Foundation (grants #192121, #197353), SAP, and Hasler Foundation. We thank our shepherd Atul Adya for his valuable feedback.

References

[1] Ittai Abraham, Kartik Nayak, and Nibesh Shrestha. Optimal good-case latency for rotating leader synchronous bft. In *25th International Conference on Principles of Distributed Systems (OPODIS 2021)*, 09 2021.

[2] Marcos K. Aguilera, Naama Ben-David, Rachid Guerraoui, Virendra J. Marathe, Athanasios Xygkis, and Igor Zablotchi. Microsecond consensus for microsecond applications, November 2020.

[3] Marcos K. Aguilera, Carole Delporte-Gallet, Hugues Fauconnier, and Sam Toueg. On implementing omega in systems with weak reliability and synchrony assumptions. *Distributed Computing*, 21(4):285–314, 2008.

[4] Marcos Kawazoe Aguilera, Gérard Le Lann, and Sam Toueg. On the impact of fast failure detectors on real-time fault-tolerant systems. In *Proceedings of the 16th International Conference on Distributed Computing*, DISC '02, page 354–370, 2002.

[5] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. In *Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication*, SIGCOMM '08, page 63–74, New York, NY, USA, 2008. Association for Computing Machinery.

[6] Mohammadreza Alimadadi, Hieu Mai, Shenghsun Cho, Michael Ferdman, Peter Milder, and Shuai Mu. Waverunner: An elegant approach to hardware acceleration of state machine replication. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 357–374, Boston, MA, April 2023. USENIX Association.

[7] Sebastian Angel, Mihir Nanavati, and Siddhartha Sen. Disaggregation and the application. In *12th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 20)*. USENIX Association, July 2020.

[8] Arista 7280R Series. https://www.arista.com/assets/data/pdf/Datasheets/7280R-DataSheet.pdf. Online; accessed 20-May-2025.

[9] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '12, page 53–64, New York, NY, USA, 2012. Association for Computing Machinery.

[10] Luiz Barroso, Mike Marty, David Patterson, and Parthasarathy Ranganathan. Attack of the killer microseconds. *Commun. ACM*, 60(4):48–54, mar 2017.

[11] Adam Belay, George Prekas, Mia Primorac, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. The ix operating system: Combining low latency, high throughput, and efficiency in a protected dataplane. *ACM Trans. Comput. Syst.*, 34(4), dec 2016.

[12] James Bornholt, Rajeev Joshi, Vytautas Astrauskas, Brendan Cully, Bernhard Kragl, Seth Markle, Kyle Sauri, Drew Schleit, Grant Slatton, Serdar Tasiran, Jacob Van Geffen, and Andrew Warfield. Using lightweight formal methods to validate a key-value storage node in amazon s3. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 836–850, New York, NY, USA, 2021. Association for Computing Machinery.

[13] Christian Cachin, Rachid Guerraoui, and Lus Rodrigues. *Introduction to Reliable and Secure Distributed Programming*. Springer Publishing Company, Incorporated, Heidelberg, Germany, 2nd edition, 2011.

[14] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, mar 1996.

[15] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hy Kuang, Richard Lagar-Cavilla, Richard Lloyd, Scott Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's Globally Distributed Database. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI '12)*, pages 251–264, 2012.

[16] Alexandros Daglis, Mark Sutherland, and Babak Falsafi. Rpcvalet: Ni-driven tail-aware balancing of µs-scale rpcs. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 35–48, New York, NY, USA, 2019. Association for Computing Machinery.

[17] DPDK, Data Plane Processing Kit. https://www.dpdk.org/, 2023. Online; accessed 20-May-2025.

[18] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya

Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. The design and operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 1–14, July 2019.

[19] Etcd hardware reccomendations. https://etcd.io/docs/v3.5/op-guide/hardware/#network. Online; accessed 20-May-2025.

[20] C. Fetzer. Perfect failure detection in timed asynchronous systems. *IEEE Transactions on Computers*, 52(2):99–112, 2003.

[21] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. Caladan: Mitigating interference at microsecond timescales. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 281–297. USENIX Association, November 2020.

[22] Phillipa Gill, Navendu Jain, and Nachiappan Nagappan. Understanding network failures in data centers: measurement, analysis, and implications. *SIGCOMM Comput. Commun. Rev.*, 41(4):350–361, August 2011.

[23] Matthew P. Grosvenor, Malte Schwarzkopf, Ionel Gog, Robert N. M. Watson, Andrew W. Moore, Steven Hand, and Jon Crowcroft. Queues Don't matter when you can JUMP them! In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 1–14, Oakland, CA, May 2015. USENIX Association.

[24] Rachid Guerraoui, Antoine Murat, Javier Picorel, Athanasios Xygkis, Huabing Yan, and Pengfei Zuo. uKharon: A membership service for microsecond applications. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 101–120, July 2022.

[25] Keqiang He, Junaid Khalid, Sourav Das, Aaron Gember-Jacobson, Chaithan Prakash, Aditya Akella, Li Erran Li, and Marina Thottan. Latency in software defined networks: Measurements and mitigation techniques. In *Proceedings of the 2015 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 435–436, 2015.

[26] Kaiwen Huang, Ronghui Hou, and Yingming Zeng. Lwsbft: Leaderless weakly synchronous BFT protocol. *Computer Networks*, 219:109419, 2022.

[27] Peng Huang, Chuanxiong Guo, Jacob R. Lorch, Lidong Zhou, and Yingnong Dang. Capturing and enhancing in situ system observability for failure detection. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 1–16, Carlsbad, CA, October 2018. USENIX Association.

[28] Nicholas Hunt, Tom Bergan, Luis Ceze, and Steven D. Gribble. Ddos: taming nondeterminism in distributed systems. *SIGARCH Comput. Archit. News*, 41(1):499–508, mar 2013.

[29] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. ZooKeeper: Wait-free coordination for internet-scale systems. In *2010 USENIX Annual Technical Conference (USENIX ATC 10)*. USENIX Association, June 2010.

[30] Stephen Ibanez, Alex Mallery, Serhat Arslan, Theo Jepsen, Muhammad Shahbaz, Changhoon Kim, and Nick McKeown. The nanopu: A nanosecond network stack for datacenters. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 239–256. USENIX Association, July 2021.

[31] IEEE Instrumentation and Measurement Society. IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems (IEEE Std 1588-2008), 2008.

[32] Intel XL710. https://www.intel.com/content/dam/www/public/us/en/documents/datasheets/xl710-10-40-controller-datasheet.pdf. Online; accessed 20-May-2025.

[33] iPerf tool. https://iperf.fr/. Online; accessed 20-May-2025.

[34] Zsolt István, David Sidler, Gustavo Alonso, and Marko Vukolic. Consensus in a box: Inexpensive coordination in hardware. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 425–438, Santa Clara, CA, March 2016. USENIX Association.

[35] Patrick Jahnke, Vincent Riesop, Pierre-Louis Roman, Pavel Chuprikov, and Patrick Eugster. Live in the express lane. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 581–597, July 2021.

[36] Flavio P. Junqueira, Benjamin C. Reed, and Marco Serafini. Zab: High-performance broadcast for primary-backup systems. In *2011 IEEE/IFIP 41st International Conference on Dependable Systems & Networks (DSN)*, pages 245–256, 2011.

[37] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. Shinjuku: Preemptive scheduling for µsecond-scale tail latency. In *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation*, NSDI'19, page 345–359, USA, 2019. USENIX Association.

[38] Anuj Kalia, Michael Kaminsky, and David Andersen. Datacenter RPCs can be general and fast. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 1–16, Boston, MA, February 2019. USENIX Association.

[39] Marios Kogias and Edouard Bugnion. Hovercraft: Achieving scalability and fault-tolerance for microsecond-scale datacenter services. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery.

[40] Maciej Kuźniar, Peter Perešíni, Dejan Kostić, and Marco Canini. Methodology, measurement and analysis of flow table update characteristics in hardware openflow switches. *Computer Networks*, 136:22–36, 2018.

[41] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, may 1998.

[42] Leslie Lamport, Robert E. Shostak, and Marshall C. Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4:382–401, 1982.

[43] Joshua B. Leners, Trinabh Gupta, Marcos K. Aguilera, and Michael Walfish. Improving availability in distributed systems with failure informers. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, NSDI'13, page 427–442, USA, 2013. USENIX Association.

[44] Joshua B. Leners, Trinabh Gupta, Marcos K. Aguilera, and Michael Walfish. Taming uncertainty in distributed systems with help from the network. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, 2015.

[45] Joshua B. Leners, Hao Wu, Wei-Lun Hung, Marcos K. Aguilera, and Michael Walfish. Detecting failures in distributed systems with the falcon spy network. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, page 279–294, 2011.

[46] Jialin Li, Naveen Kr. Sharma, Dan R. K. Ports, and Steven D. Gribble. Tales of the tail: Hardware, os, and application-level sources of tail latency. In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC '14, page 1–14, New York, NY, USA, 2014. Association for Computing Machinery.

[47] Linux Kernel Documentation - Kprobes overhead. https://docs.kernel.org/trace/kprobes.html#probe-overhead. Online; accessed 20-May-2025.

[48] Vincent Liu, Daniel Halperin, Arvind Krishnamurthy, and Thomas Anderson. F10: A Fault-Tolerant engineered network. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 399–412, Lombard, IL, April 2013. USENIX Association.

[49] Chang Lou, Peng Huang, and Scott Smith. Understanding, detecting and localizing partial failures in large system software. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 559–574, Santa Clara, CA, February 2020. USENIX Association.

[50] Chang Lou, Peng Huang, and Scott Smith. Understanding, detecting and localizing partial failures in large system software. In *NSDI*, volume 20, pages 559–574, 2020.

[51] Erik Rigtorp's low latency tuning guide. https://rigtorp.se/low-latency-guide/. Online; accessed 20-May-2025.

[52] Ruiming Lu, Erci Xu, Yiming Zhang, Fengyi Zhu, Zhaosheng Zhu, Mengtian Wang, Zongpeng Zhu, Guangtao Xue, Jiwu Shu, Minglu Li, and Jiesheng Wu. Perseus: A Fail-Slow detection framework for cloud storage systems. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*, pages 49–64, Santa Clara, CA, February 2023. USENIX Association.

[53] Sarah McClure, Amy Ousterhout, Scott Shenker, and Sylvia Ratnasamy. Efficient scheduling policies for Microsecond-Scale tasks. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 1–18, Renton, WA, April 2022. USENIX Association.

[54] Mellanox Connectx-4. https://network.nvidia.com/files/doc-2020/pb-connectx-4-lx-en-card.pdf. Online; accessed 20-May-2025.

[55] V Muthumanikandan, C Valliyammai, and B Swarna Deepa. Switch failure detection in software-defined networks. In *Advances in Big Data and Cloud Computing: Proceedings of ICBDCC18*, pages 155–162. Springer, 2019.

[56] Diego Ongaro and John Ousterhout. In Search of an Understandable Consensus Algorithm. In *2014 USENIX Annual Technical Conference*, USENIX ATC '14, pages 305–319, 2014.

[57] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. Shenango: Achieving high CPU efficiency for latency-sensitive datacenter

workloads. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 361–378, Boston, MA, February 2019. USENIX Association.

[58] Biswaranjan Panda, Deepthi Srinivasan, Huan Ke, Karan Gupta, Vinayak Khot, and Haryadi S. Gunawi. IASO: A Fail-Slow detection and mitigation framework for distributed storage services. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 47–62, Renton, WA, July 2019. USENIX Association.

[59] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. Arrakis: The operating system is the control plane. *ACM Trans. Comput. Syst.*, 33(4), nov 2015.

[60] Marius Poke and Torsten Hoefler. Dare: High-performance state machine replication on rdma networks. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '15, page 107–118, New York, NY, USA, 2015. Association for Computing Machinery.

[61] George Prekas, Marios Kogias, and Edouard Bugnion. Zygos: Achieving low tail latency for microsecond-scale networked tasks. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 325–341, New York, NY, USA, 2017. Association for Computing Machinery.

[62] Redis. https://redis.io. Online; accessed 20-May-2025.

[63] Redis-benchmark. https://redis.io/docs/management/optimization/benchmarks/. Online; accessed 20-May-2025.

[64] RedisRaft, consistent key-value store. https://github.com/RedisLabs/redisraft. Online; accessed 20-May-2025.

[65] RoCEv2. InfiniBand Trade Association, Supplement to InfiniBand Architecture Specification Volume 1, Release 1.2.1, September 2014.

[66] Nicolas Schiper and Sam Toueg. A robust and lightweight stable leader election service for dynamic systems. In *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*, pages 207–216, 2008.

[67] David Schneider. The microsecond market. *IEEE Spectrum*, 49(6):66–81, 2012.

[68] SKB - The Linux Kernel documentation. https://docs.kernel.org/networking/skbuff.html. Online; accessed 20-May-2025.

[69] Jonathan Stone and Craig Partridge. When the crc and tcp checksum disagree. *SIGCOMM Comput. Commun. Rev.*, 30(4):309–319, aug 2000.

[70] Stress-ng tool. http://colinianking.github.io/stress-ng/. Online; accessed 20-May-2025.

[71] Mark Sutherland, Siddharth Gupta, Babak Falsafi, Virendra Marathe, Dionisios Pnevmatikatos, and Alexandres Daglis. The nebula rpc-optimized architecture. In *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture*, ISCA '20, page 199–212. IEEE Press, 2020.

[72] Catching Corrupted OSPF Packets! - Blog. https://routingfreak.wordpress.com/2011/03/01/catching-corrupted-ospf-packets/. Online; accessed 20-May-2025.

[73] How both TCP and Ethernet checksums fail - Blog. https://www.evanjones.ca/tcp-and-ethernet-checksums-fail.html. Online; accessed 20-May-2025.

[74] The TLA+ home page. https://lamport.azurewebsites.net/tla/tla.html. Online; accessed 20-May-2025.

[75] Ubuntu 20.04. https://releases.ubuntu.com/20.04/. Online; accessed 20-May-2025.

[76] Fengwei Wang, Hai Jin, Deqing Zou, and Weizhong Qiang. Fdkeeper: A quick and open failure detector for cloud computing system. In *Proceedings of the 2014 International C* Conference on Computer Science & Software Engineering*, C3S2E '14, 2014.

[77] Xitao Wen, Bo Yang, Yan Chen, Li Erran Li, Kai Bu, Peng Zheng, Yang Yang, and Chengchen Hu. Ruletris: Minimizing rule update latency for tcam-based sdn switches. In *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*, pages 179–188, 2016.

[78] Cilium ebpf and xdp reference. https://docs.cilium.io/en/latest/bpf/. Online; accessed 20-May-2025.

[79] Yuan Yu, Panagiotis Manolios, and Leslie Lamport. Model checking tla+ specifications. In *Proceedings of the 10th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, CHARME '99, page 54–66, Berlin, Heidelberg, 1999. Springer-Verlag.

[80] Irene Zhang, Jing Liu, Amanda Austin, Michael Lowell Roberts, and Anirudh Badam. I'm not dead yet! the role of the operating system in a kernel-bypass era. In

Proceedings of the Workshop on Hot Topics in Operating Systems, HotOS '19, page 73–80, New York, NY, USA, 2019. Association for Computing Machinery.

[81] Irene Zhang, Amanda Raybuck, Pratyush Patel, Kirk Olynyk, Jacob Nelson, Omar S. Navarro Leija, Ashlie Martinez, Jing Liu, Anna Kornfeld Simpson, Sujay Jayakar, Pedro Henrique Penna, Max Demoulin, Piali Choudhury, and Anirudh Badam. The demikernel datapath os architecture for microsecond-scale datacenter systems. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 195–211, New York, NY, USA, 2021. Association for Computing Machinery.

[82] Noa Zilberman, Matthew Grosvenor, Diana Andreea Popescu, Neelakandan Manihatty-Bojan, Gianni Antichi, Marcin Wójcik, and Andrew W Moore. Where has my time gone? In *Passive and Active Measurement: 18th International Conference, PAM 2017, Sydney, NSW, Australia, March 30-31, 2017, Proceedings 18*, pages 201–214. Springer, 2017.

[83] Zookeeper administrator's guide. https://zookeeper.apache.org/doc/r3.1.2/zookeeperAdmin.html. Online; accessed 20-May-2025.

A FiDe-based Algorithms

The reliability of FiDe allows for the implementation of a particular set of distributed algorithms based on a reliable FD primitive, referred to as perfect FD ($\mathcal{P}$) in the widely know Chandra&Toueg classification [14]. While $\mathcal{P}$ is provably impossible to implement, we find the probabilities of FiDe to break its strong completeness and strong accuracy properties to be negligible, hence our motivation for the adoption of $\mathcal{P}$ for our novel algorithm.

Such algorithms are simple and, most importantly, reliable as they can tolerate a larger number of failures ($N-1$) compared to their pure asynchronous counterparts. In traditional timeout-based FDs, those gains may come at the of cost performance degradation due to the time spent waiting for failure notifications. Instead, the µs-scale detection times provided by FiDe enables the adoption of $\mathcal{P}$-based algorithms in practice, which in many cases can improve the system performance considerably even in failure-free runs.

A.1 Primitives overview

In this work, we propose one reliable broadcast primitive, OSRB (§ A.2), and two FiDe-based novel algorithms to solve the uniform consensus problem. The first algorithm, HSUC (§ A.3), relies directly on $\mathcal{P}$ and the OSRB primitive. The second algorithm, HUC (§ A.4), exploits the piggyback feature of FiDe to implement a heartbeat broadcast primitive which is optimal in message complexity for consensus. All algorithms assume an asynchronous system with a reliable FD primitive in the crash-stop process model. We present the uniform consensus specification taken from [13] below.

Termination: Every correct process eventually DECIDEs some value.

Validity: If a process DECIDEs v, then v was PROPOSEd by some process.

Integrity: No process DECIDEs twice.

Uniform agreement: No two processes DECIDE differently.

All the handlers are assumed to execute atomically, and, if multiple handlers are enabled at the same time, they are triggered in a fair manner.

A.2 Optimistic Stabilizing Reliable Broadcast

We start by presenting specification of a reliable broadcast primitive and our implementation of that primitive, dubbed OSRB, which we later use in our consensus algorithm.

Properties. The reliable broadcast has BROADCAST downcall and DELIVER upcall that must satisfy the following.

Validity: If a correct process p BROADCASTs a message m, then p eventually DELIVERs m.

No duplication: No message is DELIVERed more than once.

No creation: If a process DELIVERs a message m from p, then m was previously BROADCAST by process p.

Agreement: If a message m is DELIVERed by some correct process, then m is eventually DELIVERed by every correct process.

Algorithm. Like other lazy reliable broadcast primitives [13], OSRB optimistically implements BROADCAST using a multi-send and buffering received messages, which it may need to retransmit later upon detecting a process failure via $\mathcal{P}$. The reception of a message for the first time triggers the DELIVER. As a novel performance improvement, OSRB introduces an additional downcall called STABILIZE, which takes a message *id* as an argument. STABILIZE gives the possibility to apply a stabilization criterion to limit the messages buffered in *retransmit* and therefore lazy message relay upon failure.

Stabilility: If any process STABILIZEs *id*, then every correct process has RECEIVEd *msg* s.t. $id = \text{ID}(msg)$.

Note, it is the responsibility of a user of OSRB to ensure that the stabilization criterion is satisfied as otherwise, the *agreement* property can be violated.

Algorithm 2: Optimistic stabilizing reliable broadcast (OSRB). Executed by every process p_i

```
1  correct ← {1,2,...,N} /* set of process ids */
2  retransmit ← {} /* set of messages */
3  delivered ← {} /* set of message ids */
4  to BROADCAST(msg)
5      for p_j ∈ correct do
6          SEND(msg) to p_j
7  upon RECEIVE(msg)
8      if ID(msg) ∉ delivered then
9          retransmit ← retransmit ∪ {msg}
10         DELIVER(msg)
11         delivered ← delivered ∪ {ID(msg)}
12 upon CRASH(j) from P
13     correct ← correct \ {j}
14     for k ∈ correct do
15         for msg ∈ retransmit do
16             SEND(msg) to p_k
17 to STABILIZE(id)
18     if ∃msg ∈ retransmit | id = ID(msg) then
19         retransmit ← retransmit \ {msg}
```

Correctness OSRB keeps track of the delivered messages in the *delivered* variable to guarantee *No duplication* since a process can receive messages it has already seen upon failure detection. The *Validity* and *No creation* properties are guaranteed by the reliable links abstraction and the fact that we send to all correct processes. The *Uniform agreement* property is derived from *Validity*, $\mathcal{P}$'s accuracy, and the message retransmission upon failure detection when STABILIZE has not been called for that message's id. If STABILIZE has been called, then it is the stabilization criterion that ensures *validity*. We demonstrate a concrete example of the use of stabilization in the HSUC Alg. 3.

A.3 Hierarchical Stabilizing Uniform Consensus

Alg. 3 called hierarchical stabilizing uniform consensus (HSUC) solves uniform consensus using a hierarchy among the processes, a reliable FD detector and the previously defined OSRB primitive.

Algorithm. The algorithm is re-adapted from the hierarchical uniform consensus algorithm in [13], originally taken from [42]. It assumes asynchronous communication, reliable links, unique message ids and sequential execution of consensus instances. The latter means that at any given process,

Algorithm 3: Hierarchical stabilizing uniform consensus (HSUC). Uses OSRB. Executed by every process p_i

```
1  correct ← {1,2,...,N} /* set of process ids */
2  leader ← MIN(correct)
3  proposal ← ⊥
4  proposer ← ⊥
5  acks ← ∅
6  lastid ← ⊥
7  to PROPOSE(v)
8      if proposal = ⊥ then
9          proposal ← v
10     if i = leader then
11         for j ∈ correct do
12             SEND(v) to p_j
13 upon RECEIVE(v) from p_j
14     if proposer = ⊥ or j ≥ proposer then
15         proposal ← v
16         proposer ← j
17         SEND(ACK) to p_j
18 upon RECEIVE(ACK) from p_j
19     acks ← acks ∪ {j}
20 upon acks ⊇ correct
21     BROADCAST(proposal)
22     acks ← ∅
23 upon CRASH(j) from P
24     correct ← correct \ {j}
25     if j = leader then
26         leader ← MIN(correct)
27         if i = leader and proposal ≠ ⊥ then
28             for k ∈ correct do
29                 SEND(proposal) to p_k
   upon DELIVER(v)
30     STABILIZE(lastid)
31     lastid ← ID(v)
32     DECIDE(v)
33     proposal ← ⊥
```

between any two PROPOSE invocations, there is exactly one DECIDE event. We also assume two kinds of implicit behaviors for all handlers in HSUC. First, a process buffers messages corresponding to future consensus instances until the process has made a decision for all the previous instances. Second, it ignores messages from all the consensus instances for which it has already decided. The consensus instance can be identified by attaching and propagating a consensus instance number inside every message. HSUC can tolerate up

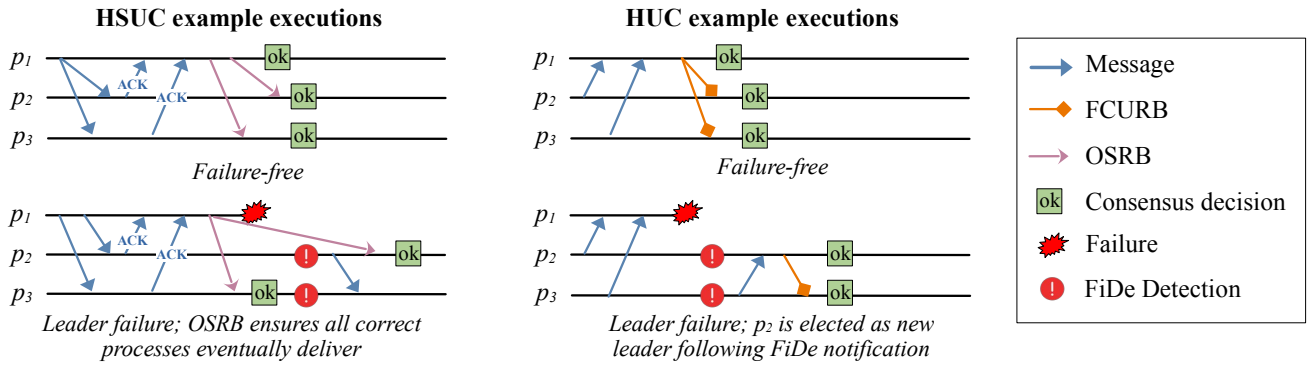


Figure 7: HSUC and HUC execution diagrams. Black lines represent timelines for each process increasing from left to right. HUC enforces message-then-failure ordering through FCURB (bottom right diagram) while HSUC allows the safe delivery of a decision message through OSRB even after a failure detection event (bottom left diagram).

to $f = N - 1$ failures where N is the number of total processes in the system.

The algorithm is round-based, with every round lasting until at least one failure is detected. The leader of every round is deterministically chosen to be the correct process with the lowest id.

In failure-free runs, the algorithm requires only one round and uses three communication messages: ① the leader multi-SENDs its proposal to all correct processes, which then ② adopt the proposal as their own and SEND an ACK to the leader upon reception. ③ Once the leader has collected ACKs from all alive processes, it reliably disseminates the decision using the OSRB primitive. The algorithm goes through these same three phases if up to $f = N - 1$ non-leader processes fail.

If the leader of a round fails, all correct processes detect it and proceed to a new round in which a new deterministically elected leader starts multi-sending the proposal message (line 11 and line 28). Note that even if the leader fails after sending its decision, all the alive processes have their *proposal* set to the leader's so any future leader would broadcast the same proposal as the failed leader. If the leader decides on v and then crashes before its decision can reach any other process, any of the following leaders will propose v as it was adopted as own proposal.

The reception of the next round's decision defines the stabilization criterion of OSRB. Once this phase of the algorithm is reached, STABILIZE($lastid$) [Alg. 3, line 30] can be safely called so that the message from the previous consensus will not be retransimitted by OSRB upon future failure detections [Alg. 2, line 15].

Process diagrams in Fig. 7 depict example executions of HSUC. Upon failure of the leader, the OSRB primitive ensures that the decision message is eventually delivered. The delivery can occur even after another process detects a failure and proceeds with the leader election. HSUC ensures that correct processes which receive a decision message (p_3 in the bottom-left diagram) will not ACK future proposal by another leader preventing mismatched decisions.

Correctness. The *validity* property follows trivially from the algorithm and from the no-creation properties of communication abstractions. The *integrity* follows from our assumption that all the messages from earlier consensus instances are ignored once the decision is made. For *Termination*, it is sufficient to show that at least some *correct* process DELIVERs, for OSRB's agreement ensures that then every correct process DELIVERs and, hence, decides. Assume no correct process DELIVERs, consider then correct process p_{i*} with the lowest process id i^*. Eventually, due accuracy and completeness of $\mathcal{P}$, p_{i*} will become and always remain a leader. Right at that point (on line 11 or line 28), p_{i*} will multisend its proposal. Note that due to accuracy of $\mathcal{P}$, no process p_j with $j > i*$ will become a leader, and, hence, will multisend its proposal or set its *proposer* variable to j. Hence, no *proposer* variable will ever get set to $j > i^*$, implying that eventually all the correct processes will respond to p_{i^*}'s proposal triggering p_{i^*} to BROADCAST, and hence, DELIVER, contradicting our assumption. The fact that processes ignore messages from older instances does not hinder termination as advancing to the next instance is only possible after DELIVER, hence all correct processes continue to participate under our assumption that no correct process DELIVERs. To show *Uniform Agreement* we argue that all the BROADCASTs necessarily carry the same value. This is guaranteed by phase ① and ② of the algorithm that ensure that every process will propose hence decide on the same value even if a leader fails before delivering a decision. Specifically, the moment process p_i receives all the ACKnowledgements all the alive processes must have their *proposal* equal to that of p_i. First of all, no proposal has been sent by p_j, $j > i$ as no such p_j could have become a leader for i is still alive. Hence, all the processes

that have received p_i's proposal have ACKnowledged it and could not yet have changed their *proposal* value. We also need to show the *Stability* criterion of OSRB since it could be violated by an unsafe call to STABILIZE and violation of the *agreement*. Stability of *lastid* is guaranteed as if a process receives a decision, then a leader must have collected all the ACKs for the *current* consensus instance, meaning that all the correct processes have responded with those ACKs and they have decided in the *previous* instance on some proposal v with $\text{ID}(v) = lastid$ as otherwise ACK requests would be buffered instead.

Complexity. In the failure-free cases, HSUC uses 3 message delays and exchanges $O(N)$ messages. Each failure of a leader adds 2 additional delays and $O(N)$ messages for consensus and $O(N^2)$ for the OSRB retransmission upon failures [Alg. 2, line 15]. If we used a lazy RB instead of OSRB , e.g. "Lazy Reliable broadcast" in [13], the added complexity overhead of the RB would be $O(M \times N^2)$ where M is the number of messages sent in all previous instances of consensus. This performance improvement is achieved through message *stabilization* logic.

A.4 Heartbeat Uniform Consensus

The second implementation of uniform consensus that we propose is called heartbeat uniform consensus (HUC) and is shown in Alg. 4.

Overview. Once again we assume asynchronous communication and reliable links. HUC uses a leader-driven decision logic with a new leader elected deterministically by selecting the process with the smallest ID upon failure of its predecessor. Here FiDe is used both as $\mathcal{P}$ and as the means to piggyback decision messages. As for HSUC, we assume that messages pertaining to old consensus instances are ignored and future ones are buffered. This simple algorithm is meant to showcase the reduced logical and message complexity that can be achieved in failure-free executions when using FiDe as backbone of distributed systems. Enabling such reductions is arguably the most relevant contribution of this work.

Fail-consistent uniform reliable broadcast. We abstract FiDe's piggyback feature into a primitive called fail-consistent uniform reliable broadcast (FCURB). Since there is a limit on the message size for the piggyback method, we assume small message sizes. A practical implementation of this assumption is to reliably disseminate the messages in advance, using FiDe piggyback for message ids only. The properties of fail-consistent uniform reliable broadcast are:

Validity: If a correct process p BROADCASTs a message m, then p eventually DELIVERs m.

Algorithm 4: Heartbeat uniform consensus (HUC). Uses FCURB. Executed by every process p_i

```
1  correct ← {1,2...N}
2  leader ← MIN(correct)
3  proposals ← [⊥]^N
4  acks ← ∅
5  to PROPOSE(val)
6      proposals[i] ← val
7      SEND(val) to p_leader
8  upon RECEIVE(val) from p_j
9      proposals[j] ← val
10     acks ← acks ∪ {j}
11 upon acks ⊇ correct
12     val ←DET(proposals)
13     acks ← ∅
14     BROADCAST(val)
15 upon CRASH(j) from FiDe
16     correct ← correct \ {j}
17     proposals[j] ← ⊥
18     if j = leader then
19         leader ←MIN(correct)
20         if proposals[i] ≠ ⊥ then
21             SEND(proposals[i]) to p_leader
22 upon DELIVER(val)
23     DECIDE(val)
```

No duplication: No message is DELIVERed more than once.

No creation: If a process DELIVERs a message m from p, then m was previously BROADCAST by process p.

Uniform agreement: If a message m is DELIVERed by some process, then m is eventually DELIVERed by every correct process.

Fail-consistency: If process p DELIVERs a message from process q, then p has not yet received CRASH notification for q.

Validity, *uniform agreement*, and *fail-consistency* are guaranteed by FiDe's reliability: a heartbeat message m BROADCAST by p is guaranteed to be either (i) DELIVERed by all correct processes *and* confirming that p is still alive, hence *fail-consistency*; or (ii) being lost for all processes as a result of p failing before actually sending out m. *No creation* and *no duplication* are guaranteed by underlying guarantees of the links and the fact that no re-transmission are needed. We use FCURB to implement Alg. 4, heartbeat uniform consensus.

Algorithm. HUC uses two communication steps in failure-free executions: ① every process sends its proposal to the

leader, ② once the leader has collected a proposal from every correct process it deterministically decides using the DET function (Alg. 4, line 12) and then disseminates the decision by using FCURB. If the leader fails, every alive process correctly detects and proceeds with the next leader (line 19). Note that the delivery of a decision can never follow a failure detection event thanks to *fail-consistency* properties of our FCURB i.e., FiDe piggybacking mechanism. A leader can fail and not broadcast a decision message (scenario depicted in Fig. 7) or broadcast and immediately fail, in which case the decision is reliably delivered to all processes by FCURB..

Correctness. *Validity* follows from the validity of underlying communication abstractions, and *integrity* follows trivially from the assumption of ignoring old consensus messages after the decision is taken. *Termination* for Alg. 4 is guaranteed once *any* process DELIVERs due to FCURB's *uniform agreement*. Assume, similar to HSUC that never happens, and consider again the correct process p_{i^*} with the lowest i^*. Eventually every process would recognize p_{i^*} as a leader (due to $\mathcal{P}$'s completeness and accuracy) and, hence, will eventually send their proposals to p_{i^*} either on line 21 or on line 7, which would be collected by p_{i^*} and, hence BROADCAST, and hence DELIVERed. *Uniform agreement* follows easily once we show that only one successful BROADCAST every happens. To see that, assume that process p_i and p_j, $j > i$ both successfully broadcast, for that to happen, p_j should have received a proposal from some *correct* process p_k. That implies that p_k thought p_j was the leader and, hence, must have received a failure notification from p_i, but due to *fail-consistency* p_i must have DELIVERed p_i's decision first, and, hence, should have advanced to the next instance, a contradiction.

Complexity. In the failure-free cases, HUC uses 2 message delays and exchanges $O(N)$ messages. Each failure of a leader adds 1 additional delays and $O(N)$ messages for consensus. HUC has better performance than HSUC (Alg. 3) and other asynchronous with $\mathcal{P}$ algorithms in literature [13]. It also has equivalent or better performance than widely known partially synchronous consensus algorithms (e.g. Paxos [41], Raft [56]), but higher fault tolerance: Typical quorum-based algorithm can tolerate $f = N/2 - 1$ failures compared to $f = N - 1$ of the algorithms proposed in this section. The downside is that the throughput and message size are limited FiDe's periodic multicast protocol which may be a bottleneck in very large systems. To overcome limited message size, clients might need to broadcast messages to replicas in advance.

B FiDe Traffic Engineering Algorithm

The key responsibility of FiDe traffic engineering is to construct the multicast trees and derive the parameters of the *periodic multicast protocol*, while incorporating global view on the behavior of network nodes provided made available by SDN, namely bandwidth, latency and jitter of network devices. Latency and jitter parameters of the network subsystem at endhosts are estimated empirically using a purpose-built tool. Crucially for FiDe's reliability, there must be K multicast trees for the same sender/receivers pair and those tree must be *vertex-disjoint in their internal nodes* (leaves are always the same) providing necessary redundancy in case of link or switch failures; henceforth we use *vertex-disjoint* to mean "vertex-disjoint in internal nodes".

To decouple optimization process from ensuring latency guarantees, we employ a similar high-level approach as in [35], namely, first, we find the most promising set of multicast trees, and only then we compute the protocol parameters. For the latter, we take the *adjustment* phase from [35] and extend it to account for redundant trees: given a set of vertex-disjoint trees $\mathcal{T}$ for sender p_i^* and receivers $\{p_j^*\}_{j\in J}$, the parameters for $\mathcal{T}$ are derived from tree parameters as:

$$\sigma_{\max}(i,J) = \min(\sigma_{\max}(T) : T \in \mathcal{T})$$
$$\pi_{\min}(i,J) = \max(\pi_{\min}(T) : T \in \mathcal{T})$$
$$\lambda_{\min}(i,j) = \min(\lambda^T_{\min}(i,j) : T \in \mathcal{T})$$
$$\delta_{\max}(i,j) = \max(\lambda^T_{\min}(i,j) + \delta^T_{\max}(i,j) : T \in \mathcal{T}) - \lambda_{\min}(i,j)$$

The tree-finding task is more challenging as we need to ensure vertex-disjointness. The main building block is finding the set of trees for a single request, represented by a FiDe sender process p_i^*, receiving processes $\{p_j^*\}_{j\in J}$, requested packet size $\sigma_{\max}(i,J)$, and desired sending period $\pi_{\min}(i,J)$. The key observation that we make to enable practical algorithm for finding a set of *vertex-disjoint* paths is that datacenter networks have regular structure. Let us denote the network graph as $G = (V,E)$, the distance between $u,v \in V$ as $d(u,v)$, the set of endhosts as $H \subseteq V$, for a FiDe process p_k^* we let h_k denote the endhost where p_k^* is running and let $H_J = \{h_j\}_{j\in J}$. We also define the set of all nodes at distance exactly r from v as $N^r(v) = \{u \in V : d(v,u) = r\}$, and the set of all endhosts h' such that there exists a shortest path from $h \in H$ to h' through v as $N_H(h \rightsquigarrow v) = \{h' \in H : d(h,h') = d(h,v) + d(v,h')\}$. Now, the regularity of the network topology is expressed by the assumption that for any endhost $h \in H$, and any two nodes $v,v' \in G^r(h)$, either $N_H(h \rightsquigarrow v) = N_H(h \rightsquigarrow v')$ or $N_H(h \rightsquigarrow v) \cap N_H(h \rightsquigarrow v') = \emptyset$; moreover for any $h' \in H$, $h' \neq H$ we have $N_H(h \rightsquigarrow h') = \{h'\}$.

The core idea of the algorithm is to produce a set of multicast trees $\mathcal{T}$ bottom-up, each step being identified by $r \in \{0,\ldots,r^*\}$, where r^* is the largest satisfying $N^{r^*}(h_i) \neq \emptyset$. At layer r, we consider $G^r(h_i) = V_1^r \cup \ldots \cup V_{l_r}^r \cup V^r$, each V_k^r representing an equivalence class w.r.t. $N_H(h \rightsquigarrow -)$, so $N_H(h \rightsquigarrow V_k^r)$ is well-defined, and $N_H(h \rightsquigarrow V_k^r) \cap H_J \neq \emptyset$. We call V_k^r *leaf* if $V_k^r = \{h_j\}$ for some $h_j \in H_J$. For each V_k^r, for each $X \subseteq V_k^r$, s.t., $X = V_k^r$ for leaf V_k^r or $|X| = K$ for non-leaf V_k^r the algorithm constructs set $\mathcal{T}_k^r(X)$ of $|X|$ rooted

vertex-disjoint trees, s.t. (1) the set of roots must be exactly X; (2) each tree must have exactly $N_H(h \rightsquigarrow V_k^r) \cap H_J$ as leaves; and (3) all paths from the root must be shortest in G.

The base case corresponds to V_k^r being leaf, where $\mathcal{T}_k^r(X)$ has a single trivial tree. For the inductive case, we want to build $\mathcal{T}_k^r(X)$ for non-leaf V_k^r. To that end, we consider all $V_{k'}^{r+1}$ such that $N_H(h_i \rightsquigarrow V_{k'}^{r+1}) \subseteq N_H(h_i \rightsquigarrow V_k^r)$. Each such $V_{k'}^{r+1}$ would give a separate child to each of the K trees in $\mathcal{T}_k^r(X)$ as due to our assumption of $N_H(h_i \rightsquigarrow V_{k'}^{r+1})$ being disjoint and the constraint on all the paths being the shortest makes the branching necessary at this point. Concretely, to build $\mathcal{T}_k^r(X)$, the algorithm loops over each $V_{k'}^{r+1}$, at each iteration trying all possible $Y \subseteq V_{k'}^{r+1}$ and trying all possible ways of attaching each tree from $\mathcal{T}_{k'}^{r+1}(Y)$ as a child of some $v \in X$, finding one that minimizes the *cost*, to be defined shortly, of the resulting tree. Note, since the overall *cost* of a tree is monotone with respect to each child, the choices for each $\mathcal{T}_{k'}^{r+1}(Y)$ are independent. The overall complexity is $O(|V|^{2K} \cdot K! \cdot d)$, where d is the G's diameter.

The cost of a set of trees is computed as the sum of two components, one measuring the maximum latency of the current set of trees, the other being an upper bound on the maximum latency increase of existing trees. The former is defined simply as $\lambda_{\min}(r,j) + \delta_{\max}(r,j)$, where $j \in J$ and r is the root of the tree, while the latter is defined as a sum of queuing delays $\sigma_{\max}(i,J)/\beta(u,v)$ introduced by the new tree at each of the links for each of the other trees using the same link, where $\beta(u,v)$ is the bandwidth of the (u,v) link.

C Critical Compound Failure Probability Estimation

This section details the procedure used to estimate the probability of a *critical compound failure*, i.e., a combination of network failures not tolerated by FiDe. This type of failure is outside the scope of FiDe's model, hence monitoring processes can possibly misinterpret it as a remote process failure (false suspicion). Recall that the two-fold redundancy and the tree recovery mechanism are robust towards a single network failure, since processes can rely on the other (healthy) tree, and also tolerates multiple failures in the redundant trees given they allow FiDe TE to recover. We will shortly see how the recovery time is in fact not influential for critical compound failures. Moreover, we assume that FiDe terminates gracefully, when one of the two trees is still alive and there are no more healthy links available for tree recovery.

In what follows, we consider all events with respect to a given operational *year*. We define two auxiliary events:

- f^l: link l fails at least once;
- $f^{l,l'}$: links l and l', $l \neq l'$, fail at least once within the tree recovery time.

Now we can define a critical compound failure event:

$f_{\text{CCF}}^{\mathcal{T}}$: at least two distinct network failures occur within the tree recovery time interval affecting all multicast trees in multicast tree set $\mathcal{T}$ (in FiDe, we have $|\mathcal{T}| = 2$).

Formally, for redundancy level two $f_{\text{CCF}}^{\{T_1,T_2\}} = \{\exists\ l_1 \in T_1, l_2 \in T_2 : f^{l_1,l_2}\}$.

We use switch and link failure statistics from a seminal large scale analysis of datacenter network failures from Gill et al. [22]. Note that, while FiDe's failure model defines a network failure as a more generic packet drop, we restrict the scope to switch and link failures only because packet drops caused by congestion are prevented by FiDe's TE-driven periodic multicast protocol. We report probability estimates relevant to our topology from [22] below (marked with $'$).

- If l is a "trunk" link, then $p(f^l) \leq p'_{tr} = 0.054$ (probability of a "trunk" link failure) [22, Fig. 5].
- If l is a "core" link, then $p(f^l) \leq p'_{cr} = 0.095$ (probability of a "core" link failure) [22, Fig. 5].
- If $p(\exists\, l' \neq l : f^{l,l'} | f^l) \leq p'_g = 0.41$ (probability of a given failure affecting a group of links) [22, Fig. 10].

Gill et al. [22] report that (a) switch failures do not show correlation and that (b) link failures are either "simultaneous" (without giving an exact quantification) or the time between them is too large to indicate a meaningful correlation. Finding (a) suggests that FiDe is very likely to always tolerate switch failures since the trees in $\mathcal{T}$ are disjoint on inner nodes. For finding (b), consider that FiDe adopts a conservative tree recovery time of several seconds (our measurements indicate a recovery time ms-range similar to several studies [25,40,77]), which, for the purposes of this evaluation, we can confidently assume to be much smaller than the "large grouping thresholds" evaluated in the empirical study. We therefore consider only simultaneous link failures as potential causes of a critical compound failure and do not delve into a detailed quantification of the tree recovery time.

We compute an upper bound on $p(f_{\text{CCF}}^{\mathcal{T}})$ for multi-rooted fat trees of different *height*, which we define as half the diameter of a fat tree. Fig. 8 provides a visual example of a multi-rooted fat tree, showing how deployment of FiDe processes can create "sub-trees" withing a broader topology. The formulas below are used to derive the upper bound on the probability of critical compound failure.

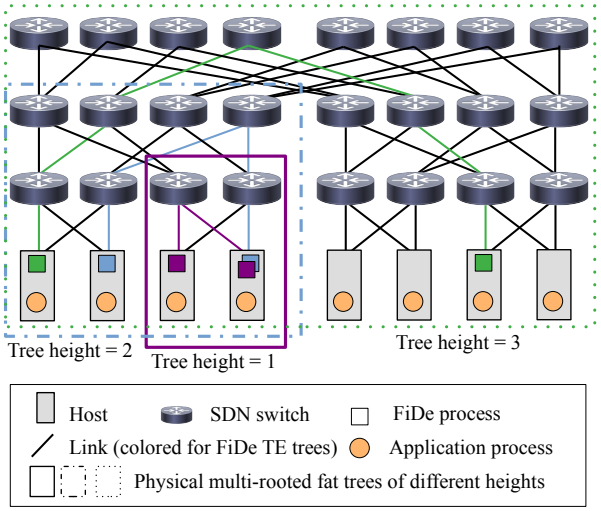


Figure 8: Widely-adopted multi-rooted fat tree topology with three different FiDe deployments (color-coded) showcasing different tree heights.

$$
\begin{aligned}
p(f_{\mathrm{CCF}}^{\{T_1,T_2\}}) &\leq \sum_{l_1 \in T_1} p(\exists\, l_2 \in T_2 : f^{l_1,l_2}) && (1)\\
&= \sum_{l_1 \in T_1} p(\exists\, l_2 \in T_2 : f^{l_1,l_2} | f^{l_1}) \cdot p(f^{l_1}) && (2)\\
&\leq \sum_{l_1 \in T_1} p(\exists\, l' \neq l_1 : f^{l_1,l'} | f^{l_1}) \cdot p(f^{l_1}) && (3)\\
&\leq p'_g \cdot (p'_{tr} \cdot n_{tr}(T_1) + p'_{cr} \cdot n_{cr}(T_1)) && (4)\\
&= p'_{\mathrm{CCF}}(\{T_1,T_2\})
\end{aligned}
$$

where $n_{tr}(T)$ and $n_{cr}(T)$ denote the number of "trunk" and "core" links, respectively. In (1) we use a union bound over links in T_1, in (2) we use the fact that $f^{l_1,l'} \subseteq f^{l}$ and we can use the definition of conditional probability, in (3) we use the fact that any link T_2 contains only a subset of links different from l_1, and, finally, in (4) we apply the bounds.

For a practical example, consider the topology with height $H = 1$ from Fig. 8, which has only two trunk links in each tree, i.e. $n_{tr}(T_1) = 2$ and $n_{cr}(T_1) = 0$. We obtain the upper bound on the failure probability $p'_{\mathrm{CCF}}(\{T_1,T_2\}) = p'_g \cdot p'_{tr} \cdot 2 \approx 0.044$. In general, for a set of multicast trees $\mathcal{T}$ connecting n FiDe processes and having height $H \in \{1,2,3\}$ we have $p'_{\mathrm{CCF}}(\mathcal{T}) = p'_g \cdot n \cdot (\max(H-2,0) \cdot p'_{cr} + \max(H-1,0) \cdot p'_{tr})$. Concretely, if $n = 2$, then for $H = 2$ we get $p'_{\mathrm{CCF}}(\mathcal{T}) \approx 0.089$ and for $H = 3$ we get $p'_{\mathrm{CCF}}(\mathcal{T}) \approx 0.17$.

Critical compound failure probability in perspective. The expected frequency of a critical compound failure ($freq_{\mathrm{CCF}}$) based in the smallest (ideal) topology is 1 every *22.73 years*, hence extremely likely to outlive any realistic service uptime. To put this number in perspective, we compare it against the probability of packet corruption in TCP, widely regarded and used as reliable communication layer by coordination services such as Zookeeper [29]. A popular study [69] reports that "the Ethernet CRC + TCP checksum will fail to detect errors for roughly 1 in 16 million to 10 billion packets", with recent real-world cases being reported [72, 73]. If we assume equivalent TCP traffic as FiDe's throughput when set with the lowest possible sending period of $T = 20\mu s$ (500Mbps) used in our evaluation, the reported error rate would translate to one packet corruption every 320 seconds to 2.3 days, making it several orders of magnitude more reliable. Considering that error reports [22, 69] are from decades ago, we expect both probabilities to be significantly smaller in modern networks, challenging the common misconception that any network is unreliable.

LEOCraft: Towards Designing Performant LEO Networks

Suvam Basak*, Amitangshu Pal*, Debopam Bhattacherjee†
*Indian Institute of Technology Kanpur, †Microsoft Research India

Abstract

Low Earth Orbit (LEO) satellite constellations have revolutionized Internet access for millions of users. OneWeb and SpaceX are already operating constellations of hundreds and thousands of satellites, offering Internet service directly from space across 100+ countries. These exceptionally large networks come at a cost – thousands of routers (satellites) need to fly at ∼22× the speed of sound, thus making network design a non-trivial challenge. While the systems research community with decades of deep networking expertise has a relatively short window to influence the design of these networks, there is a serious lack of the right tools to enable such efforts. To address this, we introduce LEOCraft – an LEO network design framework to help the community visualize and evaluate the performance of different choices. LEOCraft offers integrated optimization techniques tuned upon the domain knowledge acquired from thousands of LEO constellation design's performance evaluations to optimize a new constellation design ∼5× faster than other off-the-shelf black-box optimization techniques. LEOCraft scales up seamlessly, tested for 83K satellites across multiple shells (more than 2× SpaceX's long-term proposal) with 1K ground stations, thus making it feasible for the community to explore LEO trajectory and topology design for even the largest of *mega*-constellations.

1 Introduction

In the past few years, thousands of satellites have been deployed in Low Earth Orbit (LEO) to serve worldwide high-speed Internet broadband service directly from space [44]. The early movers of this new networking landscape, SpaceX's Starlink and OneWeb, initially relied upon bent-pipe connectivity in the first generation of constellations [58]. Further technological advancements have introduced robust laser Inter-Satellite Links (ISLs), enabling space-borne mesh networks with high bandwidth and speed-of-light grade latency [55]. Such a LEO satellite network can carry long-haul Internet traffic over the satellite network, thus reducing the requirement for denser Ground Station (GS) deployment across the globe [2]. This development enabled Starlink and Amazon's Project Kuiper to propose a 2nd generation mega-constellation of 30,000 [3] and 3,236 satellites [1] respectively, to wrap the Earth with a dense mesh of LEO satellites. However, the FCC approved only 7,500 satellites of Starlink across three different altitudes [3]. SpaceX started launching these new generation satellites in December 2022 [61]. In contrast, Kuiper was recently in the testing phase of the optical mesh network in space [5]. Therefore, none of the operators has achieved the true scale of ISL-enabled mega-constellations yet, keeping the *brief window* open for the networking community to influence such design.

The secret of constellation design: The success of Starlink's early launches sparked researcher's interest in exploring various aspects of LEO networks. Such recent works [44–46, 50, 54–56, 58, 64, 67, 68, 76] have used Starlink, OneWeb, and a few other constellation designs in their experiments since details are publicly available in the FCC filings [1, 3, 8, 12, 21, 31]. However, none of these existing works include the complete set of design parameters determining the satellite's trajectory as well as topology [45, 72, 92, 93]. Most of these efforts are restricted to a single shell, considering the inter-shell connectivity and aggregated performance measurement of multiple independent shells as *"an open problem"* [93].

Furthermore, Starlink and OneWeb, the biggest constellation operators, rely on proprietary systems, making their LEO networks function like black boxes. Why they chose a particular design, and what traffic matrices it targets, is unknown to the community. Therefore, in this work, we aim to unveil the subtle crux of designing an optimized LEO network from scratch. Our effort will help networking researchers understand these 'new space' networks better.

Why constellation design is non-trivial?: Designing a satellite network is not a new problem. In the 90s, serious efforts [37, 41, 52, 65, 66, 89, 90] have been devoted in studying constellation design. However, the current scale and design objectives have significantly transformed over time, imposing new challenges. Optimizing a satellite constellation in the 90s emphasized coverage with a limited budget of a few 10s of

satellites, ensuring availability worldwide [42]. In contrast, now, with the scale of a few thousand or more satellites, network performance has become comparable to the terrestrial networks [53]. Therefore, optimizing a constellation today is no longer a geometrical optimization of overlapping satellite coverage [51], but rather, an optimization of the satellite's orbital trajectories and their ISL topology for a given Internet traffic demand matrix. Therefore, given the number of satellites, the design problem is to decide upon several parameters describing ISL topology and each satellite's orbital trajectory at $\sim$22$\times$ the speed of sound, such that network performance is maximized. A network design is already a hard problem [62]; further joint optimization of orbital parameters with added dynamics of satellite motion makes it a non-trivial high-dimensional optimization problem. A naive brute-force grid search approach looking for maximum network performance measurement could take years due to the curse of dimensionality. A few previous efforts explored the aspect of improving constellation performance with better ISL topology [45, 72]; however, the approach [45] suffers from scalability issues hence could not be extended to the direction towards superior constellation design, i.e., optimization of satellite's trajectory at scale to enhance the network performance.

Missing the right tools: An in-depth study of trajectory and ISL topology optimization of the LEO network requires the execution of tens of thousands of simulations while varying one or more orbital/design parameters to measure how these choices impact the network performance. Therefore, an LEO network evaluation framework must offer the flexibility to (i) rapidly implement different ideas (topology, ISL connectivity, routing strategy, etc.), (ii) plug and play with various optimization strategies, (iii) evaluate the effectiveness of optimization strategies and further tuning these implementations. Existing platforms, i.e., *Hypatia*'s [64] packet-level simulator, or *StarryNet*'s [67] *Docker* container-based emulator, suffer from fundamental scalability issues [63]. Hence does not go well for large-scale (tens of thousands of satellites) trajectory simulation. Another LEO network emulation platform *xeoverse* [63] is built upon a lightweight process-based network emulator platform *Mininet* [20] and claims to outperform *Hypatia* and *StarryNet* by large margins; however, their source code is not public for community use. Therefore, the community lacks the right tools to approach large-scale trajectory and topology optimization problems.

Our contributions: In this paper, we address these gaps with `LEOCraft` – an LEO satellite constellation design exploration, optimization, and visualization framework. `LEOCraft` relies upon process-based parallelism [14], thus bypassing the Global Interpreter Lock, a major computational bottleneck [17] in the existing platforms [64, 67]. It is designed to be a modular framework that allows rapid implementation of new design ideas with minimal code change (typically 100 lines of code), which helps to run new experiments quickly. It operates at the flow level, computes network measurements from the intuitive mathematical model of the LEO network instead of resource-intensive packet-level simulations. For instance, the performance evaluation of Starlink's Gen1 three shells of 3,888 satellites (S-1 to S-3 in Table 1) in `LEOCraft` takes $\sim$2.5 minutes on a standard Desktop PC. Whereas with *Hypatia*, it takes hours for only RTT measurement of Starlink's single shell of 1,584 satellites. We also tested `LEOCraft` on a hypothetical mega-constellation of 83+ satellites across multiple shells (as shown in Table 1). Thus, `LEOCraft` enables rapid evaluation of design choices at a large scale. The visualization module also generates interactive views of the constellation, showcasing topology evolution and end-to-end route changes to augment the intuition of LEO dynamics.

We use `LEOCraft` to rigorously study the constellation design for given traffic demand matrices across a set of GSes and illustrate how the choices of design parameters impact the performance of these LEO networks. Since Internet traffic demand is skewed towards specific regions of the Earth with higher populations, the performance measurements of constellation designs exhibit common trends irrespective of the satellite budget. We use this domain knowledge and predictable characteristics of network performance measurements to prune the search space drastically. This reduces the running time by $\sim$5$\times$ as compared to the naive off-the-shelf black-box optimization techniques. We also demonstrate how inter-shell ISL can improve the throughput of a constellation and highlight some shortcomings of such ISL usage. We made `LEOCraft` publicly available [35] for community use.

Paper outline: The remaining paper is organized as follows – §2-3 summarize our overall background and problem overview. Then §4-6 discusses the development, optimization, and system design strategies used in `LEOCraft` framework. §7-8 present the experimental evaluation. §9 summarizes the related works. The paper is concluded in §10.

2 Background

Building an LEO constellation is a relatively new topic, so this section aids the reader with the relevant background.

2.1 LEO constellation design parameters

A satellite's trajectory around the Earth can be described by six parameters given by Kepler's law of interplanetary motion [84] using Two Line Element set (TLEs) [59], which is a standard format of defining the position, velocity, and trajectory of any Earth-orbiting object at a specific time (epoch). Since establishing an ISL takes a few 10s of seconds [45], satellites in a constellation follow a similar trajectory to sustain the ISL connectivity for a longer period of time. `LEOCraft` automatically generates the TLEs for each satellite for the given six parameters defined in the following:

Altitude (h): The height of the LEO satellites from the sea level typically falls between 500 to 2,000 km. These satellites need to orbit the Earth at a fixed velocity to maintain an orbit

Table 1: Constellation parameters of well-know space players [1, 3, 8, 12, 21, 31]. In this table *S-x* denotes *x*-th shell.

Starlink Gen1					
s	*o*	*n*	*i*	*e*	*h*
S-1	72	22	53°	25°	550 km
S-2	72	22	53.2°	25°	540 km
S-3	36	20	70°	25°	570 km
S-4	6	58	97.6°	25°	560 km
S-5	4	43	97.6°	25°	560 km
Telesat					
s	*o*	*n*	*i*	*e*	*h*
S-1	27	13	98.98°	10°	1,015 km
S-2	40	33	50.88°	10°	1,325 km

Starlink Gen2					
s	*o*	*n*	*i*	*e*	*h*
S-1	48	110	53°	25°	340 km
S-2	48	110	46°	25°	345 km
S-3	48	110	38°	25°	350 km
S-4	30	120	96.9°	25°	360 km
S-5	28	120	53°	25°	525 km
S-6	23	20	43°	25°	530 km
S-7	18	18	115.7°	25°	614 km
S-8	12	12	148°	25°	604 km
S-9	28	120	33°	25°	535 km

Kuiper					
s	*o*	*n*	*i*	*e*	*h*
S-1	34	34	51.9°	35°	630 km
S-2	36	36	42°	35°	610 km
S-3	28	28	33°	35°	590 km
OneWeb					
s	*o*	*n*	*i*	*e*	*h*
S-1	36	49	87.9°	5°	1,200 km
S-2	32	720	40°	5°	1,200 km
S-3	32	720	55°	5°	1,200 km

at a particular height; hence, the orbital period of a satellite is also fixed for a particular height. For instance, satellites at 500 km of altitude need to orbit the Earth at $\approx$ 27,000 kmph; therefore, the orbital period would be $\approx$ 90 minutes. Increasing the altitude of the satellites will decrease the orbital periods, as these two factors are inversely proportional.

Angle of inclination (i): The angle between the orbital plane and the Earth's Equatorial plane (when satellites are moving towards the north). Hence, a satellite with inclination 90° orbits the Earth over the north pole. Meanwhile, a lower inclination tilts the trajectory; hence, the satellite orbits through the lower latitudes.

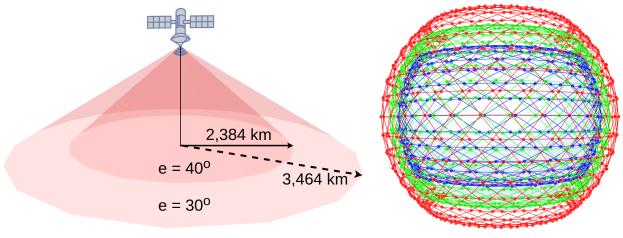
Number of satellites per orbits (n): Each orbit in a LEO satellite constellation contains a fixed n number of satellites placed equidistantly across the orbit. All the satellites of fixed altitude orbit the Earth at the same velocity; hence, the in-between distance of $360°/n$ is maintained consistently.

Number of orbits (o): Similarly, the constellation would have o orbits, each with n satellites, uniformly spaced along the Equatorial line to ensure even coverage of Earth's surface.

Phase offset (p): Determines the relative positioning offset between satellites of adjacent orbits. In our implementation, p varies from 0 to 0.5. A phase offset of 0.5 denotes that the n-th satellite of $(o+1)$-th orbit is positioned at the middle of n and $(n+1)$-th satellites of o-th orbit.

Angle of elevation (e): The satellite's coverage area on Earth's surface depends on the minimum elevation angle e as shown in Fig. 1(a). This parameter depends upon the capability of space- and ground-based communication hardware. LEO satellites use software-defined phased array antennas to create spot beams on Gateway Ground Stations and User Terminals [1, 3, 8]. Hence, the onboard software can regulate the coverage area with the elevation angle. Table 1 summarizes the constellation parameters of major space players.

Furthermore, the LEO satellite mega constellation design comprising a few thousand or more satellites goes one step ahead, where the satellites are divided into groups that may have different designs; the satellites within a group follow similar trajectories that help sustain the ISLs for longer periods, as shown in Fig. 1(b) using Kuiper's proposed design. In LEO constellation terminology, these groups are known as *shells*. We assume that the satellites within a shell establish



(a) (b)

Fig. 1: (a) Low elevation e covers larger areas on the Earth's surface. (b) Kuiper design of three shells at altitudes of 630, 610, and 590 km with inclination 51.9°, 42°, and 33°, respectively.

four ISLs, two in the same orbit with leading and trailing satellites, while the other two with satellites of two adjacent orbits. Hence, each shell creates a layer of + shaped mesh network, defined as +Grid topology [45]. The `LEOCraft` takes six parameters to describe the design of each shell and automatically generates the TLEs for all the satellites for these shells, as illustrated in Fig. 1(b).

2.2 LEO network modelling and assumptions

The proprietary nature of the LEO constellation prohibits public disclosure of key details, such as satellite connectivity policies, user terminal associations, and bandwidth distribution mechanisms, etc. Thus, we base our assumptions on standards widely adopted in prior literature.

Network model: In `LEOCraft`, given a set of design parameters, we model the LEO satellite constellation as a big network graph, where nodes represent the satellites and GSes, and edges represent Ground to Satellite Links (GSLs) and ISLs. Unavailability of precise geospatial information prevents the modelling of individual user terminals [88]. Therefore, we assume the GS located in a city serves as a proxy for the demands of all the users in that city. Consequently, we allow a GS to establish GSLs with all the satellites available above the minimum angle of elevation e (from the horizon) at that particular time.

GSL budget model: In the above network, capacities of GSLs C are calculated from Shannon's channel capacity theorem [82]. The atmospheric path loss is estimated using the Free-Space Path Loss (FSPL) model [60]. With these, the channel capacity is expressed as $C = B\log_2\left(1+\frac{P_t G_t \lambda^2}{(2\pi d)^2 k_B B}\times\frac{G_r}{T}\right)$, where, P_t is the Tx power

(EIRP), G_t and G_r are the Tx and Rx antenna gains respectively, λ is the wavelength, and antenna gain-to-noise-temperature ratio G_r/T are taken from Ka-band specification available in FCC filling [50]. d denotes the distance between the Tx and Rx antenna, i.e., the distance between the satellites and GSes, k_B is Boltzmann's constant, and B is the transmission bandwidth. Note that, in our implementation, B is divided equally among the GSes under the coverage of each satellite.

ISL budget model: The laser ISLs are expected to have extremely low beam divergence properties and narrow beam widths, minimizing interference issues [4]. Currently, ISLs are in the early phase of deployment [30] or the testing phase of different operators [5, 11]. Since the largest LEO network operator Starlink's satellites use 4 ISLs [43] and subsequently all the prior work [45, 64, 67, 92, 93] made this a de facto standard, this work assumes and extends the same with capacity of each ISL conservatively set to 50 Gbps [6, 7, 9]. Although it is a configurable parameter in `LEOCraft`.

2.3 Internet traffic demand matrices

Since a geospatial dataset of individual Starlink users is not public [88], we use the following intuitive traffic demand matrices (TM) that follow the gravity model [79].

High population TM: We use World Cities Database [34] for the population dataset, where GSes are located in 100 most populous cities around the globe. We assume that 10% of a city's population is the targeted customer, with average data usage per head of 300 Kbps [76]. The traffic demand across the GSes is inversely proportional to their geodesic distances.

High GDP population TM: Apart from the city population, typically, the Internet traffic is also proportional to the city's GDP [38]. So, in this demand matrix, we assume that the targeted market is 10% of the total population of the 100 most populous cities. However, the data usage is weighted based on the city's GDP. We keep the mean data usage to be 300 Kbps. Similar to the previous matrix, the demand across the GSes is inversely proportional to their geodesic distances.

Country capital TM: In this matrix, the GSes are located in the capital of 233 countries. The demands across the GSes are weighted proportionately to the country's population and in inverse proportion to the geodesic distances between GSes.

Global flight TM: In-flight Wi-Fi market is already a billion-dollar industry, forecasting an annual growth rate of 11.8% [18]. Therefore, domestic and international airlines are also potential target markets for the in-flight high-speed Wi-Fi service [80]. For that, we use *FlightRadarAPI* [33] to collect details of the in-flight aircraft worldwide. Then we assume 50% of passengers of each flight above 10,000 feet of altitude are communicating with GSes located at the 100 most populous cities via satellite network. We weighted the demand matrix between the flights and the GSes using the gravity model discussed above, whereas the weights are directly proportional to populations/passengers and inversely proportional to the distance between the flights and the GSes.

Moreover, notice that the TMs are subject to the target market. A satellite constellation designed to serve the shipping industry might weight the demand matrices analogous to the busiest trade routes in the ocean. Hence, many such intuitive TMs can be formulated. However, our study is focused only on the TMs discussed above.

2.4 Network performance metrics

We evaluate the constellations on the following three performance metrics.

Throughput: We measure the throughput of the LEO satellite constellation as a multi-commodity flow across the GSes. We compute n shortest routes between the GSes at any given time using Yen's algorithm [91]. Then, we use the following linear program to maximize the flow between all pairs of GSes.

$$\text{Maximize: } \sum_{f \in \mathbf{F}} D_f \times \sum_{r}^{n} R_r^f \tag{1}$$

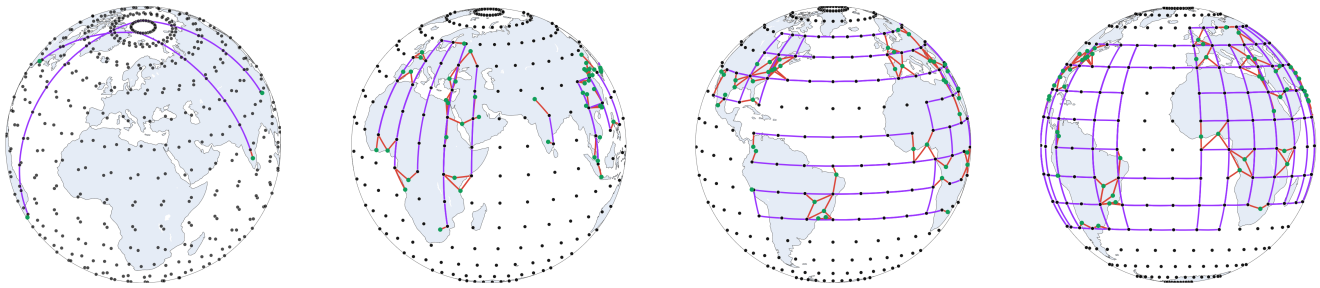
$$\text{Subject to: } \quad 0 \leq \sum_{r}^{n} R_r^f \leq 1 \quad | \, \forall f \in \mathbf{F} \tag{2}$$

$$\sum_{(f,r) \in \mathbf{S}_l} D_f \times R_r^f \leq C_l \quad | \, \forall l \in \mathbf{L} \tag{3}$$

Here, $\mathbf{F}$ is the set of flows, where a flow is defined as an ordered pair of GSes denoting traffic flow between one GS pair. D_f is the traffic demand for the flow f (demand between two GS). Since we have n shortest routes between each pair of GSes, R_r^f denotes the fraction of demand D_f flowing through the r-th shortest route. Therefore, $\sum_r^n R_r^f$ the sum of n continuous variable bounded within zero and one in constraint (2), which ensures n shortest routes together can accommodate at most 100% of the demand D_f of any flow $f \in \mathbf{F}$. Finally, the links could be shared by more than one flow in the network. Therefore, aggregated traffic flows sharing a particular link are constrained by the capacity of that link. Hence, in constraint (3), $\mathbf{L}$ is the set of all the links in the network, and $\mathbf{S}_l$ denotes a set of flows f and their r-th shortest route that shares link l. Therefore, $\sum_{(f,r) \in \mathbf{S}_l} D_f \times R_r^f$ denotes total traffic flow via link l, which is constrained by the C_l, the capacity of link l. For all our simulations, we use $n = 20$.

Stretch (latency): We measure *stretch* between source and destination GS pair as the ratio of the distance between two locations over the satellite network through ISLs and their geodesic distance. This metric demonstrates the inflation in the length of the end-to-end routes. Since the scope of this work is a comparative performance study of constellation design, lower stretch and lower hop counts for end-to-end routes are expected to give better network latency.

To understand the generalized nature of latency worldwide across diverse routes, we classify all the routes into five categories based on their nature. The first two categories are based on the distance between the source and destination



(a) HG routes (b) NS routes (c) EW routes (d) NESW routes

Fig. 2: Illustration of the route categories based on the path orientation and distance, where red lines are *GSLs*, violet lines are *ISLs*, black dots are the *satellites*, and green dots are the *GSes*.

GSes. Therefore, (a) any pair of GSes located within a distance of 2,000 km falls in the *Low Geodesic (LG) distance* class. Such routes will always come with higher stretch because of the route overhead of going up from the source GS to a satellite and coming down from a satellite to the destination GS, only to use one satellite hop (most of the time). (b) Any source and destination pair of GSes located far apart in different hemispheres ($\geq$ 8,000 km) falls in the class of *High Geodesic (HG) distance* routes as illustrated in Fig. 2(a). All other routes that do not fall into the above two classes are classified based on the orientation of the path. Therefore, the routes between the GSes pairs oriented with (c) angle above 75° shown in Fig. 2(b) are classified as *North-South (NS)* routes, (d) angle below 15° shown in Fig. 2(c) are classified as *East-West (EW)* routes, and (e) angle within 15° and 75° shown in Fig. 2(d) are classified as *NorthEast-SouthWest (NESW)* routes. For overall stretch tendency, we report the median stretch and hop counts of each category.

Coverage: One of the unique potentials of the LEO satellite constellation is to offer genuinely global high-speed Internet service irrespective of geographical challenges. However, to sustain the business, the obvious choice of a constellation operator would be to optimize the satellite trajectories to provide higher availability where the volume of customers is high. Therefore, we measure the coverage of a constellation as $\sum_{t=0}^{T}\sum_{g=0}^{G}\log_{10}(n_{t,g})/T$, where G is the number of GSes, T is the number of epochs, and $n_{t,g}$ denotes the number of satellites visible from the g-th GS at t-th epoch. The $\log(.)$ function models the diminishing returns as the number of visible satellites per GSes increases.

3 LEO network design from scratch

With the preceding background, we now delve into the LEO trajectory design problem, along with the research challenge of high dimensionality.

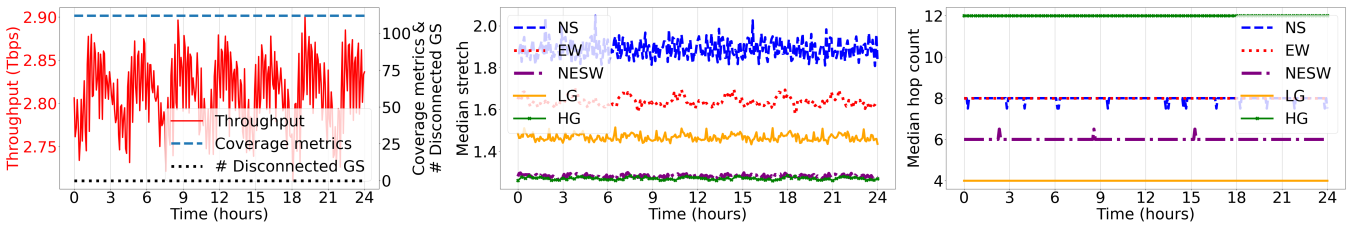
Problem statement: We define our problem statement as follows – given the number of satellites (which is our network budget), location of GSes, and traffic demand matrix across these GSes, we need to find the constellation design parameters of one or more shells (s) and their operational altitude (h), inclination (i), minimum elevation angle (e), number of orbits (o), satellites per orbit (n), and the phase offset (p) between satellites in adjacent orbits that maximize some objective function such as throughput, coverage, latency, or a combination of these. However, in this work, we focus on a performant constellation that maximizes the throughput. Higher throughput produces more salable bandwidth, which might fetch higher revenue for the constellation operator.

The curse of dimensionality: Therefore, in a mega-constellation of multiple shells, a brute-force grid search approach to find near-optimal design parameters will require the execution of $|s| \times |h| \times |i| \times |p| \times |o| \times |n| \times |e| \times |t|$ simulations, where $|x|$ means number of distinct choices of parameter x, and $|t|$ denotes the number of epochs over 24 hours (one complete rotation of Earth) to estimate the performance fluctuation for LEO dynamics. Each simulation involves generating a network graph, calculating link budgets, computing routes globally, and evaluating network performance. That could take from a few minutes to hours or even days, depending on the number of satellites and GSes in the constellation. Therefore, executing these many simulations with many computational cores could take years.

Can we prune the search space? Given the large search space, meta-heuristic methods like *Simulated Annealing* [85], *Differential Evolution* [83], and *Adaptive Particle Swarm Optimization* [81] can offer near-optimal solutions. In addition, we leverage domain knowledge and heuristics related to the design parameter's impact on network performance to reduce the search space significantly. Our comprehensive analysis of design parameters highlights their performance characteristics and inter-dependencies. This insight refines meta-heuristic approaches. Further enables tuning of local search strategies to outperform out-of-the-box meta-heuristics.

4 Exploring the search space

We first explore the search space by conducting a single-dimensional analysis and extend the understanding to higher dimensions. Therefore, first, we focus on the single shell analysis; to characterize the effect of different design parameters, we use Starlink's first shell, which consists of 1,584 LEO satellites operating at 550 km of altitude. The satellites are



(a) Throughput and coverage (b) Stretch (c) End-to-end hop counts

Fig. 3: The propagation of satellites over time leads to fluctuation in (a) throughput (left Y-axis), (b) stretch, and (c) hop counts within certain bounds. However, (a) coverage is consistent (right Y-axis).

distributed among 72 orbits inclined at 53°, each with 22 satellites. The elevation angle of Starlink's first shell is chosen as 25°. Unless otherwise mentioned, we use a phase offset of 0.5 in our simulations. All these parameters and upper Ka-band specifications are taken from the FCC filing [10, 50]. Performance characteristics of other proposed constellations are discussed in §A.1. We use the *High population TM* for this section; however, in §A.2, it shows that the findings are generalizable to other traffic metrics as well.

Performance impact of LEO dynamics: We begin by measuring network performance fluctuations due to LEO dynamics. We simulate Starlink's first shell for 24 hours with 5 minutes of granularity. The network performance metrics from the simulation are shown in Fig. 3. Notice that in Fig. 3(a), the constellation gives consistent coverage, and none of the GSes experience an outage within 24 hours. However, throughput shows a periodic pattern of fluctuation of $\sim$ 150 Gbps, which is up to 7% of the maximum recorded throughput. Similar behavior is also noticeable for other performance metrics, i.e., stretch fluctuates within a small margin, and hop counts are mostly stable, sometimes changing by at most two hops, as shown in Fig. 3(b)-(c), respectively. It is essential to notice that due to the Earth's rotation, one orbit will eventually take the position of the next. Also, within each orbit, one satellite will take the position of the next satellite after a specific time. Hence, the fluctuation rate of performance metrics will be within a relatively small margin.

Key takeaway(1):- The network performance remains stable, fluctuating within 6.5% due to LEO dynamics.

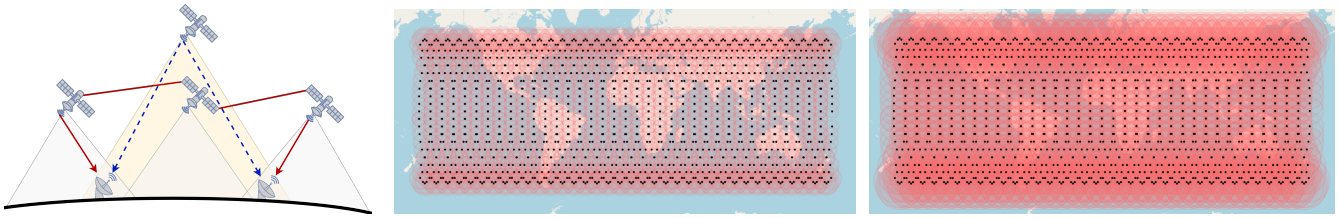
Performance impact of altitude (h): Operating a satellite at a higher altitude with the same elevation angle e will increase the coverage area on the Earth's surface as illustrated in Fig. 4(a). Therefore, operating a shell at higher altitudes of 600 km gives much denser coverage than operating the same at an altitude of 300 km, as shown in Fig. 4(b)-(c), respectively. We simulate Starlink's first shell by gradually increasing the altitude h of the shell from 300 to 2,000 km, while all the other parameters are kept constant. The results of the simulation are shown in Fig. 5. At higher altitudes, the availability of satellites per GS increases, which is visible in the coverage metrics of the constellation in Fig. 5(a). This higher availability of satellites per GS gives higher end-to-end path diversity, leading to a throughput hike up to a certain point. However, beyond a certain point, increasing h results in longer GSLs with higher atmospheric loss. Also, at higher altitudes, many GSes connect to the satellites, which results in low bandwidth for each GSL. As a result, throughput eventually starts falling.

Next, we discuss the effect of h on network latency. Notice that the shells at lower altitudes limit satellite availability per GSes. For that, the shortest path has to take a few more hops to reach the satellites above the destination GSes, as illustrated in Fig. 4(a). This behavior is also visible in our simulation results in Fig. 5(b)-(c). Apart from the LG routes, when the altitude of the shell is increased beyond 300 km, the stretch and hop count decrease drastically. The stretch starts inflating beyond 700 km because of longer GSLs. For LG routes, source-destination pairs are close to each other; a higher h results in longer GSLs, leading to increased stretch.

Key takeaway(2):- A fixed number of satellites at lower altitudes could lead to coverage gaps and higher latency, while higher altitudes reduce aggregated throughput. Slight altitude changes have minimal impact.

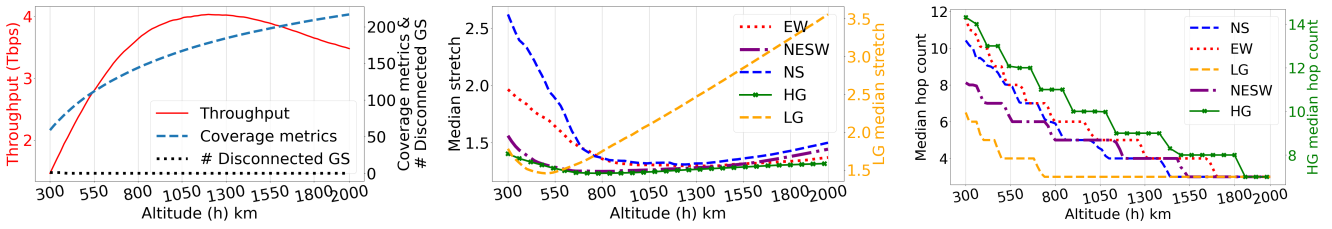
Performance impact of inclination angle (i): Similarly, we simulate Starlink's first shell with the angle of inclination i ranging from 5° to 175°. The results are shown in Fig. 6. From Fig. 6, we can observe that the network performance corresponding to inclination 5° to 90° is completely symmetric to the performance from 90° to 175°. This is because when a shell is inclined at 45°, the satellites that move towards the north at the angle of 45° in one hemisphere are the same satellites that move towards the south at an angle of 135° in the opposite hemisphere. Thus, a shell inclined above 90° is symmetric to an inclination below 90°. Therefore, we focus on the first half of the performance metrics (5° to 90°) to understand the inclination's impact on the network performance.

In Fig. 6(a), we can observe that the throughput and coverage peak arise around 40° of inclination. This is because, if we look at the 100 GSes locations at most populated cities in Fig. 7(a), then we can observe that 40° of inclination restricts the satellite trajectories exactly above these 100 GSes, as shown in Fig. 7(b). However, moving towards the polar inclination (towards 90°) reduces the coverage since satellites spend a certain amount of time in the polar regions, where not a single GS is located in our setup, as illustrated



(a) (b) At $h = 300$ km (c) At $h = 600$ km

Fig. 4: (a) Higher altitude increases coverage areas, resulting in route changes with fewer hops. (b) and (c) illustrate coverage footprint expansion of Starlink's first shell when the altitude is increased.



(a) Throughput and coverage (b) Stretch (c) End-to-end hop counts

Fig. 5: (a) Increasing altitude h improves the coverage (right Y-axis) and throughput up to a certain point (left Y-axis). (b) Very high/low h negatively impacts stretch (left Y-axis), LG routes are most affected (right Y-axis) (c) high h reduces hop counts for all routes (HG routes in right Y-axis).

in Fig. 7(c). Thus, it negatively impacts the throughput. In contrast, a too-low inclination squeezes all the satellites close to the equatorial region. Therefore, most GSes at relatively higher latitudes go out of coverage, as illustrated in Fig. 7(d), and as a result, the throughput falls, whereas the number of disconnected GSes rises in Fig. 6(a).

Fig. 6(b)-(c) shows the performance of stretch and hop counts. Here, polar orbits give better results only for the NS routes because constellations with lower inclinations can not provide any straight path over ISLs in the north-south orientation. This is illustrated in Fig. 8. From Fig. 8, we can observe that the shortest path from St. Petersburg to Johannesburg is stretched with a lower inclination. On the other hand, lower inclination provides better stretch and hop count for EW and NESW routes, as observed for the routes between Dar es Salaam to Kabul and Khartoum to Hong Kong, respectively.

Key takeaway(3):- The orbit's inclination should align with GSes placement and route orientation to enhance coverage and throughput and minimize latency.

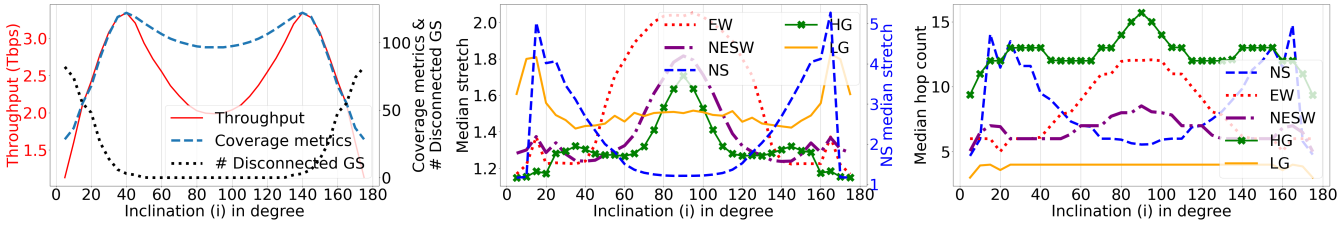
Performance impact of elevation angle (e)**:** As illustrated in Fig. 1(a), a reduction in e increases the coverage area of the satellites on the Earth's surface. That is clearly visible in Fig. 9(a), where e is varied from 5° to 50° for Starlink's first shell. As higher coverage results in more available satellites per GS, the end-to-end path diversity and throughput increase. However, after 13° of elevation, even though coverage increases, throughput starts falling drastically. This is because such a small value of e ($e < 10°$) results in GSLs over the Earth's horizon and therefore experiences enormous atmospheric path loss. This is illustrated in Fig. 10, where we can observe that at 5° elevation, the routes between two continents go through a single satellite hop, as compared to a few 10s of hops for 30° and 50° elevation. Thus, a lower e provides excellent hop counts and stretch results for all categories of routes, as shown in Fig. 9(b)-(c).

However, line-of-sight GSLs with a very low e may not be feasible without obstruction in urban areas (unless antennas are placed at higher elevations). However, a low value of e may be useful for providing connectivity over a long distance in less obstructed areas, such as oceans or polar regions. For instance, OneWeb FCC reports [8] mention that OneWeb satellite can reach up to 5° of elevation on the ocean to reach gateway GSes on the land area, compromising on power density, as they do not have plans of using ISLs yet.

Key takeaway(4):- While too extreme values are unfavorable, e should be tuned based on other parameter values to unleash better performance.

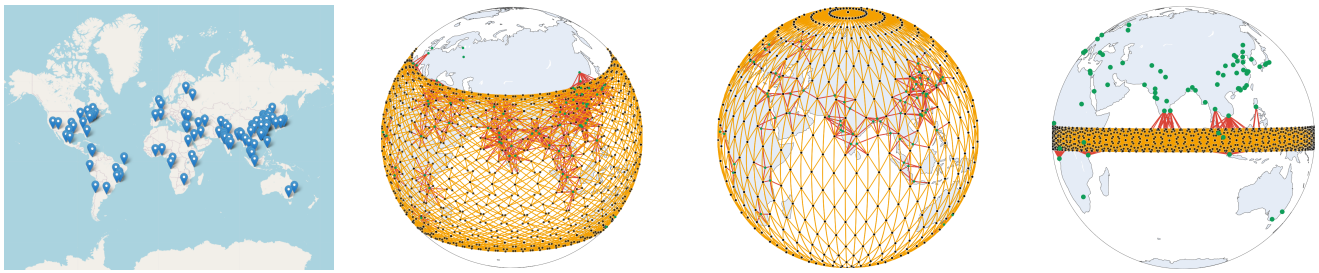
Performance impact of the number of orbits (o) **and phase offset** (p)**:** Since all the orbits have the same number of satellites, changing the number of orbits o inherently means adjusting the number of satellites n per orbit to maintain a fixed number of satellites. These orbits are equally spaced on the Earth's equator. Hence, a design with few orbits generates high east-west path diversity due to the large number of ISLs between satellites in adjacent orbits, as shown in Fig. 11(a). On the other hand, routes with high geodesic distances experience higher hop counts, as these routes follow intra-orbital ISLs to reach the destination GSes. Similarly, a design with many orbits reduces both hop counts in the high geodesic distance and east-west path diversity, as shown in Fig 11(b).

The phase offset p decides the relative positioning of satellites between adjacent orbits, i.e., in a constellation with 0.5 phase offset, a satellite n in orbit $(o+1)$ will be in the middle of satellite n and $(n+1)$ of orbit o, hence creating zigzag



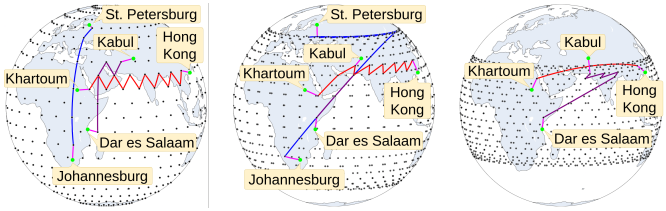
(a) Throughput and coverage (b) Stretch (c) End-to-end hop counts

Fig. 6: Performance measurement of i from 0° to 90° is symmetric to the measurement of 90° to 180°. (a) 40° provides best throughput (left Y-axis) and coverage (right Y-axis), too low/high i impacts negatively. (b), (c) Polar i provides better stretch (right Y-axis) and hop counts for *NS routes* and lower i provides better stretch (left Y-axis) and hop counts for other routes.



(a) GS locations (b) At $i = 40°$ (c) At $i = 90°$ (d) At $i = 5°$

Fig. 7: (a) Location of the GSes at the most popular cities. (b) $i = 40°$ restricts the trajectory of satellites on main population centers. (c) Higher i provides denser coverage at the polar region, whereas (d) lower i leaves most of *GSes* out of coverage.



(a) At $i = 90°$ (b) At $i = 50°$ (c) At $i = 30°$

Fig. 8: (a) Higher i provides better stretch with straight *NS* routes and zigzag ISLs for *EW* and *NESW* routes inflate the stretch. (b) Inclined trajectory provides better stretch for *NESW* and slightly improves *EW* stretch, whereas (c) very low i provides better EW stretch, leaves end-to-end *GSes* of *NS* out of coverage.

EW routes as shown in Fig. 11(d). On the other hand, with 0 phase offset, the ISL creates a precise +Grid shape network, as shown in Fig. 11(c). Hence, it is essential to note that these three parameters o, n, and p together describe the topological structure of the satellite mesh network. Therefore, after deeply exploring and analyzing the high-dimensional search space, our key discoveries are as follows:

(i) When the values of h, i, and e are close to optimal values, any number of orbit o which is sufficiently larger than the number of satellites per orbit n ($o >> n$) at a phase offset $p = 0.5$ produce higher throughput for any sufficiently large constellation. This behavior is shown in Fig. 12(a) for Starlink's first shell, where 100 GSes are located across most populated cities and the optimized value of i and e is chosen to 40° and 15° (§4) respectively. Notice that when the number of orbits is sufficiently large ($o > 66$), it gives a clear trend of increase in throughput with higher p.

(ii) This behavior is consistent across all the shells of Starlinks and Kuipers where the number of satellites is sufficiently large (≥ 300 satellites). The reason behind this behavior is demonstrated in Fig. 12(b)-(c). Notice that as we move towards higher o and p, median hop counts reduce, and the path diversity increases for HG routes without impacting other routes (i.e., NS, EW, NESW, LG); as a result, the constellation accommodates more traffic demands. The same is demonstrated between two GSes located in South Asia and North America in Fig. 13. When the constellation is designed with large o at $p = 0.5$, these two GSes experience high path diversity. Reducing either o or p negatively impacts the path diversity.

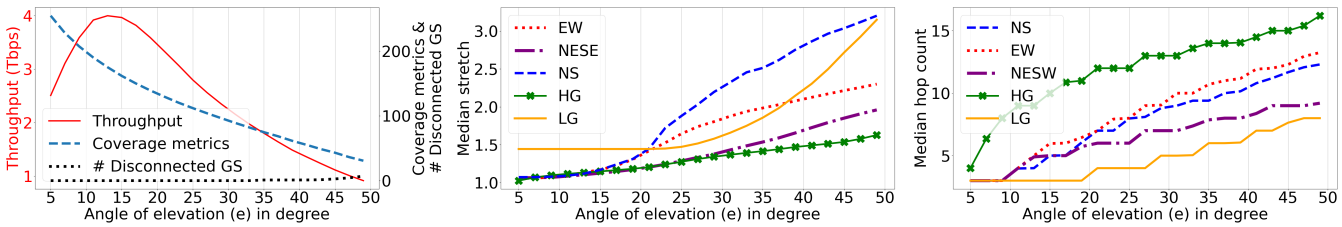
(iii) Change in the value of o or p essentially tweaks the mesh network at space, i.e., changes the ISL topological structures, which is illustrated in Fig.11. Therefore, when h, i, and e are fixed, any combination of o (hence n) and p produces consistent coverage.

> Key takeaway(5):- For a sufficiently large constellation (≥ 300 satellites), using larger o ($o >> n$) with a phase offset of 0.5 increases throughput.

5 Shaping an optimization strategy

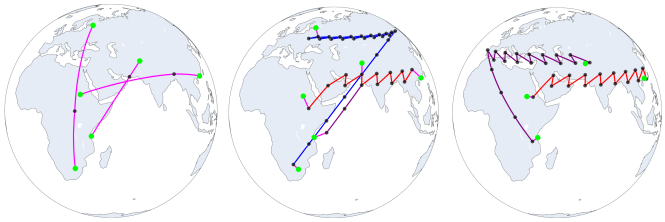
Now, we narrow down the search space using domain knowledge and key takeaways from the previous section. The outlined strategy helps optimization techniques converge faster.

Reducing the search space: As discussed earlier, the design of a single shell of a constellation deals with six parameters, i.e., h, i, e, o, n, and p. Since the topological structure is consistent and performance fluctuation is not significant over time, t could be ignored (takeaway (1)). Based on the observation of



(a) Throughput and coverage (b) Stretch (c) End-to-end hop counts

Fig. 9: Lower e provides better (a) coverage (right Y-axis) and throughput up to a certain point (left Y-axis). (b), (c) Lower e also provides better stretch and hop counts, whereas higher e (≥ 50) start creating coverage gaps.



(a) At $e = 5°$ (b) At $e = 30°$ (c) At $e = 50°$

Fig. 10: (a) Too low e enables cross-continent single hop routes with over the horizon *GSLs*, whereas (b) higher e reduces coverage area, thus, cross-continent *NS*, *EW*, and *NESW* routes increase up to 10s of *satellite* hops. (c) Too high e creates coverage gaps, hence, leaves *GSes* of *NS* routes out of coverage.

coverage metrics, these six parameters can be classified into two groups:

- GROUP-I: Set of parameters that impacts the constellation's coverage, i.e., h, e, and i.
- GROUP-II: Set of parameters that do not impact coverage, i.e., o, n, and p.

Now, notice that when the parameters of GROUP-I are optimized, then in GROUP-II, any number of orbits o which is sufficiently larger than the number of satellites per orbit n at phase offset $p = 0.5$ produce higher throughput than any other combinations (takeaway (5)). Hence, in our search strategy, we keep o to its maximum value (which will determine the value of n as well), $p = 0.5$, and then optimize the values of h, e, and i. Therefore, the search space is reduced to these three parameters of GROUP-I.

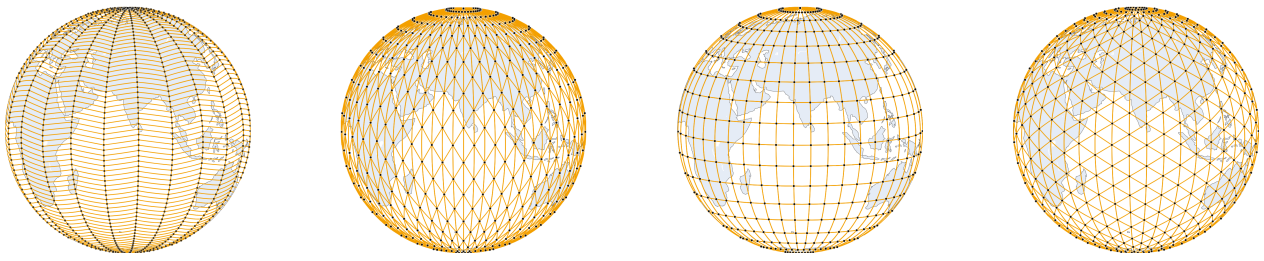
Furthermore, the scope of varying h is limited by various non-networking factors, i.e., FCC, ITU regulation [19, 24], radiation hazards [73, 78], rate of orbital decays [86], etc. Apart from that, because of symmetric performance outcomes (takeaway (3)), we restrict the parameters i up to 90°. It is also desirable to avoid too low inclination to reduce the interference with GEO satellites that operate over the Equator [26]. The range of e is also limited by the satellites and GSes communication hardware, which is not incorporated in our implementation.

Proposed search strategy: After trimming down the search space, `LEOCraft` tries to optimize the values of i, e, and h by using *Variable Neighborhood Search (VNS)* [57, 74] strategy. VNS starts with an initial solution x; in every iteration, it discovers the neighborhood of the current solution $N(x)$ and takes the best neighborhood. The process is repeated until the convergence criteria are reached (i.e., the solution starts saturating). In `LEOCraft`, we use a random step size at every iteration. While choosing the initial solution, we observe that $i \approx 30°$ and e between 10° to 20° converge faster. The intuition behind this is that the median latitude of the 100 most populous cities is 29.6°, and lower elevation gives good throughput for high path diversity (takeaway (4)).

6 Designing of `LEOCraft`

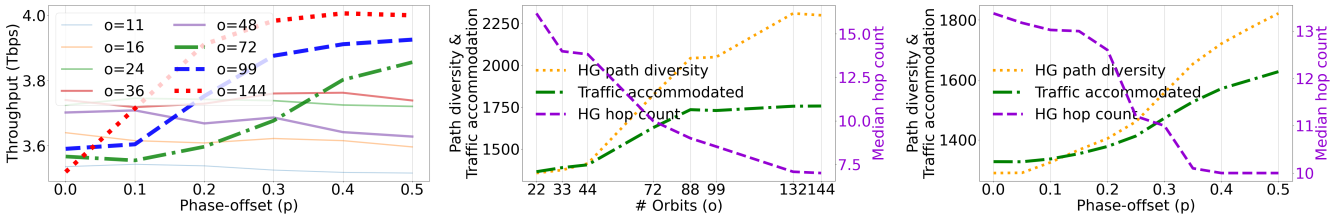
Overview of other simulation platforms: The well-known LEO network simulation and emulation platforms in the community are *Hypatia* [64], *StarryNet* [67], and *xeoverse* [63]. *xeoverse* is a relatively new LEO network emulation platform built upon network emulation platform *Mininet* [20]. *xeoverse* claims to outperform *Hypatia* and *StarryNet* by large margins [63]; however, *xeoverse* is not public for community use. On the other hand, *StarryNet* is a data-driven emulation platform partially available [70] to the community. *StarryNet* uses *Docker containers* [15] to emulate each node (satellites, GSes, user terminals) and Python's thread-based parallelism [32] for updating the state of virtual links (ISLs and GSLs) connecting containers [63]. Thus, *StarryNet* is resource-intensive, and scalability by default is constrained by the Docker engine's upper limit of its bridge interface (up to 1,023 containers [13] on a single machine [63]). Due to CPython's Global Interpreter Lock (GIL) [17], the link state update procedure of *StarryNet* with thousands of threads experiences a bottleneck, too. *Hypatia* is based upon ns-3 packet-level simulation platform [22]; hence, the scalability is constrained by the ns-3's limitations. Since *Hypatia* fails to utilise multiple CPUs, simulation execution drastically slows down with larger constellation [63, 64].

System design: After exploring the implementation of the above platforms, we found that a significant portion of the performance bottleneck arises from the software layer. This is because the LEO satellite constellation is treated as a single, monolithic block. However, within this block, most computations are independent of one another. For example, determining the coverage area of one satellite does not depend on calculations for other satellites. Similarly, computing a route from one GS to another is independent of the routes



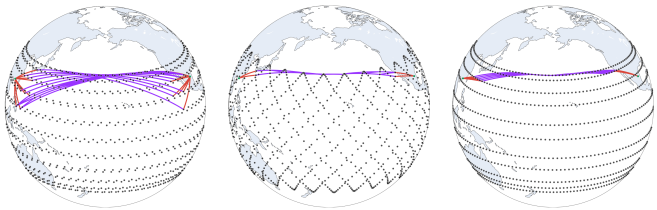
(a) $o = 22, n = 72$ (b) $o = 72, n = 22$ (c) $p = 0$ (d) $p = 0.5$

Fig. 11: Parameter o, n, and p together determine the *ISL* topological structures. (a), (b) n and o manipulates height and width to deform +Grid topological structure to rectangular grid cells. (c) Lower p forms a perfect +Grid topology, (d) higher p transforms +Grid to diamond ($\diamond$) shaped cells.



(a) Changing o and p (b) Changing o while $p = 0.5$ (c) Changing p while $o = 72$

Fig. 12: When i and e are set to 40° and 15°, (a) larger o (≥ 66) at a higher p provide better throughput. (b), (c) Such combinations reduce the *hop counts* (right Y-axis) and increase the *path diversity* for HG routes, and therefore accommodates *more traffic* for both parameter o in and p.



(a) $o = 144, p = 0.5$ (b) $o = 48, p = 0.5$ (c) $o = 144, p = 0$

Fig. 13: 20 shortest *HG routes* from South Asia to North America (a) have way higher path diversity with $o = 144$ and $p = 0.5$ than (b) lower o at $p = 0.5$ and (c) lower p at $o = 144$.

between other pairs of GSes, and so on. We exploit this in the implementation of `LEOCraft` [35], where each component (such as satellites and GSes) operates as an independent block with format-restricted data exchange APIs. This allows us to distribute workloads uniformly across all available CPUs, effectively shifting the performance bottleneck from the software layer to the underlying hardware.

In Fig. 14, we represent `LEOCraft`'s simulation workflow. In `LEOCraft` *LEO constellation builder* creates a top-level instance of *LEO constellation* based on specified design parameters and GS locations. This *LEO constellation* instance consists of all LEO network component instances (GSes, satellites, and ISLs) in it. The *LEO Constellation Simulator* acts as an execution framework in `LEOCraft`. It accumulates *LEO constellation* instances in the *Task queue* and evaluates the given batch of instances in parallel. This parallelism is at the functional level across all the *LEO constellation* instances in the *Task queue*. For that, *LEO Constellation Simulator* creates a pool of worker processes corresponding to each available CPU [14]. These worker processes remain active for the entire duration of the simulation execution, thus minimizing process creation overhead too. Each worker receives data (instances of LEO network components) and function references, executes these functions on the data, and sends the results back to the respective *LEO Constellation* instance through the dispatcher. The *LEO Constellation* accumulates the results of each asynchronously called function. Once all the function calls return, it writes back the evaluation results. Then *LEO Constellation Simulator* removes the instance from the *Task queue*.

This process-based concurrency [14] effectively bypasses the GIL [17], the primary performance bottleneck for CPU-bound tasks, allowing uniform use of all CPUs even for a single *LEO Constellation* instance in the *Task queue*. As a result, `LEOCraft` evaluates a large constellation within a few minutes. For the interested readers, §A.5 provides an overview of `LEOCraft` APIs for simulating any LEO constellation.

7 Performance evaluation of `LEOCraft`

We now evaluate the performance of `LEOCraft` to examine constellation-scale behavior. We compare the effect of different traffic matrices, optimization strategies, inter-shell connectivity, etc., using `LEOCraft`. In addition to these, we also compare and evaluate the performance of `LEOCraft` against existing platforms in §8.

Comparison with other optimization techniques: We now evaluate `LEOCraft` with different optimization strategies while scaling the constellation sizes. Specifically, we compare the network performance and running time of `LEOCraft` with four optimization techniques, i.e., *Simulated Annealing (SA)* [85], *Differential Evolution (DE)* [83], *Adaptive Particle Swarm Optimization (A-PSO)* [81], and *Variable Neighbourhood Search (VNS)*. The implementations of *A-PSO* and *VNS* are integrated into the `LEOCraft` framework, whereas for *SA* and *DE*, we use *SciPy*'s `optimize` APIs [25]. We use a max-

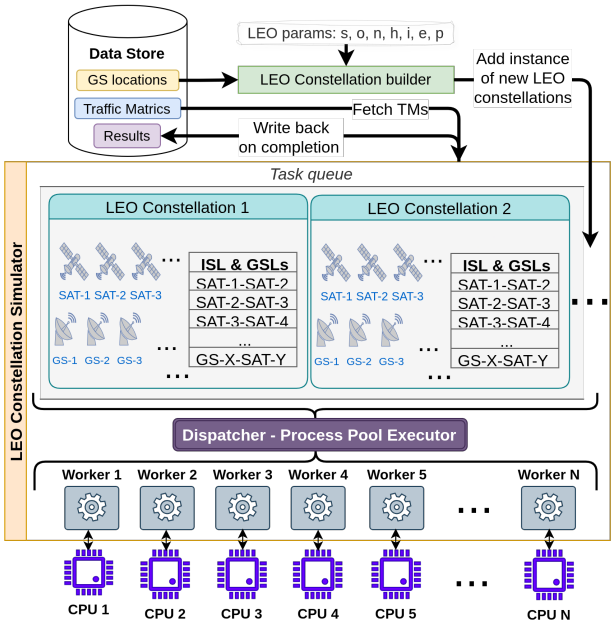


Fig. 14: LEOCraft's simulation framework.

imum iteration of 100 (60) for *SA* when domain knowledge of the search space is not in use (domain knowledge in use[1]). Similarly, for *A-PSO*, the number of iterations and particles is 25 and 20 (25 and 10), respectively, when domain knowledge is not in use (domain knowledge in use). These hyperparameters are empirically decided upon through a series of trials of synthetic constellations of different sizes. We evaluate the running time of these optimization techniques on a workstation powered by 16 core (24 thread) 12th Gen Intel(R) Core(TM) i9-12900 with 64 Gigabytes of memory.

The time taken to optimize the design parameters of different constellation sizes using different optimization techniques is shown in Fig. 15(a). Notice that when the same optimization techniques are used after pruning the search space using domain knowledge (§5), the average running time is reduced by ∼2.1×, ∼4.2×, ∼2.2×, and ∼13.7× for *SA*, *DE*, *A-PSO*, and *VNS*, respectively. Interestingly, *VNS* with domain knowledge outperforms all three metaheuristic techniques, as shown in Fig. 15(a). *VNS* is on average ∼2.2× times faster than the fastest metaheuristic technique when the search space is pruned with domain knowledge and at least on average ∼4.9× faster than the naive approaches. It is worth noting that *DE*, *VNS*, and *A-PSO* evaluate the cost function in parallel (evaluation of different design parameter combinations). In contrast, *SA* executes serially. However, if the running time is not the constraint, then the optimized design parameters produced by these four techniques are pretty similar. This is observed from Fig. 15(b)-(c), where we compare the throughput and stretch of NS routes of these optimized designs. Notice that the measured throughput varies marginally (standard deviation ±60 Gbps for 3,888 satellites) in Fig. 15(b), and the median stretch of NS routes deviation remains within 0.3, which is negligible variation as compared with the LEO dynamics (takeaway (1)).

> Outcome(1):- Insights drawn from extensive simulations can greatly reduce the search time for suitable constellation parameters.

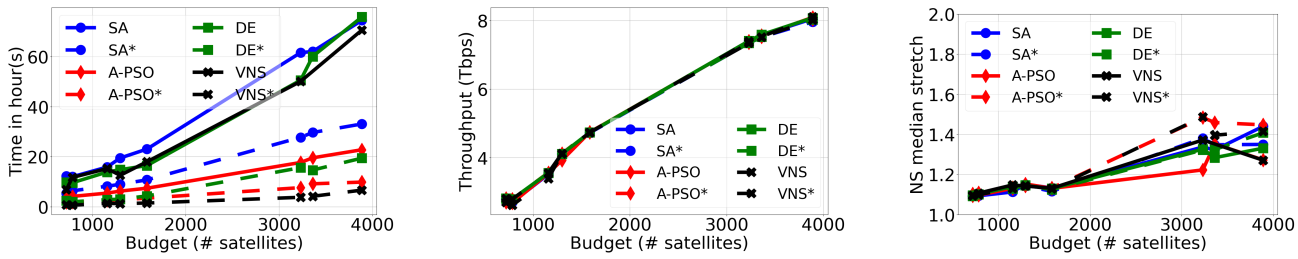
Single-shell vs Multi-shell design: We now explore the impact of multiple shells on the network throughput. For that, we take Starlink's first three shells, where the first two have 1,584, and the last one has 720 satellites, with altitudes of 540, 550, and 570 km [21], respectively. We do not consider inter-shell connectivity in this set of experiments. The results are shown in Fig. 16(a). Since these shells are independent of each other, we optimized the design of these three shells separately for our 100 GSes for the *High population TM*. The optimized Starlink constellation with three shells produced a throughput of 7.5 Tbps. Then, we merged the total budget of satellites into one shell, i.e., a single shell with 3,888 satellites produces a throughput of 8 Tbps when optimized. Further, we test the same while merging the satellite budget of any two shells out of three; thus, the constellation with two optimized shells produced throughput between 7.5 and 8 Tbps as illustrated in Fig. 16(a). We also test the same with a Kuiper budget of three shells with 1,156, 1,296, and 784 satellites, respectively, at altitudes 590, 610, and 630 km, respectively [1]. The observations are similar, i.e., the throughput observed when three shells are separately optimized is 6.6 Tbps, and when all satellites are merged and optimized in one shell, it is 7.4 Tbps. And any other combination falls in between these two measurements illustrated in Fig. 16(b).

> Outcome(2):- A single dense shell is more beneficial when higher throughput is the objective. A dense mesh network provides better path diversity for n shortest routes than multiple shells with sparser satellites.

Inter-shell ISL connectivity: However, designing a large constellation in one shell for higher throughput may increase the risk of collision [69,75]. This motivates us to explore inter-shell ISLs where we maintain the same +Grid ISL topology across the satellites deployed in different shells. Therefore, in LEOCraft, we simulate three shells of Starlink and Kuiper with inter-shell links. After optimizing the design, the throughput of these constellations reaches 8.01 and 7.34 Tbps for Starlink and Kuiper, respectively. This is almost equal to the throughput of merging the entire budget of a satellite into a single shell since a few 10s of km difference in altitude has a negligible impact on throughput (takeaway (2)).

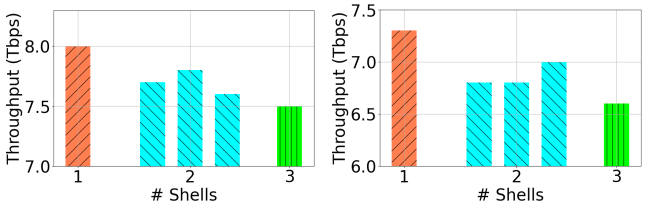
The main drawback of this inter-shell connectivity is the variation of phase offset with time. Since the orbital periods of satellites at higher altitudes are slightly lower than those at

[1]Here "using domain knowledge" refers to running meta-heuristics within the reduced search space discussed in §5. In contrast, "without domain knowledge" excludes evaluations over 24 hours, as they fail to converge.



(a) (b) (c)

Fig. 15: (a) Comparing the running time of black-box optimization techniques without and with domain knowledge (shown with superscript '*'). Comparison of (b) throughput and (c) median stretch of NS routes after optimization of the constellation design.



(a) Starlink (b) Kuiper

Fig. 16: Comparing the throughput with multiple shells for (a) Starlink and (b) Kuiper.

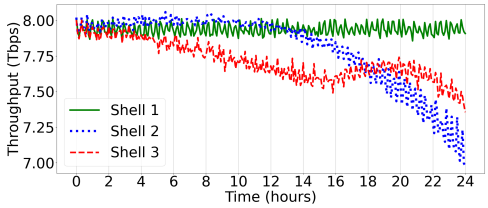


Fig. 17: Throughput measurement with inter-shell ISL connectivity over 24 hours without hand off.

lower altitudes, the phase offset will gradually deviate with time, and therefore, periodic hand-offs are needed to maintain the ISL connectivity and topological structure across the neighboring shells. However, we observe that the hand-off frequency for such a slight altitude difference (in 10s of km) is low. Fig. 17 shows the throughput measurement over a day when Starlink's 3,888 satellites are deployed in different shells, and inter-shell ISL connectivity is established in case of multiple shells. As expected, the throughput remains mostly consistent with minor fluctuation over time for the single shell design. For multi-shell design, deformation of the ISL topology due to the phase offset shift with time negatively impacts the throughput. We can also observe that a handoff after $\sim$ 13 and $\sim$ 4 hours may be needed for maintaining the inter-shell ISL topology across two and three shells, respectively.

Outcome(3):- Inter-shell ISLs enhance throughput slightly but require occasional handoffs to maintain connectivity and topology between shells.

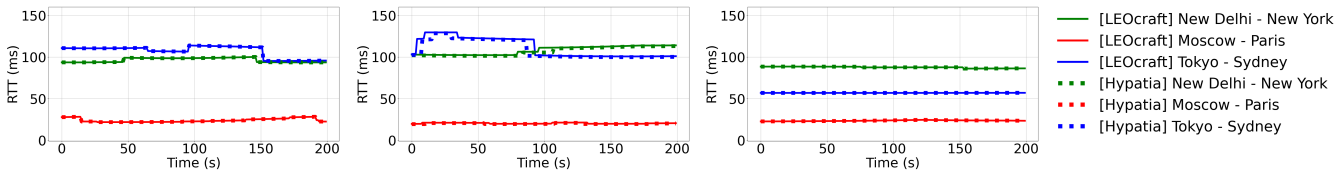
Visualization features: `LEOCraft` provides a range of visualization features, such as routes, satellite coverage, and constellation topologies. Some figures such as Fig. 1, 2, 7, 8, 10, 11, 13 are snapshots of the LEO network design's interactive and responsive visualization. Multi-shell mega constellations can be insanely dense networks that make it difficult to visualize particular areas of interest. `LEOCraft` visualization APIs allow cherry-picking of the components of interest (set of satellites, GSes, ISLs, GSLs, routes, etc.) for rendering. `LEOCraft` also generates frame-by-frame evolution of the network at the granularity up to nanoseconds to demonstrate how network topology and end-to-end routes change over time due to the LEO dynamics. Some additional details of visualization features are available in §A.3.

8 Comparing `LEOCraft` with other platform

Since *StarryNet* [67] does not scale up to a few thousand satellites on a single machine, and *xeoverse* [63] is not public, we choose to compare `LEOCraft` with *Hypatia* [64].

Packet-level vs. flow-level simulation: First, we highlight the scope of comparison and the fundamental differences between flow-level and packet-level simulations to avoid any misinterpretation for the readers. Notice that *Hypatia* is well-suited for analyzing detailed network protocol behaviors, like packet drops, retransmissions, or reordering due to handovers. A flow-level simulation like `LEOCraft` is more appropriate for examining architectural and policy-level trade-offs, such as the effects of ISL topologies, routing choices, and GSes placements on overall performance. Consequently, `LEOCraft` and *Hypatia* are not competitors; rather, they complement each other. A future simulation pipeline could utilize `LEOCraft` to uncover systematic design issues in LEO networks at scale, which could then be closely examined using *Hypatia*.

Latency comparison with *Hypatia*: In *Hypatia*, we simulate '`pings`' at 1 second interval between three GS pairs, i.e., Delhi to New York, Moscow to Paris, and Tokyo to Sydney through the shortest route over the first shell of Kuiper, Starlink, and Telesat for a duration of 200 seconds. Then we compute the round-trip time (RTT) in `LEOCraft` between the same pair of GSes over the same constellations. The results are shown in Fig. 18. From this figure, we can observe that the measured RTT obtained from *Hypatia* closely matches with the '`computed`' RTT of `LEOCraft`. The RTT of Moscow to Paris is one-third of the latency of Delhi to New York since the distance between Delhi to New York is $\sim 5\times$ than the distance between Moscow and Paris. In contrast, Tokyo to Sydney



(a) Kuiper first shell of 1,156 satellites (b) Starlink first shell of 1,584 satellites (c) Telesat first shell of 351 satellites

Fig. 18: *Hypatia*'s `Ping` closely matches with `LEOCraft`'s computed RTT for (a) Kuiper, (b) Starlink, and (c) Telesat.

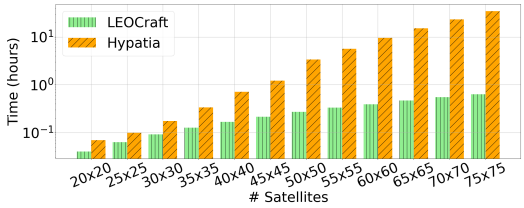


Fig. 19: Comparison of simulation times with *Hypatia* and `LEOCraft` for different constellation sizes.

routes experience latency equivalent to the HG route Delhi to New York, despite these two endpoints being relatively close to each other. This is because the orbital inclination of Starlink and Kuiper are 53.0° and 51.9° do not generate the straight route in the north-south orientation. In contrast, Telesat's inclination of 98.98° generates a straight route in a north-south orientation, hence Tokyo to Sydney latency is reduced by $\sim$ 50% as compared to Starlink and Kuiper.

Simulation time comparison with *Hypatia*: We now compare the simulation time of *Hypatia* and `LEOCraft` in Fig. 19, where we simulate the end-to-end latencies between 50 most populous cities while increasing the size (number of satellites) of a synthetic single shell +Grid constellation. The inclination angle, elevation angle, and altitude of the satellites are kept as 90°, 30°, and 1,000 km, respectively. From the figure, we can observe that the `LEOCraft` is $\sim$ 1.7 to 54.5$\times$ faster as compared to *Hypatia*, and this difference keeps increasing with the size of the constellation. While having more CPUs will further speed up the simulation in `LEOCraft` due to its design, which is not the case with *Hypatia*.

Additionally, to evaluate the scalability of `LEOCraft` with a hypothetical mega-constellation consisting of 20 highlighted shells in Table 1[2], which comes out to be a mega-constellation of 83K satellites. We have observed that `LEOCraft` manages to simulate this mega-constellation within a week on a system powered by Intel Xeon Silver 4309Y (16) @ 3.6GHz and 128 Gigabytes of memory. This shows that `LEOCraft` is well-suited for simulating mega-constellations consisting of tens of thousands of satellites within a reasonable time.

9 Related work

Simulation and measurement: Several efforts [47, 63, 64, 67, 68] have focused on creating simulation platforms for measuring LEO networks. Notable examples include *Hypatia* [64], *StarryNet* [67], and *xeoverse* [63], their strengths and limitations has been briefly discussed in §6. Additionally, [46] explores RTT fluctuation using intuitive mathematical modeling, avoiding resource-intensive simulations.

LEO topology: Most of recent studies [40, 47, 63, 64, 67, 68, 92, 93] have widely adopted the +Grid topology as the de facto standard for LEO networks. In contrast, [45] proposed an improved ISL topology to minimize hand-offs and enhance throughput latency. Meanwhile, [72] highlighted the drawbacks of +Grid, advocating for $\times$Grid topology to reduce packet drops and hop counts. A recent study [92, 93] used *Hypatia* [64] to analyze single-shell LEO topologies with a limited set of design parameters. This work extends the analysis to all parameters, including multiple shells, optimizing satellite trajectories based on the traffic demand matrix between GSes.

10 Conclusion and future work

In this paper, we introduce `LEOCraft` [35], an open-source, modular, and scalable framework for evaluating and visualizing LEO constellations. `LEOCraft` analyzes high-dimensional search spaces to uncover the crux of LEO trajectory design that enables trimming of search space to optimize constellation design quicker. We also present findings on the performance gain for inter-shell ISL connectivity.

In the future, we plan to extend the `LEOCraft` towards topology and trajectory joint optimization [45, 72]. While this work focuses on maximizing throughput, mega-constellation operators with humongous budgets often target diverse sectors. For instance, Starlink's deployment of two polar-orbit shells improves connectivity in the polar region [27–29]. Additionally, seeks FCC approval for VLEO deployment to enable low-latency applications like fin-tech and gaming [16]. Optimizing mega constellations for multiple objectives presents another promising research direction.

Acknowledgments

We express our gratitude to our shepherd, Ivan Beschastnikh, and the anonymous reviewers for their valuable feedback. This research is funded by the Prime Minister's Research Fellowship (PMRF) from the Ministry of Education, Government of India.

[2]The last two shells of Starlink's first generation are ignored as they only have 6 and 4 orbits respectively, which leads to a sparse LEO network with no inter-orbit connectivity.

References

[1] Kuiper systems llc (technical appendix). https://tinyurl.com/FCCKuiperSystems, 2019.

[2] Spacex adds laser crosslinks to polar starlink satellites. http://tinyurl.com/starlinkAddISLs, 2020. Accessed: 2023-12-20.

[3] Fcc partially grants spacex gen2. https://tinyurl.com/Gen2Approval, 2022.

[4] Spacex non-geostationary satellite system/attachment a. https://fcc.report/IBFS/SAT-LOA-20161115-00118/1158350.pdf, 2022. Accessed: 2024-08-04.

[5] Kuiper completes successful tests of optical mesh network in leo. https://tinyurl.com/KuiperTestsISLLEO, 2023.

[6] Mynaric selected by esa. https://tinyurl.com/MynaricSelected, 2023. Accessed: 2023-12-06.

[7] Next-generation condor mk3. https://tinyurl.com/MynaricReleases/, 2023. Accessed: 2023-12-06.

[8] Oneweb fcc report. https://fcc.report/IBFS/SAT-MPL-20200526-00062/2379706.pdf, 2023. Accessed: 2023-12-06.

[9] Space products condor mk3. https://tinyurl.com/CONDORMk3, 2023. Accessed: 2023-12-06.

[10] Starlink fcc report. https://fcc.report/IBFS/SAT-MOD-20190830-00087/1877671.pdf, 2023. Accessed: 2023-12-06.

[11] Starlink launches v2 mini-satellites with 'space lasers'. https://tinyurl.com/starlinkISL, 2023. Accessed: 2023-12-06.

[12] Application for modification of market access authorization. https://fcc.report/IBFS/SAT-MPL-20200526-00053/2378318.pdf, 2024. Accessed: 2024-07-05.

[13] Bridge network driver. https://docs.docker.com/network/drivers/bridge/, 2024.

[14] concurrent.futures — launching parallel tasks. https://docs.python.org/3/library/concurrent.futures.html#concurrent.futures.ProcessPoolExecutor, 2024.

[15] Docker. https://www.docker.com/, 2024. Accessed: 2024-11-16.

[16] Fcc denies starlink low-orbit bid for lower latency. https://spectrum.ieee.org/starlink-vleo-below-iss, 2024. Accessed: 2024-06-23.

[17] Global interpreter lock, or gil. https://wiki.python.org/moin/GlobalInterpreterLock, 2024. Accessed: 2024-11-16.

[18] In flight wi-fi market analysis. https://www.coherentmarketinsights.com/market-insight/in-flight-wi-fi-market-3806, 2024. Accessed: 2024-07-01.

[19] International satellite coordination. https://www.fcc.gov/space/international-satellite-coordination, 2024. Accessed: 2024-07-03.

[20] Mininet. https://mininet.org/, 2024. Accessed: 2024-11-16.

[21] Modification authorization spacex. https://fcc.report/IBFS/SAT-MOD-20200417-00037/2274315.pdf, 2024. Accessed: 2024-07-05.

[22] ns-3 network simulator. https://www.nsnam.org/, 2024. Accessed: 2024-11-16.

[23] Oneweb targets maritime market with expanded satellite coverage. https://tinyurl.com/OneWebTargetsMaritimeMarket, 2024. Accessed: 2024-11-16.

[24] Regulation of ngso satellite constellations. https://digitalregulation.org/regulation-of-ngso-satellite-constellations/, 2024. Accessed: 2024-07-03.

[25] Scipy, fundamental algorithms for scientific computing in python. https://scipy.org/, 2024. Accessed: 2023-12-06.

[26] Spacex v-band non-geostationary satellite system. https://fcc.report/IBFS/SAT-LOA-20170301-00027/1190019.pdf, 2024. Accessed: 2024-07-03.

[27] Starlink disrupting greenland internet market. https://tinyurl.com/StarlinkDisruptingGreenland, 2024. Accessed: 2024-06-23.

[28] Starlink expands coverage. https://tinyurl.com/StarlinkExpandsCoverage, 2024. Accessed: 2024-06-23.

[29] Starlink internet service goes live in alaska. https://tinyurl.com/StarlinkServiceLiveAlaska, 2024. Accessed: 2024-06-23.

[30] Starlink's laser system. https://tinyurl.com/StarlinkLaserSystem, 2024. Accessed: 2024-06-23.

[31] Telesat low earth orbit non-geostationary satellite system. https://fcc.report/IBFS/SAT-MPL-20200526-00053/2378320.pdf, 2024. Accessed: 2024-07-05.

[32] threading — thread-based parallelism. https://docs.python.org/3/library/threading.html, 2024. Accessed: 2024-11-16.

[33] Unofficial sdk for flightradar24 for python 3. https://pypi.org/project/FlightRadarAPI/, 2024. Accessed: 2024-06-23.

[34] World cities database. https://simplemaps.com/data/world-cities, 2024. Accessed: 2024-07-01.

[35] Leocraft source code. https://github.com/suvambasak/LEOCraft.git, 2025. Accessed: 2025-01-01.

[36] Mercator projection wiki. https://en.wikipedia.org/wiki/Mercator_projection, 2025.

[37] R. Akturan and W.J. Vogel. Path diversity for leo satellite-pcs in the urban environment. *IEEE Transactions on Antennas and Propagation*, 45(7):1107–1116, 1997.

[38] Shahram Amiri and Brian Reif. Internet penetration and its correlation to gross domestic product: An analysis of the nordic countries. *International Journal of Business, Humanities and Technology*, 3(2):50–60, 2013.

[39] Arthur H Ballard. Rosette constellations of earth satellites. *IEEE transactions on aerospace and electronic systems*, (5):656–673, 1980.

[40] Suvam Basak, Amitangshu Pal, and Debopam Bhattacherjee. Exploring low-earth orbit network design. In *Proceedings of the 1st ACM Workshop on LEO Networking and Communication*, pages 1–6, 2023.

[41] Jason H Bau. *Topologies for satellite constellations in a cross-linked space backbone network*. PhD thesis, Massachusetts Institute of Technology, 2002.

[42] Theresa W Beech, Stefania Cornara, Miguel Bell Mora, and GD Lecohier. A study of three satellite constellation design algorithms. In *14th international symposium on space flight dynamics*, 1999.

[43] Dhiraj Bhattacharjee, Aizaz U. Chaudhry, Halim Yanikomeroglu, Peng Hu, and Guillaume Lamontagne. Laser inter-satellite link setup delay: Quantification, impact, and tolerable value. In *2023 IEEE Wireless Communications and Networking Conference (WCNC)*, pages 1–6, 2023.

[44] Debopam Bhattacherjee, Waqar Aqeel, Ilker Nadi Bozkurt, Anthony Aguirre, Balakrishnan Chandrasekaran, P Brighten Godfrey, Gregory Laughlin, Bruce Maggs, and Ankit Singla. Gearing up for the 21st century space race. In *Proceedings of the 17th ACM Workshop on Hot Topics in Networks*, pages 113–119, 2018.

[45] Debopam Bhattacherjee and Ankit Singla. Network topology design at 27,000 km/hour. In *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies*, pages 341–354, 2019.

[46] Vaibhav Bhosale, Ketan Bhardwaj, and Ahmed Saeed. Astrolabe: Modeling rtt variability in leo networks. In *Proceedings of the 1st ACM Workshop on LEO Networking and Communication*, pages 7–12, 2023.

[47] Xuyang Cao and Xinyu Zhang. Satcp: Link-layer informed tcp adaptation for highly dynamic leo satellite networks. In *IEEE INFOCOM 2023 - IEEE Conference on Computer Communications*, 2023.

[48] Michel Capderou. Orbit and ground track of a satellite. *Satellites: Orbits and Missions*, pages 175–263, 2005.

[49] María García Colón, Elena Martínez Lizuain, Felix Mora-Camino, and Antoine Drouin. Design of air corridor structures for enhanced traffic performance. In *2016 IEEE/AIAA 35th Digital Avionics Systems Conference (DASC)*, pages 1–7, 2016.

[50] Inigo Del Portillo, Bruce G Cameron, and Edward F Crawley. A technical comparison of three low earth orbit satellite constellation systems to provide global broadband. *Acta astronautica*, 159:123–135, 2019.

[51] Karthick Dharmarajan. Coverage optimization of satellite formations using instantaneous overlap area. In *2022 IEEE 9th International Workshop on Metrology for AeroSpace (MetroAeroSpace)*, pages 582–587, 2022.

[52] John V Evans. Satellite systems for personal communications. *Proceedings of the IEEE*, 86(7):1325–1341, 1998.

[53] J. Fomon. Speedtest data. https://tinyurl.com/ooklaReport, 2024. Accessed: 2024-06-23.

[54] Johan Garcia, Simon Sundberg, Giuseppe Caso, and Anna Brunstrom. Multi-timescale evaluation of starlink throughput. In *Proceedings of the 1st ACM Workshop on LEO Networking and Communication*, pages 31–36, 2023.

[55] Mark Handley. Delay is not an option: Low latency routing in space. In *Proceedings of the 17th ACM Workshop on Hot Topics in Networks*, pages 85–91, 2018.

[56] Mark Handley. Using ground relays for low-latency wide-area routing in megaconstellations. In *Proceedings of the 18th ACM Workshop on Hot Topics in Networks*, pages 125–132, 2019.

[57] Pierre Hansen, Nenad Mladenović, and Jose A Moreno Perez. Variable neighbourhood search: methods and applications. *Annals of Operations Research*, 175:367–407, 2010.

[58] Yannick Hauri, Debopam Bhattacherjee, Manuel Grossmann, and Ankit Singla. " internet from space" without inter-satellite links. In *Proceedings of the 19th ACM Workshop on Hot Topics in Networks*, pages 205–211, 2020.

[59] Felix R Hoots and Ronald L Roehrich. *Models for propagation of NORAD element sets*. Office of Astrodynamics, 1980.

[60] Syed Kamrul Islam and Mohammad Rafiqul Haider. *Sensors and low power signal processing*. Springer Science & Business Media, 2009.

[61] Spacex preparing to start starlink gen2 launches this month, 2022. https://spacenews.com/spacex-preparing-to-start-starlink-gen2-launches-this-month/.

[62] David S Johnson, Jan Karel Lenstra, and AHG Rinnooy Kan. The complexity of the network design problem. *Networks*, 8(4):279–285, 1978.

[63] Mohamed M Kassem and Nishanth Sastry. *x* eoverse: A real-time simulation platform for large leo satellite mega-constellations. In *2024 IFIP Networking Conference (IFIP Networking)*, pages 1–9. IEEE, 2024.

[64] Simon Kassing, Debopam Bhattacherjee, André Baptista Águas, Jens Eirik Saethre, and Ankit Singla. Exploring the" internet from space" with hypatia. In *Proceedings of the ACM Internet Measurement conference*, pages 214–229, 2020.

[65] John D Kiesling. Land mobile satellite systems. *Proceedings of the IEEE*, 78(7):1107–1115, 1990.

[66] Kenneth Chun Hei Kwok. *Cost optimization and routing for satellite network constellations*. PhD thesis, Massachusetts Institute of Technology, 2001.

[67] Zeqi Lai, Hewu Li, Yangtao Deng, Qian Wu, Jun Liu, Yuanjie Li, Jihao Li, Lixin Liu, Weisen Liu, and Jianping Wu. {StarryNet}: Empowering researchers to evaluate futuristic integrated space and terrestrial networks. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 1309–1324, 2023.

[68] Zeqi Lai, Hewu Li, and Jihao Li. Starperf: Characterizing network performance for emerging mega-constellations. In *2020 IEEE 28th International Conference on Network Protocols (ICNP)*, pages 1–11. IEEE, 2020.

[69] HG Lewis and G Skelton. Safety considerations for large constellations of satellites. *LPI Contributions*, 2852:6099, 2023.

[70] Wenhao Lu, Zhiyuan Wang, Shan Zhang, Qingkai Meng, and Hongbin Luo. Opensn: An open source library for emulating leo satellite networks. In *Proceedings of the 8th Asia-Pacific Workshop on Networking*, pages 149–155, 2024.

[71] R David Luders. Satellite networks for continuous zonal coverage. *ARS Journal*, 31(2):179–184, 1961.

[72] Joseph Mclaughlin, Jee Choi, and Ramakrishnan Durairajan. $\times$ grid: A location-oriented topology design for leo satellites. In *Proceedings of the 1st ACM Workshop on LEO Networking and Communication*, pages 37–42, 2023.

[73] Rositsa Miteva, Susan W Samwel, and Stela Tkatchova. Space weather effects on satellites. *Astronomy*, 2(3):165–179, 2023.

[74] Nenad Mladenović and Pierre Hansen. Variable neighborhood search. *Computers & operations research*, 24(11):1097–1100, 1997.

[75] Michael Nicolls and Darren McKnight. Collision risk assessment for derelict objects in low-earth orbit. In *Proceedings of First International Orbital Debris Conference, Sugar Land, TX*, pages 9–12, 2019.

[76] Nils Pachler, Inigo del Portillo, Edward F Crawley, and Bruce G Cameron. An updated comparison of four low earth orbit satellite constellation systems to provide global broadband. In *2021 IEEE international conference on communications workshops (ICC workshops)*, pages 1–7. IEEE, 2021.

[77] L Rider. Optimized polar orbit constellations for redundant earth coverage. *Journal of the Astronautical Sciences*, 33:147–161, 1985.

[78] Jakub Řípa, Giuseppe Dilillo, Riccardo Campana, and Gábor Galgóczi. A comparison of trapped particle models in low earth orbit. In *Space Telescopes and Instrumentation 2020: Ultraviolet to Gamma Ray*, volume 11444, pages 597–606. SPIE, 2020.

[79] Matthew Roughan. Simplifying the synthesis of internet traffic matrices. *ACM SIGCOMM Computer Communication Review*, 35(5):93–96, 2005.

[80] John P Rula, James Newman, Fabián E Bustamante, Arash Molavi Kakhki, and David Choffnes. Mile high wifi: A first look at in-flight internet connectivity. In *Proceedings of the 2018 world wide web conference*, pages 1449–1458, 2018.

[81] Georgios Sermpinis, Konstantinos Theofilatos, Andreas Karathanasopoulos, Efstratios F Georgopoulos, and Christian Dunis. Forecasting foreign exchange rates with adaptive neural networks using radial-basis functions and particle swarm optimization. *European Journal of Operational Research*, 225(3):528–540, 2013.

[82] William Stallings. *Data and computer communications*. Pearson Education India, 2007.

[83] Rainer Storn and Kenneth Price. Differential evolution–a simple and efficient heuristic for global optimization over continuous spaces. *Journal of global optimization*, 11:341–359, 1997.

[84] Arjun Tan. *Theory of Orbital motion*. World Scientific, 2008.

[85] Constantino Tsallis and Daniel A Stariolo. Generalized simulated annealing. *Physica A: Statistical Mechanics and its Applications*, 233(1-2):395–406, 1996.

[86] Joshi Om Vaibhav, Lee Xun Yong, and Samuel Joo Jian Wen. In-orbit lifetime of satellites.

[87] John Gerard Walker. *Circular orbit patterns providing continuous whole earth coverage*. Royal Aircraft Establishment, Ministry of Aviation Supply, 1970.

[88] Bingsen Wang, Xiaohui Zhang, Shuai Wang, Li Chen, Jinwei Zhao, Jianping Pan, Dan Li, and Yong Jiang. A large-scale ipv6-based measurement of the starlink network. *arXiv preprint arXiv:2412.18243*, 2024.

[89] Lloyd Wood. *Internetworking with satellite constellations*. University of Surrey (United Kingdom), 2001.

[90] William W Wu, Edward F Miller, Wilbur L Pritchard, and Raymond L Pickholtz. Mobile satellite communications. *Proceedings of the IEEE*, 82(9):1431–1448, 1994.

[91] Jin Y Yen. Finding the k shortest loopless paths in a network. *management Science*, 17(11):712–716, 1971.

[92] Wenyi Zhang, Zihan Xu, and Sangeetha Abdu Jyothi. An in-depth investigation of leo satellite topology design parameters. In *Proceedings of the 2nd International Workshop on LEO Networking and Communication*, pages 1–6, 2024.

[93] Wenyi Morty Zhang, Zihan Xu, and Sangeetha Abdu Jyothi. A deep dive into leo satellite topology design parameters. In *International Conference on Passive and Active Network Measurement*, pages 247–275. Springer, 2025.

A Appendix

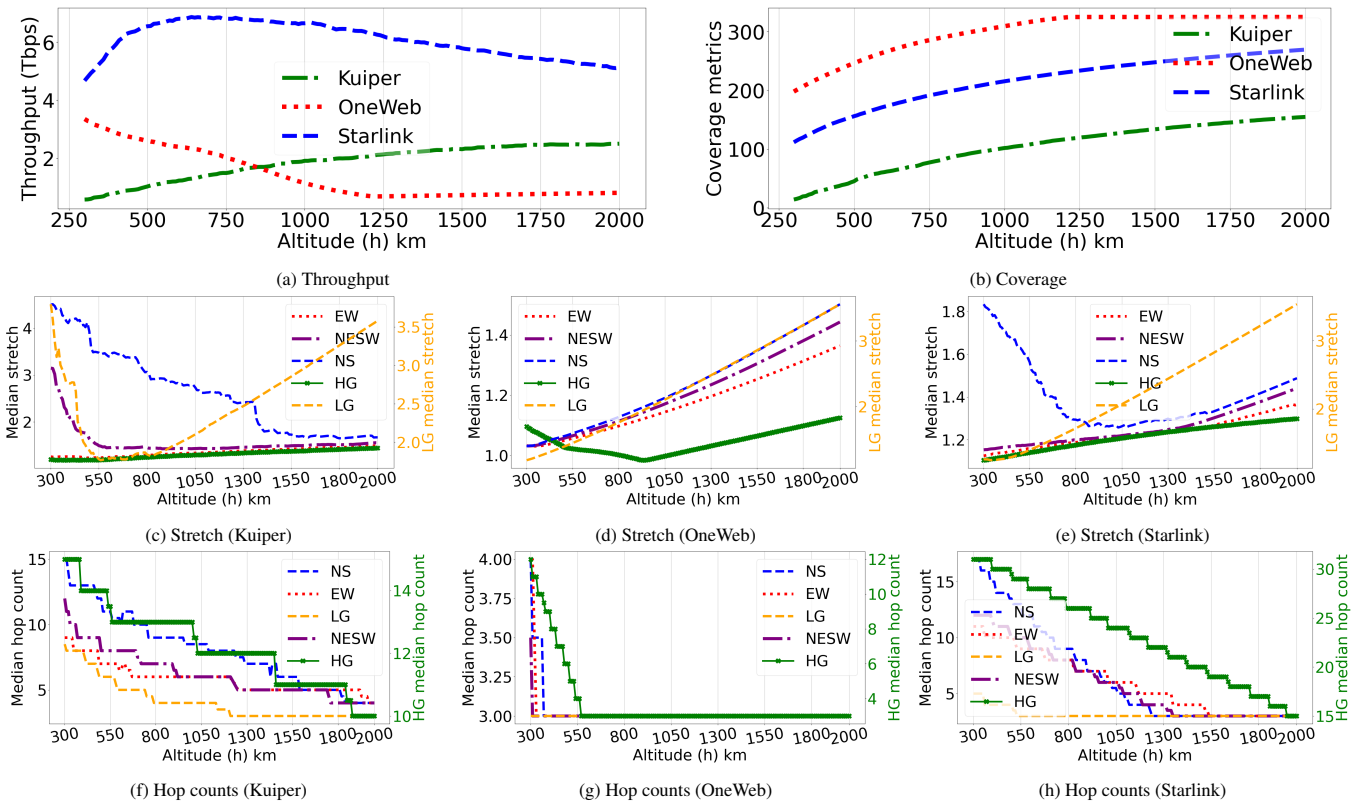
A.1 Comparison across various constellations

All LEO satellite network operators have proposed a wide variety of constellations so far. So, in this section, we evaluate the performance characteristics of different constellation designs to illustrate how different types of constellation designs impact performance. For that, we pick three shell configurations of different LEO operators shown in Table 1, i.e., (a) Kuiper's third shell, which is a relatively smaller shell of 784 satellites at a low inclination of 33°, (b) OneWeb's first shell[3], which is a moderate size shell of 1,764 satellites at polar inclination of 87.9° and relatively higher altitude of 1,200 km, and (c) Starlink Gen2's first shell, which is a large shell of 5,280 satellites at VLEO of altitude 340 km. These three shells cover a wide variety of constellation design choices at different altitudes and inclinations. Through detailed simulations, we observed that the performance characteristics of these shells with different TMs (§2.3) are consistent with each other, while varying one or more constellation design parameters, whereas others are kept at their default value as mentioned in Table 1. Therefore, for brevity, we restrict the following discussion to *High-population TM* only.

Altitude (h): The throughput, coverage, stretch, and hop count characteristics of these three shells while varying altitude h are shown in Fig. 20. For most of these performance metrics, all these constellations show similar characteristics. The only difference that we can observe is how the throughput trends change with altitude. Notice that Kuiper, with a low satellite budget of 784, produces higher throughput at relatively higher altitudes with larger coverages. In contrast, Starlink Gen2 shell, with a large satellite budget of 5,280, starts experiencing a decline in throughput and inflation in stretch (latency) above 500 km (900 km for NS routes) of altitude. This is evident since these large VLEO shells are focused on low-latency connectivity [16].

Interestingly, for OneWeb, the throughput trend is a bit odd as compared to others. This is because their operational altitude is as high as 1,200 km, whereas the elevation is too low at 5°. Since their target is the commercial sector and maritime market [23], their design is inclined towards offering globally reachable connectivity rather than achieving high performance for the population density.

[3] Notice that OneWeb does not have a plan for using ISL yet, but is open to incorporate it in future generations of satellites [8]. Therefore, in our analysis, we study +Grid ISL topology for OneWeb as well.



(a) Throughput (b) Coverage

(c) Stretch (Kuiper) (d) Stretch (OneWeb) (e) Stretch (Starlink)

(f) Hop counts (Kuiper) (g) Hop counts (OneWeb) (h) Hop counts (Starlink)

Fig. 20: (a) Throughput, (b) coverage (c)-(e) stretch and (f)-(h) hop counts measurement across most populous cities with varying altitude (h).

Inclination (i): The performance metrics of three constellations while varying inclination i are shown in Fig. 21. Notice that, except for OneWeb's stretch, coverage, and hop count, all other performance characteristics are consistent. Due to OneWeb's higher altitude with a low elevation angle, coverage and hop count do not show any noticeable impact due to the inclination change. However, a low angle of inclination restricts the availability of satellites near the equatorial regions and inflates the stretch of NS and LG routes.

Elevation (e): We show the performance characteristics while varying elevation e in Fig. 22. The performance metrics are consistent. Only LG routes of OneWeb experience higher stretch than usual due to a single satellite hop at an altitude of 1,200+ km between the nearby GS pairs.

Number of orbits (o) **and phase offset** (p): Similarly, the performance characteristics while varying the number of orbits o and phase offset p, are also consistent in Fig. 23 with our previous discussion in §4.

Outcome:- The characteristics of various performance metrics of different constellation sizes are pretty similar and predictable. Thus, knowledge of the search space in §4 could be used to intuitively trim the search space for different constellation sizes as well.

A.2 Comparison of traffic metrics

All the results discussed previously in §4 and in §A.1 are simulated with *High population TM*. In Fig. 24, we illustrate how performance characteristics change with other TMs for the first Starlink shell (Table 1). Since demands are concentrated in the regions with higher population density, we have observed that overall performance measurement characteristics are similar for all TMs across all the constellation design parameters. Therefore, we only demonstrate the impact of inclination i and elevation angle e here for brevity. Notice that the throughput for *High population TM* and *High GDP population TM* overlaps for both i and e in Fig. 24(a)-(b) except for the lower inclination. Since countries with high populations and lower GDP are mostly located close to Earth's Equators (i.e., lower latitude lines), lower inclination provides slightly higher throughput for *High population TM*. Whereas for *Country capital TM*, the constellation consistently provides higher throughput than the other TMs irrespective of the values of i and e. The GSes for *Country capital TM* are sparsely located across the capital of 233 countries, they do not experience much congestion around multiple high population/GDP cities located closely in China, India, USA, and Europe.

On the other hand, *Global flight TM* throughput measurements are consistently lower than other TMs. Our dataset includes total 8,384 flights on the air (> 10,000 feet altitude)

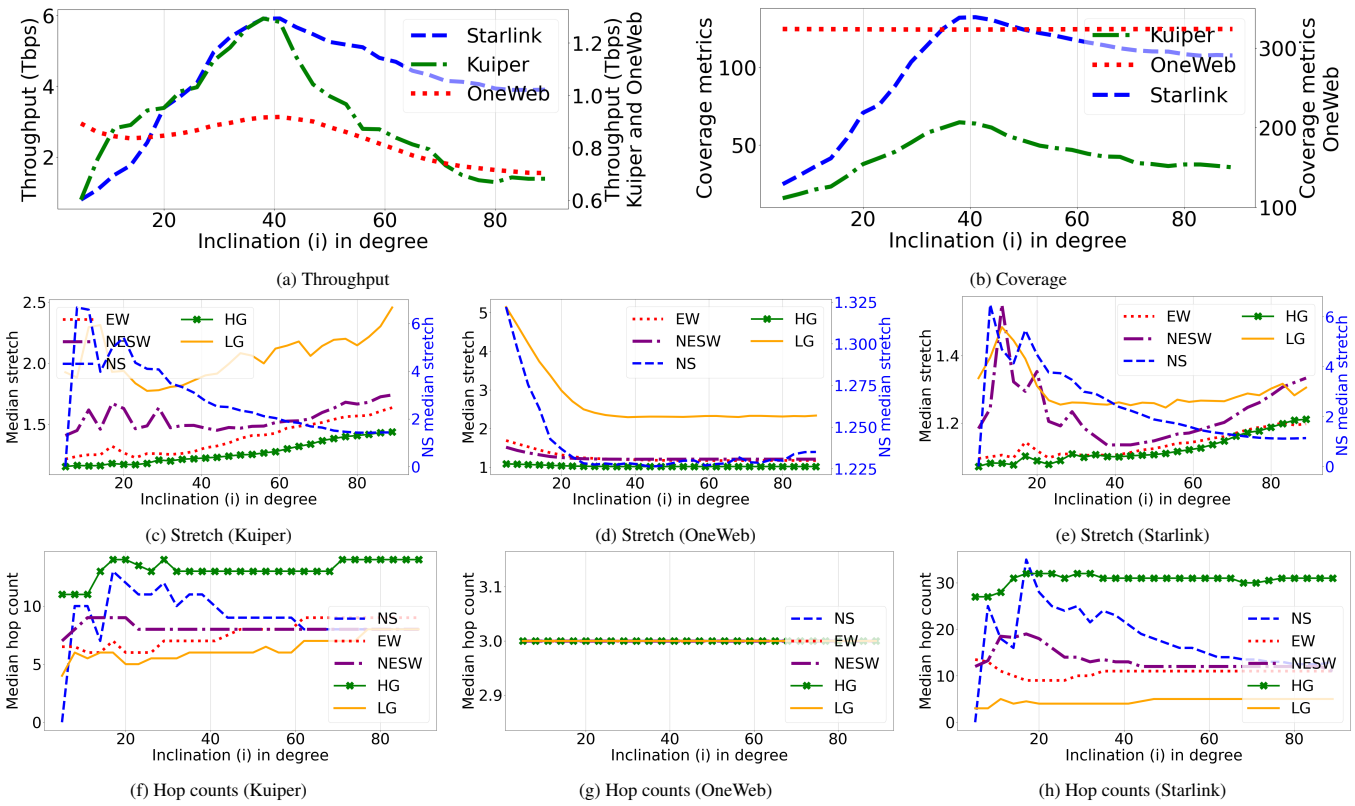


Fig. 21: (a) Throughput, (b) coverage, (c)-(e) stretch, and (f)-(h) hop counts measurement across most populous cities while varying inclination angle (i).

worldwide. Therefore, in regions with a high air traffic density, each satellite serves the demand of 100+ flights, and the GSL bandwidth is divided among these; thus, the throughput deteriorates. Only Starlink's first shell of 1,584 satellites could not cope with this many flights. Nevertheless, the overall behaviors of the throughput measurement are the same, as most flight densities are of the domestic flights in the USA, Europe, and the eastern part of China within the airspace of 20° N to 55° N latitude lines. All international air traffic follows a few specific flight corridors [49] to cross the Atlantic and Pacific oceans within latitude lines 70° N to 5° N and 60° N to 30° S, respectively, where the flight density is significantly sparser than domestic flight. Thus, the throughput is relatively low towards lower and polar inclinations.

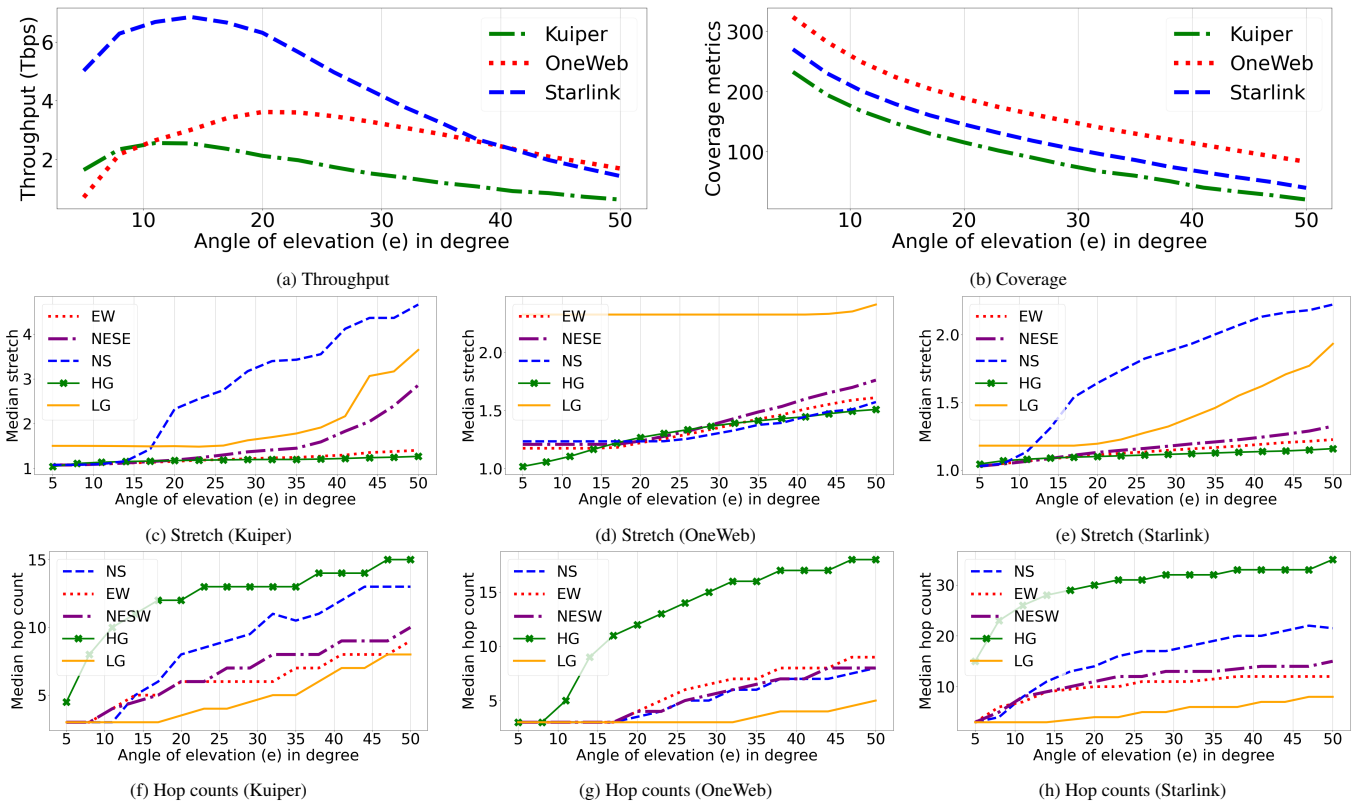
Apart from throughput, characteristics of other performance metrics are also consistent across all the orbital parameters. Again, for brevity, we only show the coverage and NS routes median stretch measurement with i and e in Fig. 24(c)-(d) and Fig. 24(e)-(f), respectively.

Outcome:- The characteristics of various performance metrics with different constellation parameters remain consistent with different traffic demands. Therefore, the proposed search strategy in (§5) is also generalizable and applicable to different target markets.

A.3 Visualization with `LEOCraft`

In this section, we provide an overview of the visualization capabilities of `LEOCraft`. Ideally, `LEOCraft` generates 2D and 3D interactive views to inspect LEO constellations. However, we use some snapshots of interactive views to demonstrate various features and aspects of LEO constellation networks.

Multi-shell LEO constellation: `LEOCraft` can generate a multi-shell LEO network topology. Fig. 25 shows some snapshots of the Starlink [10] and Telesat [12] constellations, whereas Kuiper [1] constellation is already depicted in Fig. 1(b). In these figures, the color distinguishes different shells in the constellation, where the dots represent satellites, and the lines represent the ISLs. Notice that the characteristic of having one shell at a relatively higher inclination ($\geq 60°$) is consistent for most of these proposed designs. For example, in Fig. 25, Starlink Gen1's blue shell and Telesat red shell have inclinations of 70° and 98.98° respectively. OneWeb's and Starlink Gen2 also have shells at inclinations of 87.9°, and 96.9° respectively, as shown in Table 1. Our simulation results in §4 discuss the effects of inclination angle to demonstrate the trade-off between throughput and global coverage, i.e., higher inclination gives more Earth surface coverage at higher latitudes, whereas lower inclination provides better throughput to the well-populated areas with limited coverage at higher (polar) latitudes. We speculate that the LEO constel-

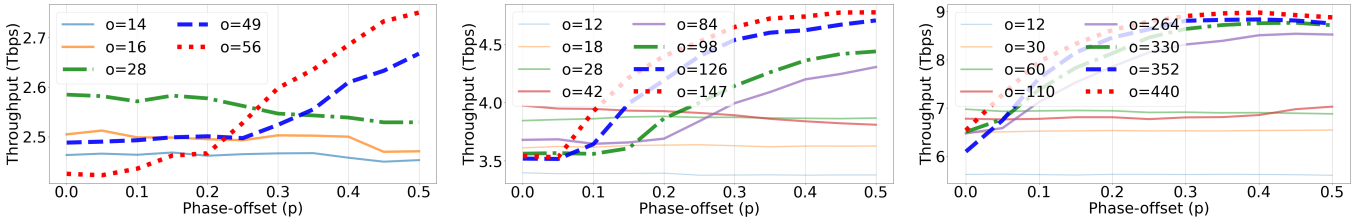


(a) Throughput (b) Coverage

(c) Stretch (Kuiper) (d) Stretch (OneWeb) (e) Stretch (Starlink)

(f) Hop counts (Kuiper) (g) Hop counts (OneWeb) (h) Hop counts (Starlink)

Fig. 22: (a) Throughput, (b) coverage, (c)-(e) stretch, and (f)-(h) hop counts measurement across most populous cities with varying elevation angle (e).



(a) Kuiper (b) OneWeb (c) Starlink

Fig. 23: Changes in the number of orbits (o) and phase offset (p) for all shells with GSes across the most populous cities show similar throughput characteristics.

lation operator tries to address the issue by deploying multiple shells for targeting different regions of interest. They deploy a shell at a higher inclination ($\geq 60°$) to address the coverage at higher latitudes, whereas multiple shells are deployed at lower inclinations to address the demand at densely populated areas.

Route evolution over time: LEO satellites orbit the Earth at a very high velocity of $\sim$ 27,000 kmph [45]. Therefore, the route between two endpoints (i.e., user terminal or gateway ground station) on the Earth's surface continuously evolves over time. `LEOCraft` can be used to visualize such evolution of routes between any pair of GSes over a granularity of nanoseconds. In Fig. 26, we show the snapshots of the evolving shortest paths between Bengaluru, India, and Tokyo, Japan, at different time instances. In this particular example scenario, we can observe that the shortest route between the end-points experiences a route length variation of above 1000 km over just 100 seconds. This leads to abrupt RTT changes, which are also visible for other routes, as discussed previously in Fig. 18(a)-(b).

Coverage and route change due to altitude variation: `LEOCraft` allows inspection of the coverage area of a constellation at individual satellite levels with red-shaded circles beneath it. In Fig. 27, we illustrate the same with the shortest route between Bengaluru, India, and Tokyo, Japan, over the ISL of Starlink's first shell. In this illustration, we only focus on the coverage of ingress and egress satellites. As we increase the altitude of the shell from the default height (i.e., 550 km) to 1,000 km, the coverage area of the satellite (highlighted for ingress and egress points only) increases. Thus, the two endpoints choose satellites closer to each other to form the shortest route, reducing the end-to-end hop length from

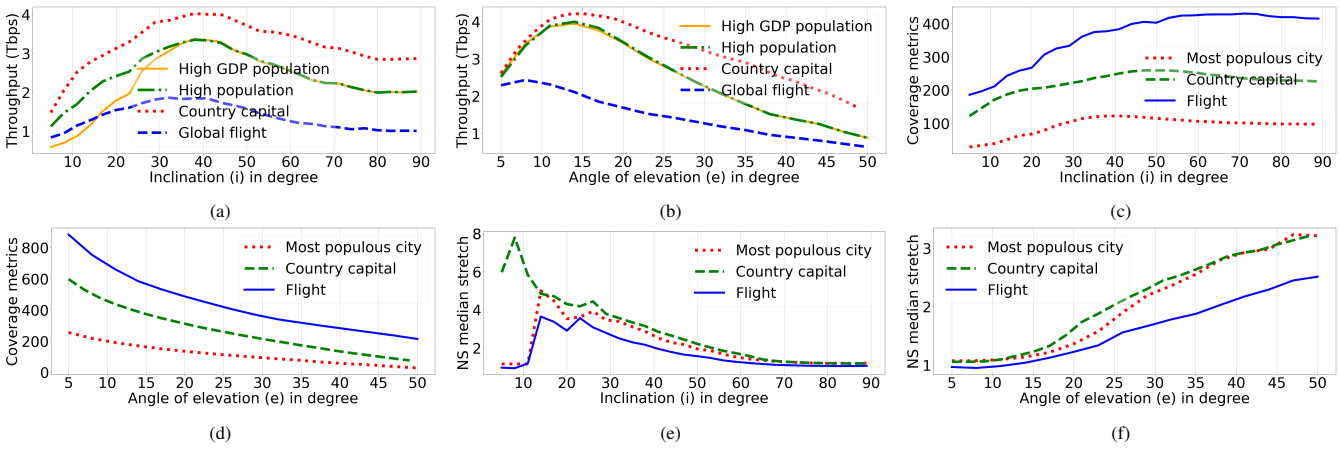
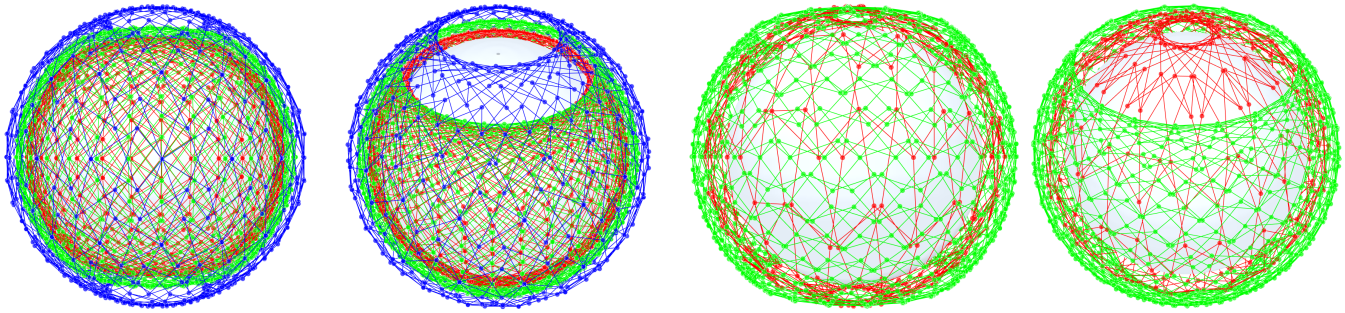


Fig. 24: (a)-(b) Throughput, (c)-(d) coverage, and (e)-(f) stretch measurements with different TMs while varying inclination and elevation angles show similar characteristics.



(a) Equatorial view (b) Polar view (c) Equatorial view (d) Polar view

Fig. 25: Visualizing multi-shell LEO network topology proposed by (a)-(b) Starlink Gen1 [4] and (c)-(d) Telesat [12].

6 satellite hops to only 2 hops. Notice that in Fig. 27(b), the coverage area of the satellite at the higher latitude appears to be larger than at the lower latitude due to the 2D Mercator projection [36].

Ground track of satellites: `LEOCraft` also helps in visualizing the ground track [48] of the satellites. Fig. 28 shows the nadir, i.e., 90° projection of the satellites on the Earth's surface with the inclination of 50° and 80°. The tracking for 120 minutes of both satellites starts at the same point. The figure shows how these two satellites deviate from the starting location after one orbit due to the rotation of Earth. From such visualizations, we can also observe how the coverage regions vary with the inclinations.

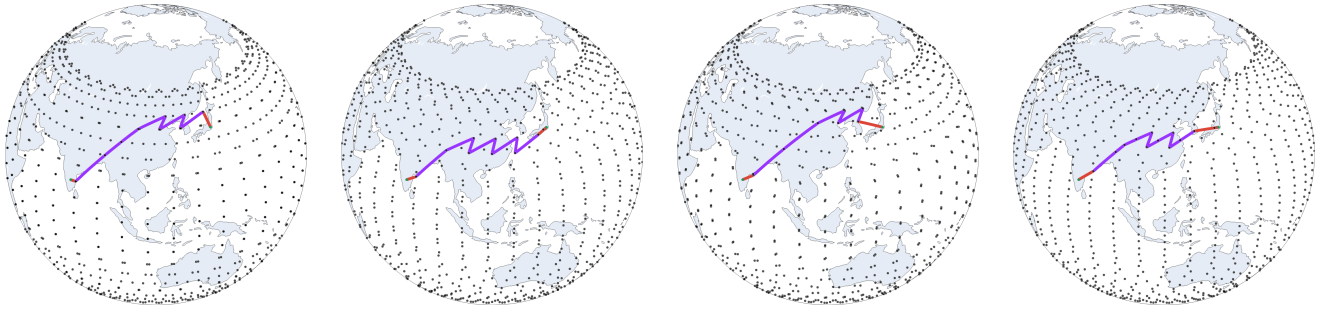
ISL utilization: `LEOCraft` can also be used to visualize the ISL utilization of individual constellations. For example, Fig. 29(d)-(f) shows the link utilization of Starlink's first shell, when the 100 populous cities communicate over the 20 shortest routes. We can extract the link selections and respective link usages in `LEOCraft` to render an interactive view of link utilization using color codes, i.e. *Unused*, *up to* 20%, *up to* 60%, *up to* 80%, *above* 80%.

From Fig. 29(d)-(f) we can observe that many ISLs above the oceans are *unused*, especially over the Pacific Ocean in Fig. 29(f). In general, ISL utilization is *up to* 20% since GSLs are the main bottleneck in these LEO networks. However, ISL utilization in the northern Pacific Ocean is pretty high (*up to* and *above* 80%.) as these ISLs are carrying the traffic between the US and the two most populated areas, i.e., India and the Chain in Fig. 29(f).

We next repeat the same study with 233 GSes located sparsely across the globe in the capital of countries, the results are shown in Fig. 29(g)-(i). From these figures, we can observe that the ISL utilization is increased, whereas the number of *unused* ISLs decreases significantly. A few ISLs are under pretty high utilization above the oceans (such as the center of the Atlantic, or the northwestern area of the Pacific) as they carry the traffic between cross-continent population centers. Such visualizations provide valuable insights about the links or regions that experience network bottlenecks.

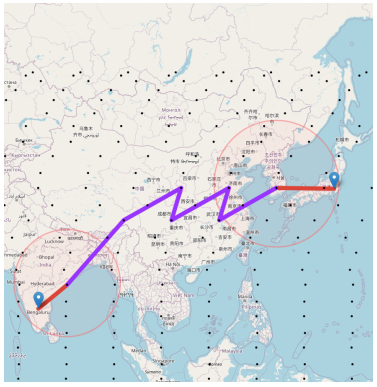
A.4 Additional comments

LEO Constellation design types: Previous works have classified satellite constellation designs into two types based on their geometry [89]. The first one is *Walker Star*, where all satellites in one hemisphere move towards the north pole,

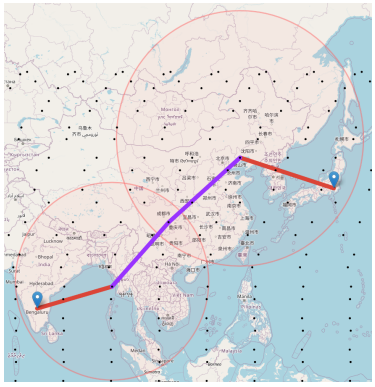


(a) At epoch (b) After 47 seconds (c) After 60 seconds (d) After 100 seconds

Fig. 26: The shortest route from Bengaluru, India, to Tokyo, Japan, changes over time due to satellite movement, leading to the RTT fluctuations.



(a) $h = 550$ km



(b) $h = 1{,}000$ km

Fig. 27: The shortest route from Tokyo to Bengaluru is reduced to two satellite hops as increasing the altitude of the shell expands the coverage area of satellites, shown with ingress and egress satellite.

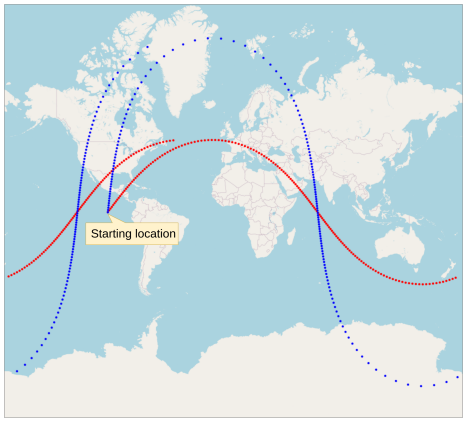


Fig. 28: The ground track over 120 minutes of two satellites orbiting at inclinations of 50° and 80°.

whereas in another hemisphere, these satellites move towards the south pole. Hence, in the polar region, the movement of the satellites creates a star shape while viewing from the top [71, 77]. Satellite constellations are also designed in *Walker Delta* structure, where orbits of less than 90° (likely) inclination are equispaced across the entire Earth's equatorial plane; therefore, northwards and southwards movement of satellites overlap across the globe. Such a design with three orbits creates a triangular shape like the Greek alphabet Δ at the polar region while viewing from the top [39, 87]. Therefore, the *Walker star* configuration provides dense coverage in the polar region; on the other hand, the *Walker delta* configuration creates a coverage gap (if inclined below 90°). Since the main population centers are in lower latitudes, most of the LEO constellations proposed for global Internet broadband fall under the category of *Walker delta* types. Therefore, this work mainly focuses on the *Walker delta* types of constellation design. However, extending `LEOCraft` to study the *Walker star* design is possible with minimal effort due to its modular design.

Impact of interference: In `LEOCraft`, we do not account for the impact of interference since satellite constellations deploy a multitude of interference mitigation techniques to maximize the throughput offered per satellite. Well-known schemes include dividing the terrestrial coverage area into tessellation shapes or spots and then reusing frequencies by assigning adjacent spots different non-overlapping frequency bands. Starlink and Kuiper have extensive plans [1,26] beyond such simple mitigation, which include (a) many steerable, shapeable beams with varying bandwidth, (b) multiple antennas per satellite, each with multiple beams, (c) beam splitting and merging to address interference as well as spatially varying demands, etc.

A.5 Simulation using `LEOCraft`'s APIs

We provide a concise overview of evaluating the LEO constellation using `LEOCraft`'s APIs, along with some code snippets. Additionally, we illustrate the framework's extensibility. For readers interested in further details, `LEOCraft`'s repository [35] includes comprehensive documentation.

Building a LEO constellation: To build a LEO constellation, first create a path loss model for the GSLs with `FSPL` instance as shown in Lst. 1, which uses the Ka band specification available in Starlink's FCC filing [50]. Then, using `LEOConstellation` API, build a LEO constellation in-

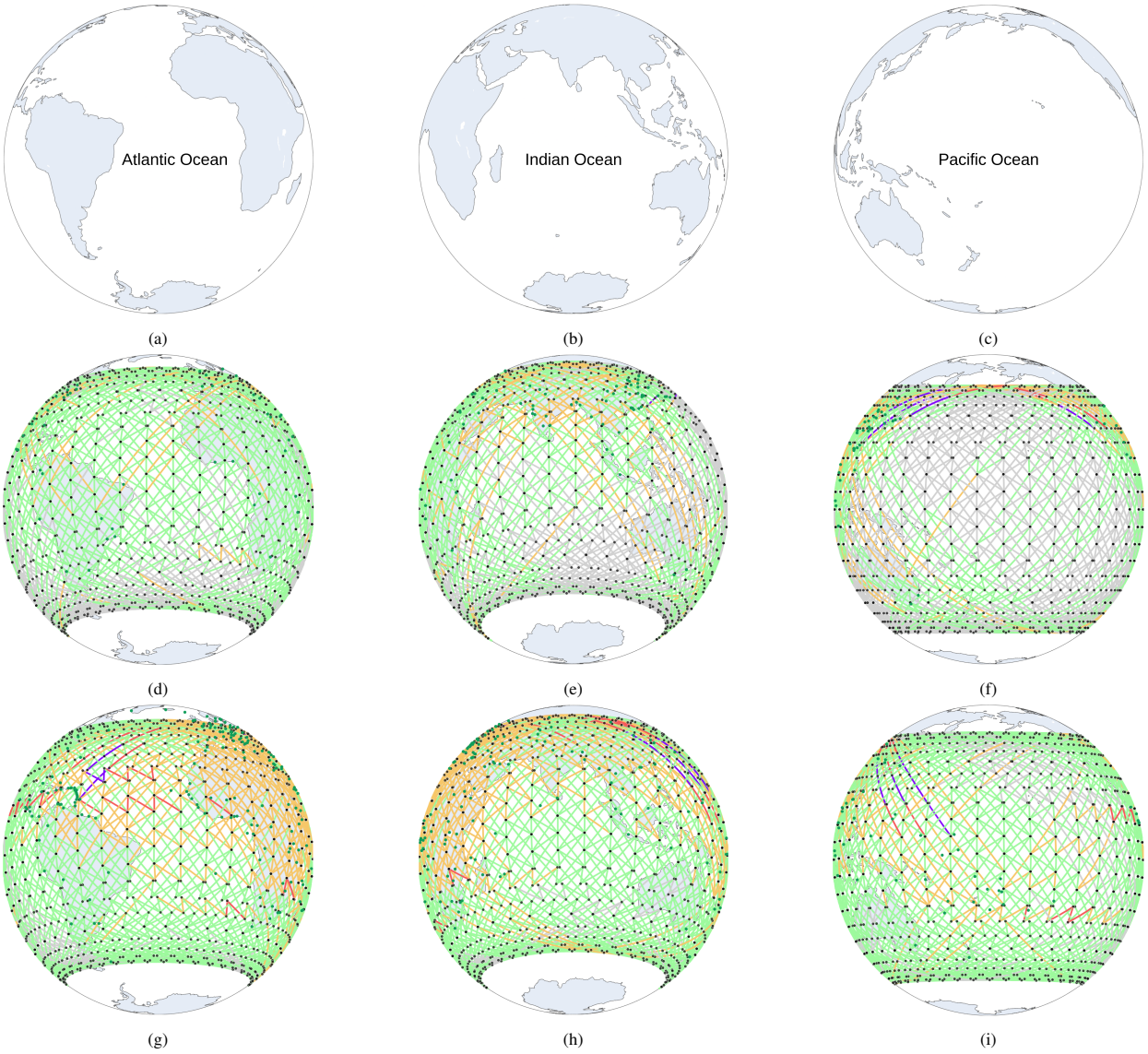


Fig. 29: ISLs utilization above the (a) Atlantic, (b) India, and (c) Pacific Ocean show that most of the ISLs are Unused for (d)-(f) high population TM across 100 cities, whereas the utilization (up to 20%, 60%, 80%, and above 80%) has increased for (g)-(i) the country capital TM across the capital of 233 countries.

```python
from LEOCraft.attenuation.fspl import FSPL

loss_model = FSPL(
    28.5*1000000000,    # Frequency in Hz
    98.4,               # Tx power dBm
    0.5*1000000000,     # Bandwidth Hz
    13.6                # G/T ratio
)
loss_model.set_Tx_antenna_gain(gain_dB=34.5)
```

Listing 1: Building loss model

stance and add all the LEO network component instances like `GroundStation` (builds GSes across cities), `PlusGridShell` (builds all LEO satellites and +Grid ISLs) as shown in Lst. 2. To simulate a multi-shell constellation, repeat the `add_shells()` method with design parameters of the other shells while increasing the `id` for each one. Followed by this, add the instance of the path loss model created in Lst. 1, and set the epoch before building and computing the routes across all the GS pairs.

Evaluating a LEO constellation: Once the routes computation is complete, the instance of `LEOConstellation` can be used to evaluate the throughput, stretch, and coverage using the performance evaluation APIs of `LEOCraft` as shown in Lst. 3. Notice that, unlike stretch and coverage, the throughput instance needs the TMs across the GSes.

Evaluating a cohort of LEO design: In addition to the above approaches, `LEOCraft` also provides a high-level API – `LEOConstellationSimulator` to evaluate a batch of LEO constellation designs. As shown in Lst. 4,

```
from LEOCraft.constellations.LEO_constellation import \
    LEOConstellation
from LEOCraft.dataset import GroundStationAtCities
from LEOCraft.satellite_topology.plus_grid_shell import \
    PlusGridShell
from LEOCraft.user_terminals.ground_station import \
    GroundStation

leo_con = LEOConstellation('Starlink')
leo_con.add_ground_stations(
    GroundStation(
        GroundStationAtCities.TOP_100
    )
)
leo_con.add_shells(
    PlusGridShell(
        id=0,
        orbits=72,
        sat_per_orbit=22,
        altitude_m=550000.0,
        inclination_degree=53.0,
        angle_of_elevation_degree=25.0,
        phase_offset=50.0,
    )
)
leo_con.set_time(second=3)
leo_con.set_loss_model(loss_model)
leo_con.build()
leo_con.create_network_graph()
leo_con.generate_routes(k=20)
```

Listing 2: Building a LEO constellation

```
from LEOCraft.dataset import InternetTrafficAcrossCities
from LEOCraft.performance.basic.coverage import Coverage
from LEOCraft.performance.basic.stretch import Stretch
from LEOCraft.performance.basic.throughput import \
    Throughput

th = Throughput(
    leo_con,
    InternetTrafficAcrossCities.ONLY_POP_100
)
th.build()
th.compute()

sth = Stretch(leo_con)
sth.build()
sth.compute()

cov = Coverage(leo_con)
cov.build()
cov.compute()
```

Listing 3: Evaluating throughput, latency/stretch, and coverage

`LEOConstellationSimulator` serves as the simulation execution framework in `LEOCraft`. It queues all instances of `LEOConstellation` and uses all the available CPUs on the system to build the constellation and compute the routes, throughput, stretch, and coverage. Finally, on evaluation completion, `LEOConstellationSimulator` writes back the results of each `LEOConstellation` in a file for further analysis.

Modularity: `LEOCraft` framework's design offers superior

```
from LEOCraft.simulator.LEO_constellation_simulator import\
    LEOConstellationSimulator

leo_1= LEOConstellation('LEOCON_1')
leo_1.add_ground_stations(
    GroundStation(
        GroundStationAtCities.TOP_100
    )
)
leo_1.set_time(second=3)
leo_1.set_loss_model(get_loss_model())
leo_1.add_shells(
        PlusGridShell(
            id=0,
            orbits=72,
            sat_per_orbit=22,
            altitude_m=1000.0*550.0,
            inclination_degree=53.0,
            angle_of_elevation_degree=25.0,
            phase_offset=50,
    )
)
# ... repeat for leo_2, leo_3, ..., leo_n
simulator = LEOConstellationSimulator(
    InternetTrafficAcrossCities.POP_GDP_100,
    'output.csv'
)
simulator.add_constellation(leo_1)
# Repeat above line for leo_2, ..., leo_n
simulator.simulate_in_parallel(max_workers=3)
```

Listing 4: Evaluating a cohort of LEO constellation

extensibility. For instance, one can inherit `PlusGridShell` and override just one method – `build_ISLs()`, to create a `XGridShell` that evaluates the performance of the LEO constellation with a $\times$Grid ISL topology [72]. Similarly, `Throughput`, `Stretch`, `Coverage` can be inherited to overwrite the `compute()` method to implement other approaches for the computation of these metrics. Hence, `LEOCraft` enables further exploration of LEO design and optimization strategies, addressing a wide range of objectives such as routing policies, bandwidth allocations, traffic patterns, ISL topologies, the number of antennas and their capabilities, weather conditions, and much more.

B Artifact Appendix

Abstract

This artifact provides detailed instructions for setting up `LEOCraft` simulation environments on Linux and macOS platforms. The primary focus is on verifying the claims made in the paper and reproducing the figures presented throughout. Additionally, it offers guidance on extending the `LEOCraft` project to experiment with traffic metrics, topology, routing, and connectivity policies.

Scope

This artifact includes straightforward steps to validate four major claims of the paper – (i) execute a flow-level LEO network simulation, (ii) optimize LEO constellation design using search space heuristic, (iii) evaluate and simulate designs for multi-shell LEO constellation, and (iv) interactive visualization the LEO constellation network. In addition to that, the `GitHub repository` hosts the `LEOCraft`'s source code under the MIT License.

Contents

The `GitHub repository` provides the complete source code for `LEOCraft`, accompanied by thorough documentation. The `Artifact Evaluation` page documents the steps to execute all experiments supporting the major claims of this paper. Additional available contents are outlined below:

Examples: The repository contains a handful of `example Python scripts` illustrating how `LEOCraft` can be programmed for different LEO network simulations and visualization.

Evaluation of constellation with all TMs: The Fig. 20– 24 presents a summarized overview of the performance measurements for various LEO constellations against different TMs. The repository contains all the detailed performance measurements for the proposed constellation, as highlighted in Table 1, against all the TMs discussed in §2.3 for `GSes` and `flights`.

Interactive visualization: The repository also contains an interactive versions of LEO constellation design (Fig. 1(b), 25), satellite coverage (Fig. 1(a), 4(b)-(c)), GSes location (Fig. 7(a)), routes (Fig. 2, 8, 10, 13, 26, 27), ISL topology and satellite trajectory changes (Fig. 7(b)-(d), 11), ground track of satellites (Fig. 28), and ISL utilization (Fig. 29) inside `Visulization Examples`.

Hosting

The source code of the `LEOCraft` under MIT License is available at – `github.com/suvambasak/LEOCraft.git`. The documentation of `LEOCraft` is available at – `suvambasak.github.io/LEOCraft/`. The source code and the documentation might be updated according to the future release with new versions of `LEOCraft`. Additionally, `Artifact Evaluation` page provides the straightforward steps for reproducing the results of this paper.

Requirements

The `LEOCraft` uses the Gurobi Optimizer to solve the linear program to maximize throughput. Consequently, a valid Gurobi license is required to use `gurobipy` for solving linear programs. Additionally, we recommend using a multi-core system with a minimum of 8 gigabytes of memory; however, having more computational resources helps `LEOCraft` to execute faster.

Emulating Space Computing Networks with RHONE

Liying Wang[ℙ*], Qing Li[𝔹*], Yuhan Zhou[ℙ], Zhaofeng Luo[ℙ], Donghao Zhang[𝔹], Shangguang Wang[𝔹]
Xuanzhe Liu[ℙ𝕂], Chenren Xu[ℙ𝕂✉†]
[ℙ]*Peking University* [𝔹]*Beijing University of Posts and Telecommunications*
[𝕂]*Key Laboratory of High Confidence Software Technologies, Ministry of Education (PKU)*

Abstract – The rapid advancement in satellite technology with the adoption of commercial off-the-shelf (COTS) devices and satellite constellation networking has given rise to Space Computing Networks (SCNs). While SCN research is typically conducted on experimental platforms due to high operational costs, the unique challenges of SCNs – such as the harsh space environment (*e.g.,* power and thermal constraints) and dynamic constellation networks – require special consideration. Existing platforms cannot fully replicate the SCN operating environment with high scalability. This paper introduces RHONE, an emulator that bridges these gaps by achieving both satellite- and constellation-level fidelity (the accurate replication of satellite and constellation states, including power, thermal, and network conditions, as well as application performance characteristics) while ensuring usability. RHONE adopts a two-phase emulation approach: *i)* an offline phase builds power, thermal, orbit, network, and computation models using real satellite telemetry data and hardware-in-the-loop chip mirroring, and *ii)* an online phase executes container-based emulation integrated with these models. Key components, the *satellite COTS aligner* and the *satellite network aligner*, dynamically align the containers with real satellite conditions. Evaluation shows RHONE 's scalability to 700 satellites on a single node, with power and computation model errors under 5% and thermal model errors within 1.3 – 2.5°C. Two case studies – satellite network energy drain attack and real-time earth observation application – demonstrate RHONE's capability to emulate satellite- and constellation-level dynamics.

1 Introduction

Low-earth orbit (LEO) satellites are evolving from large, singular, and expensive nodes focused solely on communication into networked computing systems in space, which we term *Space Computing Networks (SCNs)*. Behind this are two technological trends – *i) Adoption of COTS devices.* Conventional low-performance computing components (*e.g.,* 50 MHz CPU with 64 MB RAM [1]) specialized for satellites is being replaced by commercial off-the-shelf (COTS) hardware (*e.g.,* Raspberry Pi [2], FPGA [3], GPU [4]), which has better performance and lower costs. *ii) Satellite constellation networking.* Satellites are no longer limited to low-data-rate communication, but high-bandwidth constellations providing global coverage [5, 6, 7, 8, 9, 10]. Starlink [5], with over 6,000 satellites in low Earth orbit (LEO), provides global connectivity with approximately 100 Mbps downlink and 10 Mbps uplink speeds [7]. Planet Dove constellation, comprising 150 satellites, specializes in Earth observation tasks [11].

SCNs have revolutionized two categories of satellite applications: *i) Earth observation (EO).* SCNs employ COTS devices for in-orbit image processing, such as compression, filtering, and detection [12, 13, 14, 15, 16], reducing the volume of data transmitted. Satellite constellation networking eliminates the need to wait for satellites to pass over ground stations for data transmission. Together, these enable *near-real-time* EO in both industry [11, 17, 18] and academia [19, 15, 16] with SOTA capture-to-insight latency of 6 minutes [15], while conventional approaches take tens of hours to months, as reported by [15, 18]. Academia is actively exploring EO with higher efficiency and lower latency [19, 20] jointly involving onboard processing and data transmission for captured images. *ii) Satellite-backboned global networking.* Satellites are providing global network services, where on-satellite COTS devices enable the potential of satellites as in-orbit computing nodes to perform network functions. For example, satellites can serve as meet-up servers processing and routing video streams for cross-continent video conferencing [21]. Satellites can also integrate cellular core functionalities [22, 23, 24, 25, 26], enabling the processing of heavy user traffic (*e.g.,* User Plane Function (UPF) in the 5G core) using COTS chips. Satellite network attacks and countermeasures also raise concerns [27, 28], as satellites are more susceptible to malicious threats (*e.g.,* DDoS energy drain attacks) due to their harsh operating environments and predictable orbital trajectories.

However, SCN applications are hard to design, build and evaluate. We identify two unique characteristics of SCN that distinguish it from ground-based computing and networking: *i)* satellite COTS chips, which are not originally designed for space environment, face energy and thermal constraints. Overlooking these results in low application performance or even serious harm to the satellite such as overheat or power

*Equal contribution.
†✉: chenren@pku.edu.cn.

failure [2, 29, 30, 31]; *ii)* the dynamic nature of constellation networks (*e.g.,* frequent topology changes [32, 33, 22] and node failures [2, 31, 34, 35]) leads to poor network stability and performance, and should be properly addressed.

Therefore, SCN applications must be designed, built and evaluated with careful consideration of the space environment to ensure proper operation and high performance. However, the high costs of operating or accessing a real constellation (*e.g.,* $150,000 to launch a small satellite [36]) and increasing constellation sizes make it impractical to use real satellites for SCN research. Consequently, SCN research relies on practical experimental platforms that accurately replicate the space environment, *i.e.,* achieving *i) satellite-level fidelity*, involving heterogeneous COTS devices onboard and environmental context (*i.e.,* energy and thermal) they face; *ii) constellation-level fidelity*, involving complexity and dynamics of constellation networks, in terms of both topology changes and node failures; *iii) high usability*, scaling to encompass entire constellations while being cost-effective, flexible, and user-friendly. However, existing tools fall short of meeting all these requirements (§2.3): *i) Physical setups* [5, 37] offer high fidelity but are difficult to access and lack scalability; *ii) Simulators* [38, 39, 40, 41, 42, 13] use analytical modeling to simulate the space environment but lack system/network stacks and support for real system functionalities or interactive network traffic [43]; *iii) Emulators* [44, 45, 43] incorporate constellation network models with virtualized environments (*e.g.,* containers or virtual machines). However, they overlook heterogeneous satellite COTS computing devices facing space environment, especially energy and thermal limitations, and therefore cannot mimic COTS chips' performance, as our measurement study with real in-space satellites shows.

In this paper, we present RHONE[1], an SCN emulator that meets all the requirements above to support SCN research. The challenges in building RHONE stem from a conflict: As the computation performance of onboard chips is influenced by various factors like operating environment, hardware capacity, and application workload, it requires large system overhead (*e.g.,* CPU and memory) or high cost (*e.g.,* hardware device expenses) to achieve accurate emulation for individual satellites. However, while scaling to a constellation level, the total system overhead or cost for emulating SCN constellations exceeds the requirements for scalability and affordability. To address this, RHONE adopts a *two-phased* emulation process – it builds models based on real data collected from in-orbit satellites and on-board COTS devices, and then scales the models to constellation level via a light-weight container-based architecture with the models installed. *i) Offline model building phase*: we analyze telemetry data from in-space satellites and onboard COTS chips using *(a)* an open-source dataset [2] and *(b)* processed flight data which we collected from the Tiansuan constellation[2]. We have released the processed version of the latter on GitHub[3], containing over 800000 timestamped entries spanning December 23, 2021 to July 14, 2022. We build power, thermal, orbit, and network models based on such data to replicate the space environment, and the computation model of COTS chips using *hardware-in-the-loop chip mirroring*; *ii) Online emulation phase*: we use container-based emulation integrating the models built in the first phase to mirror space environment. We design the *satellite COTS aligner* to mirror COTS computing devices through profiling and dynamic resource tuning to ensure satellite-level fidelity. We also integrate the *satellite network aligner* to manage the container network to mirror constellation-level network dynamics. We also provide a set of easy-to-use monitor and command APIs for users to run unmodified applications.

We evaluate the scalability and fidelity of RHONE. Our scalability evaluation shows that RHONE scales to 700 satellites on a single physical machine node, matching the size of Starlink shells, which is comparable to SOTA emulator [43]. In our fidelity evaluation, RHONE shows $< 5\%$ error rate in power and computation models, and $1.3 - 2.5°C$ average error in its temperature model, compared to telemetry data we collect from in-orbit satellites as ground truth.

To illustrate how RHONE can advance SCN research, we present two use cases of SCN with RHONE: *i) Satellite network energy drain attack.* We emulate an energy drain attack [27, 28] in RHONE, where a victim satellite is attacked by malicious heavy traffic load, leading to intensive power usage, rapid battery drain, and degraded network performance. RHONE emulates both satellite-level power consumption and constellation-level network characteristics. *ii) Real-time earth observation.* We emulate a real-time EO application, comparing four processing strategies (direct transmission, compression, onboard filtering, and onboard inference) in terms of power usage, data transmission volume and end-to-end latency. RHONE emulates both satellite-level in-orbit edge computing for image processing and constellation-level network for data transmission.

Contributions.

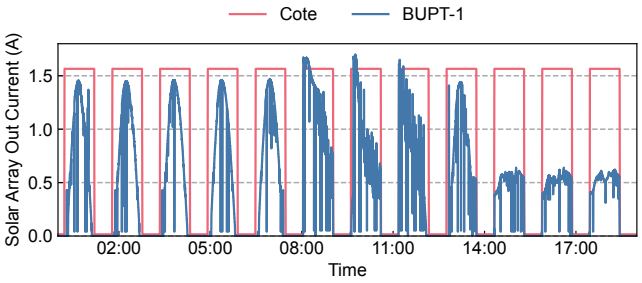
- We present RHONE, a full-fledged emulator dedicated for futuristic SCN research(§3);
- We build models of satellites' operating environment and COTS computing performance by collecting and analyzing real satellite telemetry data (§4), and integrate models with online emulation (§5);
- We conduct experiments to evaluate RHONE's scalability and fidelity (§7), and demonstrate its ability to advance SCN research by two use cases (§8).

This study does not raise any ethical issue.

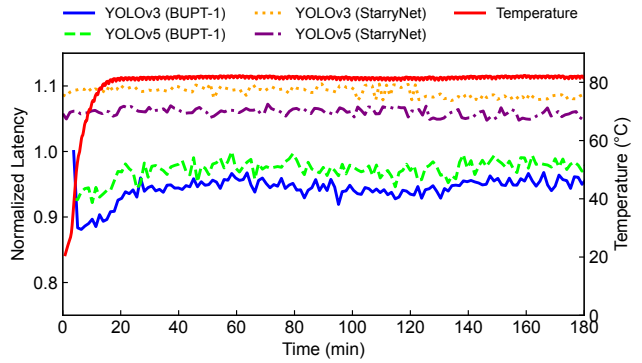
[1] RHONE is named after Vincent van Gogh's painting *Starry Night Over the Rhône*, which features a reflection of the starry sky.

[2] www.tiansuan.org.cn

[3] www.github.com/TiansuanConstellation/ATC25-RHONE-Data



(a) Satellite energy harvesting pattern (cote simulator [13] v.s. real satellite BUPT-1 [2]).



(b) COTS computation capacity comparison.

Figure 1: Inaccurate modeling of SOTA emulators.

2 Background and Motivation

2.1 Research Challenges and Opportunities

While the rise of SCN offers numerous opportunities for the research community, the unique characteristics of satellite platforms and their operating environments present significant challenges for researchers in exploring the architecture and optimization of emerging applications.

Energy and thermal dynamics and constraints. Satellites in space operate under harsh environmental conditions, leading to significant performance degradation compared to those on the ground. Regarding energy, as both energy harvesting capability is influenced by sunlit regions and onboard battery storage is limited, real satellite measurements [2] indicate that the instability in energy supply and limitations on battery discharge depth affect the power available to onboard computing devices, thereby influencing their availability and efficiency. In terms of thermal management, as satellites generally lack active cooling mechanisms to reduce costs, chips can automatically reduce their clock speeds due to thermal throttling. Short-term overheating can reduce the satellite's computing throughput and prolong task execution latency. Long-term exposure to high temperatures can diminish the reliability and lifespan of COTS devices. As a result, there are opportunities for SCN researchers to consider energy and thermal dynamics and constraints when designing and optimizing SCN applications (*e.g.,* allocate more tasks in sunlit regions and fewer in shadows).

Constellation network dynamics. Different from terrestrial networks, constellation networks in space are highly dynamic. These dynamics manifest in two ways: *i) Topology dynamics.* The high velocities of satellites relative to each other and to fixed ground stations lead to highly dynamic connectivity and frequent topology changes. As a result, SCN researchers must develop more robust routing schemes [46], topology design [47, 48] and satellite-ground collaboration strategies [19]; *ii) Node failure.* COTS devices in space are more vulnerable than those on Earth. In addition to cosmic radiation [34], their performance can also be affected by energy and thermal issues, as mentioned above. Recent studies [27, 28] show that malicious network traffic can trigger energy drain attacks on satellites, leading to node failures. Consequently, SCN researchers must address these node failures and their impact on constellation networks.

2.2 SCN Emulator Essential Elements

The complex and dynamic operating environment of SCN, combined with the extremely high costs associated with developing a mega-constellation in space, underscores the urgent need for a comprehensive SCN emulator for the research community to optimize constellation-level applications that integrate computing and networking on real satellites. Based on the preceding discussion, a comprehensive emulator should meet the following requirements to mimic the unique characteristics of SCN:

Satellite-level fidelity. A comprehensive emulator should accurately model the operating environment, computational capability, and application performance of individual satellites. This modeling must encompass both the environment and the devices: *i) Contextual environmental awareness.* The emulator should account for the effects of real satellite operational contexts on applications. For example, energy availability and thermal control should influence computing capacity and efficiency, affecting end-to-end latency in real-time EO applications. *ii) Replications of realistic heterogeneous computing devices.* The emulator should precisely replicate the computation, storage, and communication capacities of satellite COTS hardware, such as Nvidia's TX1 and TX2 (CPU+GPU) [49], Xilinx's Zynq (CPU+FPGA) [3], and Huawei's Atlas 200DK (CPU+NPU) [2].

Consellation-level fidelity. To emulate constellation networking, the emulator must replicate the complexity and dynamics of entire constellations. This includes the impact of topology changes, satellite-ground communication handoffs, ISL bandwidth, and node failures on network performance.

High usability. The emulator must be highly usable in three aspects: *i)* scalability to support simulations of extensive satellite constellations containing thousands of satellites while maintaining accurate modeling of each satellite's performance and inter-satellite networking; *ii)* flexibility to run

Category	Platform	Fidelity						Usability		
		Satellite-level				Constellation-level				
		Energy	Thermal	Computing Capability	Software Stack	Topology	Network Stack	Scalability	Flexibility	Accessibility
Physical Setup	Live Starlink [5]	✓	✓	✓	✓	✓	✓	×	×	×
	Tiansuan [37]	✓	✓	✓	✓	×	×	×	×	✓
Analytical Simulators	STK [38]	partial	×	×	×	✓	×	✓	×	✓
	GMAT [39]	×	×	×	×	✓	×	✓	×	✓
	SNS3 [40]	×	×	×	×	✓	for GEO only	✓	×	✓
	Hypatia [41]	×	×	×	×	✓	×	✓	partial	✓
	StarPerf [42]	×	×	×	×	✓	×	×	×	✓
	Cote [13]	partial	×	×	×	✓	×	✓	×	✓
Virtualized Emulators	Celestial [45]	×	×	×	✓	✓	✓	✓	✓	✓
	StarryNet [43]	partial	×	partial	✓	✓	✓	✓	✓	✓
	RHONE	✓	✓	✓	✓	✓	✓	✓	✓	✓

Table 1: Comparison of different platforms/simulators/emulators.

real applications and network stack provided by users; *iii)* accessibility with easy-to-use interfaces and low cost.

2.3 Limitations of Existing SCN Emulators

Although many simulators/emulators have been proposed in recent years [38, 39, 40, 41, 42, 13, 44, 45, 43], as shown in Tab. 1, they fall short of meeting all these requirements, particularly in accurately modeling the operational context's impact on networking and computing performance.

Physical setups. Commercial satellite constellations [5] and open-sourced experimental satellite platforms [37] with physical setups offer the highest fidelity for both individual satellites and constellations. However, commercial platforms do not provide researchers with experimental interfaces. Open-source platforms allow some experimentation, but lack support for freely running user-defined applications and face scalability limitations tied to constellation size.

Simulators. To address the issues of scalability, flexibility, and accessibility, the research community has turned to the development of analytical simulators [38, 39, 40, 41, 42, 13]. These simulators rely on theoretical modeling without real software and network stacks, limiting their ability to simulate the actual behavior of computing or networking devices. Additionally, most of them also do not allow users to run custom applications or algorithms.

Emulators. This widening gap highlights the growing need for emulators [44, 45, 43] that incorporate virtual environments (*e.g.,* containers or virtual machines), thus enabling real system/network stacks to maintain system fidelity. However, due to the lack of real in-orbit data from COTS devices on satellites, even the current state-of-the-art (SOTA) solutions struggle to accurately replicate the actual environmental context, including energy and thermal influences, as well as the characteristics of onboard computing devices.

Motivating examples: low emulation fidelity. We analyze the power, thermal, and computing capacity dynamics of COTS devices onboard with the open-source dataset from an in-space satellite BUPT-1 [2] running DNN inference tasks on a Raspberry Pi. The dataset includes telemetry data of power, temperature, and onboard tasks. With this ground-truth data, we highlight two motivating examples that illustrate the limitations of existing SOTA emulators in achieving high-fidelity emulation of individual satellites: *i) Inaccurate power model. Cote* [13], the SOTA satellite computing simulator, provides a power model for individual satellites. We compare it with the actual satellite harvested power trace, as shown in Fig. 1a. The power model in *Cote* assumes the harvested power in the sunlit region to be constant. In reality, the harvested power varies with the satellite's position, attitude, placement of solar panels, and the satellite's orientation. An inaccurate power model fails to simulate the effects of power fluctuations on computing or networking devices, such as reduced availability during power shortages, which *Cote* does not accurately represent. *ii) Inaccurate computation capacity. StarryNet* [43], the SOTA satellite networks emulator, uses Docker containers to emulate computing devices on individual satellites. We replicate the onboard DNN inference experiments within a container of *StarryNet* and compare the inference latency with those obtained from Raspberry Pi on BUPT-1. As shown in Fig. 1b, the execution latency of an application in its containers deviates significantly from that of an in-orbit COTS device. The YOLOv3 and YOLOv5 execution latency fluctuations of onboard chips are 330% and 160% larger than emulated, respectively. The application's average latency in the container differs by up to 15% compared to the real in-orbit COTS devices. Moreover, *StarryNet* fails to simulate the increase in inference latency caused by chip throttling due to overheating (initial 25 min in Fig. 1b). These inaccuracies arise because *StarryNet* simulates devices' capabilities by adjusting CPU core counts and frequencies. However, this method does not capture the inherent characteristics of COTS devices, which are often heterogeneous (*e.g.,* Jetson TX2's CPU + GPU). Additionally, COTS devices and containers operate on different computing architectures, leading to different performance.

3 RHONE Overview

We present RHONE, a comprehensive SCN emulator meeting all requirements discussed in §2.2. As shown in Fig. 2, the emulation process of RHONE consists of two phases: *Offline Model Building* (§4) and *Online Emulation* (§5). In

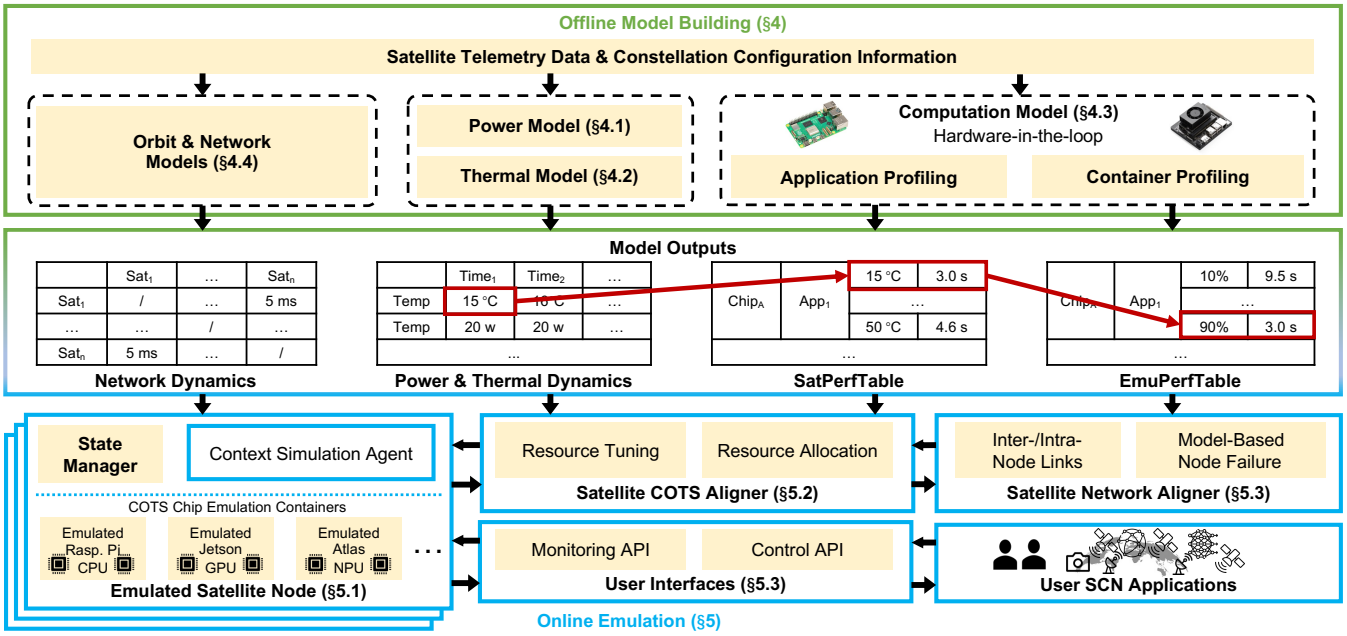


Figure 2: RHONE system overview. The red lines illustrate the query process of the resource allocation scheme.

the model building phase, RHONE initially constructs power, thermal, computation, orbit, and network models offline using telemetry and experimental data collected from real satellites and onboard COTS devices. These models replicate the complex and dynamic SCN environment, effectively mirroring the behavior of onboard chips. In the emulation phase, RHONE deploys these pre-built models into software containers to emulate each satellite node and their interactions. This two-phased design ensures high-fidelity emulation of individual satellites while scaling to the constellation level at a low cost. Users can interact with the system via monitoring and control APIs to run custom applications.

4 Offline Model Building

In this section, we propose and build the power, thermal, computation, orbital, and network models of RHONE to replicate the computation and communication capacities of individual satellites, based on real telemetry and profiling experiments conducted on devices.

4.1 Power Model

As discussed in §2.1, energy dynamics is a critical factor affecting satellite computing performance. We first collect a complete set of power telemetry data from a satellite (Fig. 3), including time series of current and voltage from the solar panels and all onboard electrical components. This dataset is sufficient for building the power models because it contains energy behaviors for all orbital working modes of the satellite (*e.g.,* Earth-pointing, Sun-pointing, etc.) during approximately 2,000 orbital cycles. Our principle is to identify general patterns of energy behavior from this dataset and build our power model. Users with satellite power telemetry can also adopt our method to fine-tune their own models.

4.1.1 Energy Harvester

Most LEO satellites regularly transition between sunlight and eclipse during their orbit, leading to cyclical energy harvesting. Additionally, the power harvested in sunlight varies dynamically, as illustrated in Fig. 1a, due to factors such as the satellite's movement and changes in its working mode. To emulate this energy-harvesting process, we adopt a widely used power model as in [29, 30, 50, 51]:

$$P_h = S \times \eta \times I_d \times L_d \times A \times \cos\theta, \quad (1)$$

where P_h represents the harvested power, such power is determined by the solar constant flux density S, the efficiency of the individual solar cells η, the inherent degradation I_d, the lifetime degradation L_d, the panel area A, and the cosine loss $\cos\theta$. The angle θ is the angle between the solar panel norm and the direction of the incoming sunlight, as shown in Fig. 3. It is primarily influenced by the satellite's orbit, geometry, the installation orientation of the solar panels, and the satellite's pointing direction. Our energy-harvesting emulation method is straightforward but effective, the accuracy of Eqn. 1 is evaluated in §7.2.2.

4.1.2 Energy Consumer

Most satellite subsystems, including the electrical power system, propulsion, guidance, attitude and orbit control, and avionics, consume electrical power. The COTS devices, acting as the computing payload, share a power supply with these systems. To determine the available electrical energy

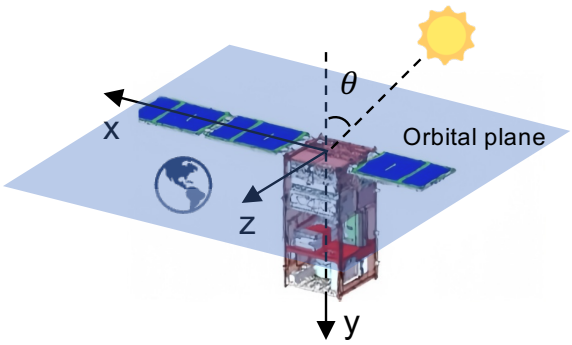


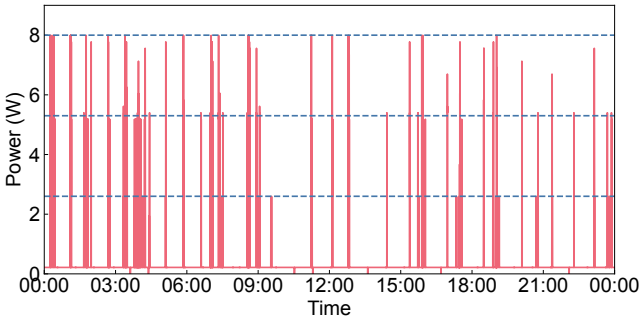
Figure 3: Satellite position and angle θ.

for the computing payload, it is essential to accurately characterize the power consumption of other subsystems.

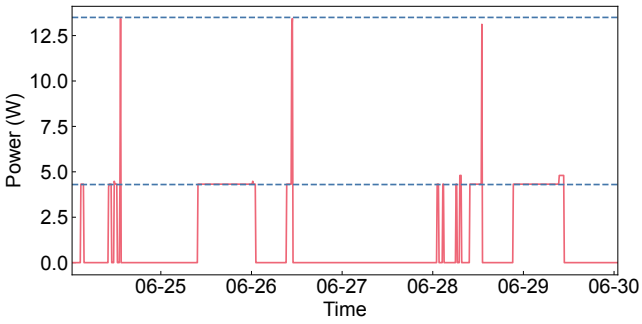
Working and power consumption patterns. In RHONE's power model, we analyze component-level power consumption and operational behavior using real satellite telemetry data, specifically current and voltage readings from various components. While components have different power shares, they follow two working patterns and two power-consumption patterns, which innovate our modeling of onboard power consumption. *i) Working pattern.* Onboard components' working patterns can be classified into two types: *continuous operation* and *intermittent operation*. Components with continuous operation, typically part of the satellite's platform system, support the satellite's normal functioning, such as the magnetic torque system. In contrast, components with intermittent operation are activated only during specific tasks, like the X-band communication component. *ii) Power-consumption pattern.* A component's power consumption can be categorized into *Markov-based* (Fig. 4a) and *function-based* patterns (Fig. 4b). Next, we detail both patterns of power consumption modeling.

Markov-based power consumption pattern. Components with a Markov-based power consumption pattern switch randomly between several power levels, resembling a Markov chain [52]. For example, the magnetic torquer switches between three power levels, as shown in 4a. To model such components, we perform clustering analysis on the power telemetry data to identify the power levels and use their occurrence probabilities to generate each power value over time. This process reveals the underlying Markov chain and its accuracy is evaluated in §7.2.2.

Function-based power consumption pattern. Some components' power consumption patterns can be modeled by specific functions, such as the X-band and computing devices. The X-band component is responsible for high-speed data downlink and receiving control commands from ground stations once the connection is established. As



(a) Markov-based pattern with three levels (magnetic torque).



(b) Function-based pattern with idle and active states (Xband).

Figure 4: Energy consumption patterns.

shown in Fig. 4b, the X-band transitions between two states: idle (approximately 4W) and active (approximately 14W). This behavior follows the classic radio link power model [27, 53, 54], which can be expressed as:

$$P_{\text{tr}}(t) = \begin{cases} P_{\text{tr}}^{\text{idle}} & \text{if } t \in \text{idle state}, \\ P_{\text{tr}}^{\text{active}} + w_{\text{tr}} dr_{\text{tr}} & \text{if } t \in \text{active state}, \\ 0 & \text{otherwise}. \end{cases} \quad (2)$$

where $P_{\text{tr}}^{\text{idle}}$ is idle power, $P_{\text{tr}}^{\text{active}}$ is baseline power in active stage, w_{tr} is a link-specific constant, and dr_{tr} is link data rate.

For COTS computing devices, we leverage the insight that their power consumption is influenced by: *i)* the operational frequency regulated by the device and *ii)* the temperature [55]. The power model can be formulated as [56],

$$P_c = P_s(f) + U \cdot P_d(f) + P_l(T_{\text{hottest}}), \quad (3)$$

where f represents the processor frequency, U is the processor utilization, T_{hottest} represents the temperature of the hottest spot on chip, P_s is the idle power, P_d is the dynamic power, and P_l is the leakage power. We use our telemetry data to tune the parameters of our power model and evaluate the model accuracy in §7.2.2.

4.2 Thermal Model

Unlike proactive cooling methods (*e.g.,* fans, water-cooling) on the ground, many cost-effective small satellites rely on passive cooling methods that depend on thermal conduction

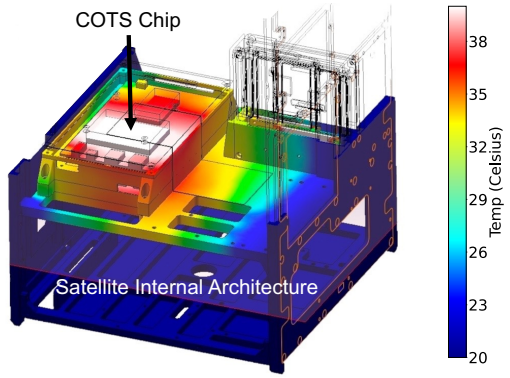
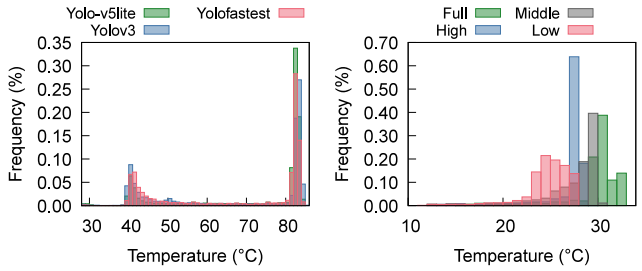


Figure 5: FEA-based thermal model illustration.



(a) Raspberry Pi (b) Atlas 200 DK

Figure 6: Temperature distribution of COTS devices under computing load.

and radiation [57]. The in-orbit temperature of COTS computing devices significantly impacts their computational performance. Therefore, in RHONE, we primarily develop a thermal model for computing devices, as shown in Fig. 5. RHONE's thermal model focuses on the chip temperature, which directly affects computing efficiency and reliability.

FEA-based thermal model. Simulating temperature variations in computing devices ideally requires real satellite data, but varying heat dissipation across different devices makes profiling costly and non-scalable. Ground-based simulations, including thermal vacuum tests, struggle to replicate external heat flux and structural thermal conduction. To overcome this, we use Finite Element Analysis (FEA) in SolidWorks [58] to model the satellite's thermal environment and heat transfer. FEA offers scalable and accurate thermal dynamics modeling, widely used in satellite engineering for designing and evaluating thermal control systems [59, 60].

The computing payload primarily dissipates heat through thermal conduction via the mounting structure. In FEA based thermal conduction simulation, the input includes the satellite's internal temperature and the energy consumption trace from the power model. The output is the chip temperature distribution over time. The thermal conduction equation used in these simulations is:

$$\rho c \frac{\partial T}{\partial t} = \nabla \cdot (k \nabla T) + q''' \tag{4}$$

where ρ is the material density, c is the specific heat capacity, T is the temperature vector of the chip region, t is the time, k is the anisotropic thermal conductivity, q''' is the local volumetric heat generation rate (*i.e.,* the total amount of heat generated per unit volume), primarily due to internal heat sources such as electronic components.

The FEA model uses a piecewise-smooth approximation for the temperature field. It models heat transfer in two main ways: (i) internal conduction within component domains (*e.g.,* chips), where Eqn. 4 applies and the $\nabla \cdot (k \nabla T)$ term describes anisotropic heat conduction; and (ii) heat transfer from chips to the satellite chassis and structure, modeled through physical interfaces (*e.g.,* thermal pads, structural contact). External thermal loads on the satellite (*e.g.,* orbital solar flux, albedo) and system-level thermal context are incorporated via a boundary anchoring strategy, using interface temperatures as empirical boundary conditions. This applies appropriate thermal boundary conditions to the model's exterior surfaces and relevant interfaces, efficiently accounting for the broader thermal environment.

Then, time discretization is carried out using an implicit method [61], ensuring stability and convergence. The temperature update equation is:

$$(M + \alpha \Delta t K) T_{n+1} = M T_n + \alpha \Delta t q''' \tag{5}$$

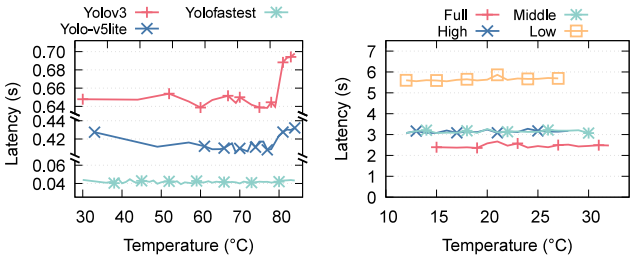
where M is mass matrix which represents thermal inertia of model, α is a coefficient, typically set to 0.5 or another appropriate value, Δt is the time step, K is Stiffness Matrix which describes the heat transfer efficiency, T_n and T_{n+1} are the temperature vectors at the current and next time steps. By integrating these steps, the thermal model accurately depicts the temperature distribution within the satellite.

Note that application workloads interact with the thermal model in two ways: *(i)* Workload drives utilization, determining chip power consumption (Eqn. 3), which serves as thermal model input. *(ii)* Application performance is determined by the output temperature of the thermal model.

4.3 Computation Model

The computation model for a COTS chip uses the current temperature to estimate the performance (execution latency) of user-defined applications. This estimation facilitates the alignment of computing performance from ground-based emulation to in-space satellites.

Challenge: heterogeneity of COTS device performance. To understand satellite COTS devices' performance, particularly under overheating, we conduct in-space experiments that complement the measurement study in [2], using a Raspberry Pi and Ascend Atlas 200 DK to run image recognition workloads. The Raspberry Pi runs YOLOv3 [62], YOLOv5lite [63], and YOLOfastest [64] for 90 minutes



(a) Raspberry Pi (b) Atlas 200 DK

Figure 7: Task latency of COTS computing v.s. Temperature (on Satellite).

Figure 8: Relative error of task execution latency between in-space and on-the-ground COTS devices.

each, while the Atlas runs YOLOv3 under four power settings (Full, High, Medium, Low) for 30, 60, 60, and 90 minutes. Data on temperature distribution (Fig. 6) and model inference latency (Fig. 7) reveal two levels of heterogeneity: *Device-level heterogeneity.* The two chips exhibit different average performance, given their differing architectures. For the same model, YOLOv3, the Raspberry Pi and Atlas show varying latency (Fig. 7a). Moreover, they respond differently to overheat-induced performance degradation. The Raspberry Pi shows increased latency for YOLOv3 and YOLOv5lite above 80°C (Fig. 7a), while the Atlas maintains stable latency (Fig. 7b) across its temperature range (Fig. 6b); *Application-level heterogeneity.* Different applications also react uniquely to overheating. On the Pi, YOLOv3 and YOLOv5lite experience increased latency above 80°C, but YOLOfastest does not (Fig. 7a). Such device- and application-level heterogeneity necessitate tailored modeling for each device and application. To model the computation performance of satellite COTS devices, one intuitive method is to directly set up real COTS devices and execute various applications on them during emulation. However, the challenges of replicating satellite thermal environments on the ground, coupled with inflexibility and high costs, make this impractical. Instead, RHONE adopts a hardware-software co-designed approach: it integrates real COTS devices into the computation model and mirrors them using software containers at scale, maintaining both fidelity and scalability.

Observation: chip overheat can be mimicked on Earth. Our design is based on the key observation that *although the space environment differs from that on the ground, the performance of COTS devices, especially the degradation due to overheating can be mimicked on Earth*. The rationale is straightforward: overheating leads to performance degradation, and at the same temperature, both satellite and ground-based COTS chips should experience similar degradation. Note that we focus on short-duration tasks (*e.g.,* object detection) where single executions induce negligible temperature changes, which are typical for satellite in-orbit computing. We validate the observation by conducting Earth-based experiments using the same COTS chips as those deployed on satellites, with their cooling fans removed to simulate the absence of active cooling in space. We cyclically stress each chip by repeatedly executing a specific task until thermal throttling occurs, then cool it, and repeat, while each task execution records temperature and execution latency. We observed that the performance degradation closely mirrors that of in-space conditions under temperatures equivalent to those encountered in space as shown in Fig. 6. As shown in Fig. 8, the relative error of task execution latency between in-space and on-the-ground devices is relatively small for most workloads, demonstrating that our Earth-based setup accurately reflects satellite performance under overheating. Note that Pi-YoloFastest's high speed and large random error resulted in a large relative error. Atlas YoloV3Full also showed high relative error, though we believe better space-simulating thermal vacuum experiments could improve this.

Hardware-in-the-loop chip mirroring. Such observation inspires us to integrate actual COTS chips without cooling fans into RHONE to gather detailed performance data across various temperatures. Specifically, when a user registers an application to be run on a particular in-orbit chip, RHONE first profiles the application using the aforementioned cyclic stressing method, generating a `satPerfTable` that records the application execution latency on the chip at varying temperatures. Next, to mirror the specific COTS chip using software containers, we perform a similar profiling process within the container. This involves adjusting resource constraints, such as CPU and GPU shares, to collect the container's performance under various resource allocations. This generates an `emuPerfTable` that records the application latency under different container resources allocated. The `satPerfTable` and `emuPerfTable` are then used by the SCA to accurately replicate in-space COTS chip performance accordingly (§5.2). The computation model accuracy is evaluated in §7.2.4.

4.4 Orbit and Network Models

The orbit and network models of RHONE are similar to existing studies [65, 43]. Based on the highly predictable satellite orbital operation, the orbit model calculates each satellite's real-time position at any given time from user-defined

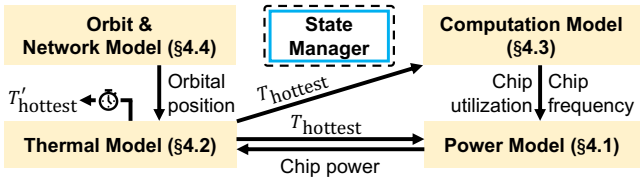


Figure 9: Model interaction in RHONE online emulation.

constellation and orbit configurations. The network model then utilizes the position data to derive network connectivity by calculating the visibility and signal propagation delay of ISLs and GSLs, providing the foundation for replicating the link performance of real satellite constellations. Additionally, RHONE also computes the real-time positions of Earth and the Sun relative to the satellite, which is passed to the power model to estimate energy harvesting current intensity.

5 Online Emulation

During the online emulation phase, RHONE leverages the models built in the offline phase (§4) to manage system resources and replicate the SCN environment.

5.1 Emulated Satellite Nodes

In RHONE, each satellite node is represented by a state manager and a set of Docker containers each representing a chip on the satellite, either representing a COTS computing chip.

State manager. The state manager is responsible for managing the satellite's emulation states, serving as the central point for interactions. A context simulation agent interfaces with pre-established models (§4) that capture the satellite's physical and operational characteristics. It continuously fetches the generated traces including energy harvesting and consuming, temperature condition variation, network connectivity and properties from the orbit and network models, and COTS hardware performance data from the computation model from the data store. RHONE coordinates nested interaction of the models iteratively (Fig. 9). For example, the power model reads current chip temperature T_{hottest}, and outputs the chip power value, which is then used by the thermal model to derive T'_{hottest}, the new chip temperature value of next time step. Additionally, the agent handles incoming commands and monitoring requests through user API, executing commands to alter the satellite's states and providing real-time status updates as needed. By maintaining an accurate representation of the satellite's current state, the state manager relays critical information to the COTS chip emulation containers, ensuring that the emulation remains consistent with the satellite's operational context.

COTS chip emulation containers. These containers are designed for emulating COTS computing hardware typically equipped on modern satellites, such as Raspberry Pi [2], Jetson [66, 14], or NPUs [2]. Each container reflects the performance of its corresponding chip under various conditions by operating with computing resources dynamically adjusted by the satellite COTS aligner (§5.2) based on the current physical state. For instance, an ARM CPU is represented by a CPU container, while a GPU container models hardware like NVIDIA Jetson GPU or Huawei Atlas NPU. This structure, using containers with dynamic computation resources instead of hardware-based emulation, offers flexibility and scalability with low cost. Additionally, the accuracy is ensured through precise modeling of chip performance and careful handling of resource constraints.

5.2 Satellite COTS Aligner

We design the *satellite COTS aligner* (SCA) to dynamically align the performance of emulated COTS hardware under emulated satellite conditions, based on offline modeling of real satellite COTS chips (§4.3), and current temperature given by the thermal model (§4.2). It adjusts resource allocations in real-time to ensure the emulated environment accurately reflects the satellite hardware's behavior.

Resource allocation. Upon invocation, SCA uses the outputs from pre-built models to map the current chip temperature to application execution latency. As illustrated in Fig. 2, SCA first retrieves the current chip temperature, calculated by the thermal model (*e.g.,* 15°C). It then finds the corresponding execution latency in the `satPerfTable`, as determined by the computation model (*e.g.,* 3.0 s). Using this expected latency, SCA queries the `emuPerfTable` to determine the required resource allocation (*e.g.,* 90%) in the container, ensuring accurate performance replication.

Dynamic resource tuning. With the determined resource allocation scheme, SCA adjusts the container's CPU/GPU share to match the satellite's expected performance. This adjustment is based on the assumption that the computational power of satellite COTS hardware is weaker than that of ground servers (*e.g.,* the satellite's CPU is weaker than a server's CPU, and its NPU/GPU is weaker than a server's GPU). Therefore, we can slow down the ground server's CPU and GPU by limiting their resources, bringing their performance closer to that of the satellite's COTS hardware.

For CPU resource tuning , SCA works by adding the process to a certain `cgroup`, and periodically adjusting the `cgroup`'s configured CPU share value, which artificially and accurately controls latency. For GPU adjustment, SCA dynamically manages and limits GPU performance by intercepting CUDA driver API calls and monitoring activity. RHONE initializes a rate limit manager that reads GPU performance thresholds from a configuration file. During each CUDA call, a rate limiter checks GPU usage against these thresholds and applies delays if the usage exceeds the limit.

5.3 Satellite Network Aligner

RHONE utilizes Docker networks to emulate complex networking scenarios, including variable latency, fluctuating bandwidth, and dynamic network topologies. The Satellite Network Aligner (SNA) inherits StarryNet [43] to manage the docker network, aligning its configuration and thus

Satellite Num.	CPU Usage Mean (%)	CPU Usage Std. Var. (%)	Memory Usage Mean (%)	Memory Usage Std. Var. (%)	Setup Time (min)
172	12.1 / 18.3	14.7 / 33.1	8.9 / 10.3	0.1 / 1.5	15.1 / 5.1
348	23.7 / 27.5	31.1 / 38.8	13.5 / 16.7	0.2 / 3.0	19.8 / 7.8
720	37.7 / 42.7	35.6 / 39.5	24.5 / 33.3	0.5 / 8.5	42.2 / 17.1

Table 2: Emulation capability of RHONE compared to StarryNet [43]. X/Y indicates the performance of RHONE/StarryNet.

performance to the orbit and network model (§4.4). Moreover, we introduce two features beyond StarryNet, enabling a more accurate and realistic emulation of SCNs.

Granular satellite node representation. In StarryNet, each satellite node is represented by a single container. This oversimplifies the satellite's structure, beacuse a satellite typically consists of multiple COTS devices communicating with each other. In RHONE, we represent each satellite node with multiple containers (§5.1), each representing a COTS computing device, thus enabling granular network emulation involving internal communications between the multiple SoCs within a satellite. RHONE also allows users to designate a certain container to represent the on-board computer (OBC), which sends/receives data to/from outside the satellite, and communicates with multiple COTS chips.

Model-based node failure. Additionally, StarryNet uses a random approach to emulate node and link failures. In RHONE, the SNA take a more controlled approach by emulating node failures, shutdowns, or other behaviors based on detailed power and thermal models (§4.2). Rather than randomly disrupting inter-satellite communications, RHONE accurately emulates the consequences of power shortages, overheating (*e.g.,* shutdown the COTS devices at low battery level or high temperature), and other operational challenges, resulting in a more realistic and reliable network emulation.

5.4 User Interfaces

RHONE provides two types of runtime APIs for user interaction: *Monitor API*s and *Command API*s.

Monitor APIs allow users to track the real-time status of satellite nodes and networks. This includes monitoring physical states such as temperature and energy levels, the load on COTS computing hardware, and the status of satellite network links (*e.g.,* routing tables and connection states).

Command APIs offer users the capability to dynamically modify the state of the satellite nodes and the constellation. Through these, users can change operational modes, modify the connectivity of inter-satellite links or satellite-to-ground links, and initiate specific tasks on a satellite, such as running an application on its onboard COTS hardware.

6 Implementation

We now highlight the key aspects of RHONE 's implementation, detailing the foundational offline model building phase and the subsequent online emulation phase.

Offline model building. We prototype RHONE using telemetry data from real satellites, sourced from both an open-source dataset [2] and our own measurements. We use some commercial (*e.g.,* SolidWorks) and open-source software (*e.g.,* PyBaMM) to run the offline emulation process to build the power, thermal, and computation model. We use two common COTS hardware for performance tests while building the computation model: a Raspberry Pi 4B and a Huawei Ascend Atlas 200 DK, both without cooling fans. We use a program-controlled thermoelectric cooler [67] to simulate colder environments when needed.

Online emulation. RHONE coordinates all models using Python Requests API. The SCA uses `cgroups` to limit CPU resources, and TGS [68] for GPUs. Network connections between nodes are managed using the docker network architecture in StarryNet [43]. The core components of RHONE are implemented in about 4500 lines of Python/C# code.

7 Emulation Evaluation

7.1 Scalability

7.1.1 Experimental Setup

To evaluate the scalability of RHONE, we perform constellation emulations on an enterprise server with both RHONE and StarryNet [43]. Our server is equipped with two Intel Xeon E5-2670 v3 processors, 64GB of DDR4 RAM, 435 GB of disk storage, meeting the requirements of StarryNet.

7.1.2 Results

Performance. As Tab. 2 shows, RHONE supports elastic scaling for large constellations. We demonstrate this by using RHONE to deploy constellations of 172, 348, and 720 satellites, which corresponds to the S3 shell of Starlink, on a single physical machine. This exhibits scalability comparable to StarryNet's performance on a single node. When a multi-node cluster is enabled, RHONE is expected to scale to constellations consisting of thousands of satellites.

System overhead. RHONE exhibits comparable or even lower system overhead in terms of CPU and memory usage. As Tab. 2 shows, CPU and memory usage scale with constellation size, with average CPU usage increasing from 12.1% for 172 satellites to 37.7% for 720 satellites. Similarly, memory usage rises from 8.9% to 24.5% across the same constellation sizes. Notably, these values are consistently comparable to or lower than those observed with StarryNet. This is because RHONE shifts modeling to the offline model-building phase and uses lightweight agents to interact with pre-defined models, resulting in negligible overhead.

Setup time. RHONE exhibits longer setup times than StarryNet due to the calculation of environmental (*e.g.,* energy

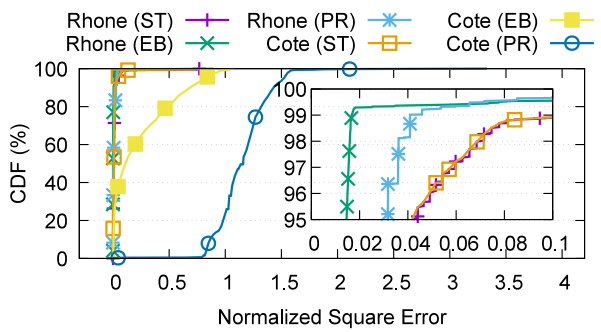
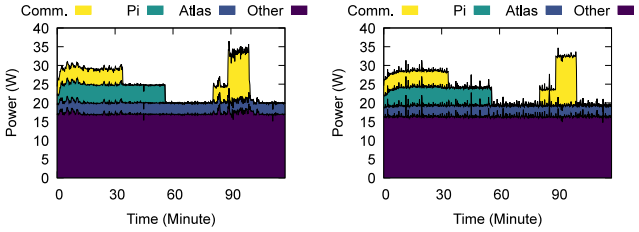


Figure 10: Energy harvesting accuracy.



(a) Real satellite trace. (b) RHONE generated trace.

Figure 11: Energy consumption accuracy.

and thermal) traces and the profiling of COTS computation during the offline phase. Once the setup is completed, the pre-built models can be loaded for subsequent emulations.

7.2 Fidelity

7.2.1 Experimental Setup

To evaluate the emulation fidelity of RHONE in power, thermal, and computation capacity. We conduct our evaluation on two enterprise servers, each equipped with two Intel Xeon E5-2470 v2 processors, 32 GB of DDR4 RAM and 557.9 GB of disk storage. We launch a 25-satellite constellation in a 5×5 grid at 550 km to evaluate RHONE's generated power and temperature traces and computing performance. We use real telemetry data from BUPT-1 and another satellite as ground truth, Cote [13] and StarryNet [43] as baselines.

7.2.2 Power Model Fidelity

Energy harvesting. We evaluate RHONE's energy harvesting behavior by comparing its generated solar array output currents with real data collected from the BUPT1 satellite, alongside a baseline model, CoteSim [13]. We generate 1.5-hour traces for three satellite attitudes: Sun-Targeted (ST), Earth-Backed (EB), and Plane-Rotating (PR) calculate the normalized square error (NSE) between emulated results and ground truth. As shown in Fig. 10, RHONE shows much higher accuracy, especially for EB and PR scenarios where CoteSim's direct solar alignment assumption does not hold. Specifically, RHONE reaches 0.05 NSE at the 99th percentile, outperforming CoteSim (over 1.5 NSE at the 90th

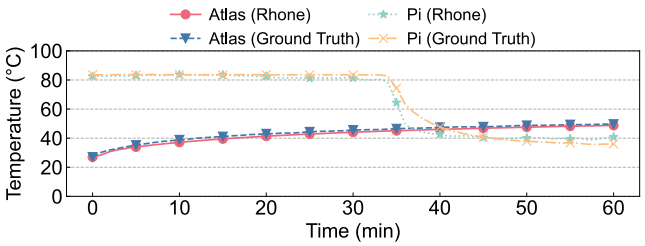
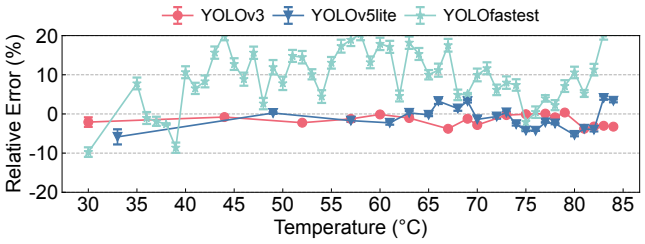
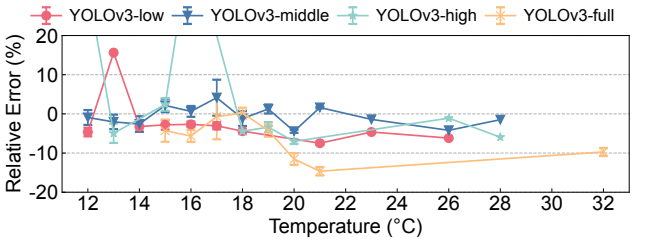


Figure 12: Temperature accuracy.



(a) Raspberry Pi



(b) Atlas 200 DK

Figure 13: Computation accuracy.

percentile) in the PR scenario. These demonstrate RHONE's ability to accurately mimic the energy harvesting behavior of real satellites. Note that both RHONE and CoteSim exhibit long-tail NSEs due to inevitable inaccuracies in orbit calculations. These errors affect the precise timing of satellite transitions between sunlight and eclipse in emulation, which in turn leads to inaccuracies in energy harvesting.

Energy consumption. We compared RHONE's energy consumption component with the actual consumption trace collected from the BUPT1 satellite. To provide a clear view of the power consumption of each component, we show the stacked power usage of different components in Fig. 11. Overall, the models for each component closely simulate the real power trace. Specifically, *Comm.* refers to communication components like X-band, with an average error of 3.0%; *Pi* and *Atlas* are the COTS devices on board, with an average 3.5% and 3.8% error, respectively; *Other* represents all other always-on components with Markov-based consumption pattern, with an average error of 4.1%. Such results demonstrate that RHONE can accurately mimic the energy consumption behavior of real satellites, which indirectly influences the performance of onboard COTS devices.

7.2.3 Thermal Model Fidelity

To evaluate RHONE's thermal model, we use the internal architecture temperature of BUPT-1 and the power consumption trace of the onboard COTS devices as the input of our thermal model. We compare the temperature of emulated COTS devices with that of the real satellite. As Fig. 12 shows, the temperature series of the mirrored chip in RHONE closely matches that of the in-orbit chip, both in terms of values and trends, validating RHONE's accuracy in predicting thermal effects under operational loads, with an average error of 2.5°C and 1.3°C for Pi and Atlas, respectively.

7.2.4 Computation Model Fidelity

To demonstrate RHONE's ability to accurately mimic the computational capacity of onboard chips, we run various DNN applications (YOLOv3-Full/High/Medium/Low, YOLOv5lite, YOLOfastest) from the BUPT-1 open-source dataset [2] on mirrored Raspberry Pi and Atlas 200 DK devices in RHONE with different temperatures. We then compare the inference latency of the emulated results with real-world traces. We show the relative error in Fig. 13, the markers represent the mean, while the upper and lower bounds indicate the standard deviation as we repeat the experiment multiple times. The mean relative error of both Pi and Atlas remains below 5% in most cases with low variations, indicating that RHONE can accurately reflect the impact of thermal throttling on computation capability.

Note that some rare cases exhibit high relative errors (*e.g.,* Atlas running YOLOv3-high at 16°C, and Pi running YOLOfastest). We summarize two possible reasons: *i)* RHONE mirrors COTS chips using on-the-ground profiling data, introducing minor absolute errors compared to the actual latency experienced in space, as shown in Fig. 8. This can be mitigated by thermal vacuum experiments strictly aligned to space, or directly integrating in-orbit satellites to profile the tasks, but both incur high costs and scalability issues; *ii)* Some workloads have short latency (*e.g.,* YOLOfastest completes within 40 ms). Therefore, a small absolute error in emulated inference latency results in high relative errors. This can potentially be addressed by finer-grained container resource tuning, but at the cost of non-trivial scheduling overhead from dynamic quota adjustments.

8 Use Cases

8.1 Satellite Energy Drain Attack

As satellites have limited energy resources, recent studies [27, 28] have explored the effects of energy drain attacks and heavy loads on satellite networks. This combines the emerging focus on satellite-specific network security [69] with longstanding challenges in satellite power management [70, 71]. Using RHONE, we emulate an energy drain attack, demonstrating its strength in accurately modeling energy constraints and evaluating network resilience.

Experiment Setup. The experiment involves a constellation of 720 satellites arranged in a 36×20 mesh grid at an altitude of 550 km. To simulate the attack or burst heavy loads, a target satellite is selected to be the victim, and botnets are used to send bulk traffic through it. The effects are tested through daylight (sun side) and night (shadow side) conditions.

Observations. We present the battery level (state of charge) and the network bandwidth of the victim satellite during the attack in Fig. 14. Our emulation process reveals two key events: *i) Rapid energy drain under heavy load.* The saturated link imposes substantial computation overhead on COTS devices for routing and switching from the onset of the attack, leading to a rapid battery drain. The battery level reaches the safety threshold (60%), prompting the onboard system to forcibly shut down the COTS device, signaling a successful attack. Note that satellites are more vulnerable to such attacks at night, as they experience accelerated energy depletion due to the lack of solar power. *ii) Abrupt network performance drop.* The sudden shutdown of onboard COTS devices causes an immediate bandwidth drop to 0. The bandwidth returns to a normal level (similar to that of BUPT-1) once the devices recover from recharging. Such observations highlight the energy concern for future SCN research, including energy-aware SCN task scheduling, traffic load balancing, and energy-related security issues.

8.2 Real-Time Earth Observation

Earth observation constellations (*e.g.,* Planet Dove constellation [11] with 150 satellites) are actively performing Earth imaging tasks. Given the limited bandwidth and computation capacity of satellites, recent studies [72, 73, 14, 74, 75, 76, 15, 13, 66, 19, 77] have explored novel and efficient strategies for in-orbit processing and data transmission. We emulate real-time earth observation in RHONE, highlighting its strength in replicating processing and transmission constraints for strategy evaluation.

Experiment Setup. We launch a 150-satellite constellation at 550 km, with each satellite having a maximum data transmission capability of 200 Mbps. This configuration replicates the constellation of Planet Dove [11], an operational SatEO project. These satellites capture images and process them with four transmission methods: *(i) Direct:* transmit the raw image immediately; *(ii) Compress:* compression the image with three quality levels ($Q = 75, 50, 25$); *(iii) Detect:* detect objects with YOLOv3 and transmit the objects; *(iv) Reasoning:* convert image content to text and transmit the results. We capture 2000 images and transfer data via FTP.

Observations. We present the bitrates of data transmission, the power consumption of the computing device (Atlas 200 DK) and communication (XBand) hardware, and the satellite battery levels in Fig. 15. Our emulation reveals two key observations: *i) Link usage reduction.* In-orbit processing before image transmission can significantly save communication bandwidth by reducing both link usage time and required bandwidth. Specifically, image compression, object detection, and model inference can decrease link usage by

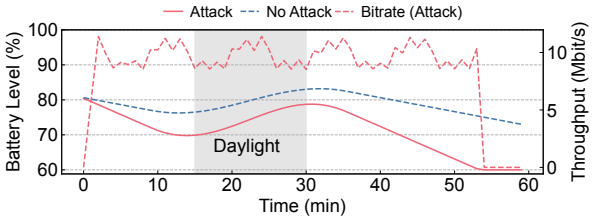


Figure 14: Energy drain attack emulation.

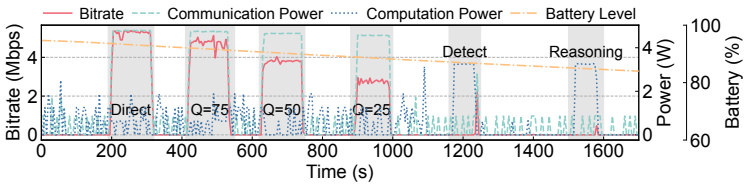


Figure 15: Real-time Earth observation (EO) emulation. Shaded areas indicate the EO working periods.

0.3×, 0.8×, and 0.99×, respectively, compared to direct image transmission. *ii) Energy consumption tradeoffs.* While onboard object detection and reasoning tasks require significantly more computational power than compression tasks, the reduced data transmission size leads to lower communication power consumption. Therefore, performing certain image inference tasks before transmission in SatEO applications is beneficial for both bandwidth reduction and energy conservation. These observations underscore the trade-offs between computation and communication resources in SCN research, especially in-orbit computing for EO tasks.

9 Discussion

Model extendability. While some models in RHONE are built based on telemetry and experimental data from two COTS devices, RHONE allows users to replace or extend models as needed. To expand existing power or thermal models, users can apply the modeling methods provided by RHONE to telemetry data from other satellites or determine model parameters accordingly. Users can also use the hardware-in-the-loop chip mirroring method provided in the computation model to incorporate their own COTS devices.

Other environmental context effects on SCN. Currently, RHONE constructs the energy and thermal context for SCN. While additional system effects on COTS devices – such as radiation failures [34], collision avoidance [78], and physical-layer challenges and security concerns [79] – can also impact SCN performance, these are left for future work.

10 Related Work

In this section, we briefly introduce SCN-related research. The network community is currently focusing on system architecture [80], topology design [81], routing [82, 83], transport-layer congestion control [84], security issues [28], real-world measurements [85], traffic scheduling [72], and on-the-ground emulators [38, 39, 40, 41, 66, 45, 43] for emerging satellite networks. The mobile computing community focuses on verifying the feasibility of satellite image processing applications [86, 87, 88, 89] and on task allocation and resource scheduling algorithms and system design [73, 45, 14, 74, 75, 76]. As satellite networking and computing converge, recent studies investigate SCN applications including real-time EO systems [15], cross-continent satellite video conferencing systems [21], and satellite in-network computing [22, 23]. For example, Li *et al.* [22] explore the feasibility of pushing mobile core network functions to LEO satellite mega-constellations to avoid signaling storm and improve network efficiency. Xing *et al.* [2] conduct a comprehensive in-orbit measurement of satellite COTS computing systems, revealing that energy and thermal contexts in real satellite environments impact satellite behavior and application performance. RHONE serves as a comprehensive emulator accounting for environmental impacts on both computing and networking performance of onboard chips, on which the SCN-related research can be conducted.

11 Conclusion

The growth of SCNs introduces unique characteristics and challenges, which should be carefully considered in designing, building and evaluating SCN applications, demanding accurate, scalable and easy-to-use developing and evaluating tools. In this paper, we introduce RHONE, a full-fledged emulator for SCNs with high satellite- and constellation-level fidelity, high scalability and high usability. It builds models from real in-orbit satellite telemetries, and coordinates the emulation via a Docker container-based environment. Crucially, RHONE provides users with satellite and constellation states (*e.g.,* power, temperature and constellation topology) in real time, and dynamically aligns emulated computing and networking performance with real satellite COTS devices. It enables researchers on Earth to reliably develop, evaluate and optimize SCN applications like satellite earth observation or satellite-backboned global networking without having to access a real in-orbit satellite. We evaluate RHONE 's fidelity using real in-orbit satellite telemetry data, and demonstrate its effectiveness in emulating typical SCN applications as use cases. We believe RHONE will help advance futuristic SCN research, fostering the development of more resilient and efficient space-based systems.

Acknowledgments

We thank our shepherd, Timothy Wood, and the anonymous ATC reviewers for their insightful comments and suggestions. This work is supported by National Key Research and Development Plan, China (Grant No. 2023YFB2903902) and the National Natural Science Foundation of China (Grant No. 62022005, 62272010, 62325201). Chenren Xu is the corresponding author.

References

[1] Satellite on board computer: Sirius tcm leon3ft. https://satsearch.co/products/aac-clyde-sirius-tcm-leon3ft, 2024. Accessed: 2024-09-05.

[2] Ruolin Xing, Mengwei Xu, Ao Zhou, Qing Li, Yiran Zhang, Feng Qian, and Shangguang Wang. Deciphering the enigma of satellite computing with cots devices: Measurement and analysis. In *ACM MobiCom*, 2024.

[3] Christopher Wilson and Alan George. Csp hybrid space computing. *Journal of Aerospace Information Systems*, 15(4):215–227, 2018.

[4] Edward J Wyrwas. Nepp processor enclave (npe) update. In *Presentation for posting on public website*. NASA Electronic Parts and Packaging (NEPP) Program, 2022.

[5] Starlink: high-speed, low latencybroadband internet. https://www.starlink.com, 2024. Accessed: 2024-08-27.

[6] Maya Posch. Starlink's inter-satellite laser links are setting new record with 42 million gb per day. https://hackaday.com/2024/02/05/starlinks-inter-satellite-laser-links-are-setting-new-record-with-42-million-gb-per-day, 2024.

[7] Starlink specifications. https://www.starlink.com/legal/documents/DOC-1400-28829-70, 2024. Accessed: 2024-08-27.

[8] Oneweb. delivering leo connectivity for new network solutions. https://oneweb.net, 2024. Accessed: 2024-08-27.

[9] Telesat. Telesat. https://www.telesat.com, 2023. Accessed: 2023-05-23.

[10] Amazon. Project kuiper. https://www.aboutamazon.com/what-we-do/devices-services/project-kuiper, 2023. Accessed: 2023-05-23.

[11] Planet Dove Earth Observation Constellation. https://www.planet.com/our-constellations, 2024. Accessed: 2024-12-07.

[12] Kuntai Du, Yihua Cheng, Peder Olsen, Shadi Noghabi, and Junchen Jiang. Earth+: On-board satellite imagery compression leveraging historical earth observations. In *ACM ASPLOS*, 2025.

[13] Bradley Denby and Brandon Lucia. Orbital edge computing: Nanosatellite constellations as a new class of computer system. In *ACM ASPLOS*, 2020.

[14] Bradley Denby, Krishna Chintalapudi, Ranveer Chandra, Brandon Lucia, and Shadi Noghabi. Kodan: Addressing the computational bottleneck in space. In *ACM ASPLOS*, 2023.

[15] Bill Tao, Om Chabra, Ishani Janveja, Indranil Gupta, and Deepak Vasisht. Known knowns and unknowns: Near-realtime earth observation via query bifurcation in serval. In *USENIX NSDI*, 2024.

[16] Zhuo Cheng, Bradley Denby, Kyle McCleary, and Brandon Lucia. Eagleeye: Nanosatellite constellation design for high-coverage, high-resolution sensing. In *ACM ASPLOS*, 2024.

[17] Palantir's metaconstellation. https://www.palantir.com/offerings/metaconstellation, 2024. Accessed: 2024-08-27.

[18] Nasa earthdata. https://www.earthdata.nasa.gov/learn/earth-observation-data-basics/data-latency, 2025. Accessed: 2025-01-15.

[19] Deepak Vasisht, Jayanth Shenoy, and Ranveer Chandra. L2d2: low latency distributed downlink for leo satellites. In *ACM SIGCOMM*, 2021.

[20] Nathaniel Bleier, Muhammad Husnain Mubarik, Gary R Swenson, and Rakesh Kumar. Space microdatacenters. In *IEEE/ACM MICRO*, 2023.

[21] Zeqi Lai, Weisen Liu, Qian Wu, Hewu Li, Jingxi Xu, and Jianping Wu. Spacertc: Unleashing the low-latency potential of mega-constellations for real-time communications. In *IEEE INFOCOM*, 2022.

[22] Yuanjie Li, Hewu Li, Wei Liu, Lixin Liu, Yimei Chen, Jianping Wu, Qian Wu, Jun Liu, and Zeqi Lai. A case for stateless mobile core network functions in space. In *ACM SIGCOMM*, 2022.

[23] Lixin Liu, Yuanjie Li, Hewu Li, Jiabo Yang, Wei Liu, Jingyi Lan, Yufeng Wang, Jiarui Li, Jianping Wu, Qian Wu, Jun Liu, and Zeqi Lai. Democratizing Direct-to-Cell low earth orbit satellite networks. In *USENIX NSDI*, 2024.

[24] 5G-Advanced Core Experiments in the Baoyun LEO Satellite by China Mobile. https://m.c114.com.cn/w118-1183668.html, 2021. Accessed: 2024-12-06.

[25] Dongfang Hour China Aerospace News Roundup 6–13 December. https://spacewatch.global/2021/12/spacewatchgl-column-dongfang-hour-china-aerospace-news-roundup-6-13-december, 2021. Accessed: 2024-12-06.

[26] Chinese Private Firm Galactic Energy Puts Five Satellites in Orbit with Second Launch. https://spacenews.com/chinese-private-firm-galactic-energy-puts-five-satellites-in-orbit-with-second-launch, 2021. Accessed: 2024-12-06.

[27] Yaoying Zhang, Qian Wu, Zeqi Lai, Yangtao Deng, Hewu Li, Yuanjie Li, and Jun Liu. Energy drain attack in satellite internet constellations. In *2023 IEEE/ACM IWQoS*, 2023.

[28] Giacomo Giuliari, Tommaso Ciussani, Adrian Perrig, and Ankit Singla. Icarus: Attacking low earth orbit satellite networks. In *USENIX ATC*, 2021.

[29] Tom Etchells and Lucy Berthoud. Developing a power modelling tool for cubesats. In *Annual AIAA/USU Conference on Small Satellites*, 2019.

[30] Jinkyu Lee, Eugene Kim, and Kang G. Shin. Design and management of satellite power systems. In *2013 IEEE 34th Real-Time Systems Symposium*, pages 97–106, 2013.

[31] Shangguang Wang and Qing Li. Satellite computing: Vision and challenges. *IEEE Internet of Things Journal*, 10(24):22514–22529, 2023.

[32] Zeqi Lai, Zonglun Li, Qian Wu, Hewu Li, Weisen Liu, Yijie Liu, Xin Xie, Yuanjie Li, and Jun Liu. Mind the misleading effects of leo mobility on end-to-end congestion control. In *ACM Hot Nets*, 2024.

[33] Jihao Li, Hewu Li, Zeqi Lai, Qian Wu, Yijie Liu, Qi Zhang, Yuanjie Li, and Jun Liu. Satguard: Concealing endless and bursty packet losses in leo satellite networks for delay-sensitive web applications. In *ACM WWW*, 2024.

[34] Haoda Wang, Steven Myint, Vandi Verma, Yonatan Winetraub, Junfeng Yang, and Asaf Cidon. Mars attacks! software protection against space radiation. In *ACM HotNets*, 2023.

[35] 40 starlink satellites doomed by geomagnetic storm. https://earthsky.org/space/40-starlink-satellites-doomed-by-geomagnetic-storm, 2023.

[36] How much do cubesats and smallsats cost? https://nanoavionics.com/blog/how-much-do-cubesats-and-smallsats-cost, 2024. Accessed: 2024-09-05.

[37] Shangguang Wang, Qing Li, Mengwei Xu, Xiao Ma, Ao Zhou, and Qibo Sun. Tiansuan constellation: An open research platform. In *IEEE EDGE*, 2021.

[38] Systems tool kit (stk). https://www.agi.com/products/stk, 2024. Accessed: 2024-08-27.

[39] General mission analysis tool. https://gmat.atlassian.net/wiki/spaces/GW, 2024. Accessed: 2024-08-27.

[40] Sns3. satellite network simulator 3. https://www.sns3.org, 2024. Accessed: 2024-08-27.

[41] Simon Kassing, Debopam Bhattacherjee, André Baptista Águas, Jens Eirik Saethre, and Ankit Singla. Exploring the "internet from space" with hypatia. In *ACM IMC*, 2020.

[42] Zeqi Lai, Hewu Li, and Jihao Li. Starperf: Characterizing network performance for emerging mega-constellations. In *IEEE ICNP*, 2020.

[43] Zeqi Lai, Hewu Li, Yangtao Deng, Qian Wu, Jun Liu, Yuanjie Li, Jihao Li, Lixin Liu, Weisen Liu, and Jianping Wu. Starrynet: Empowering researchers to evaluate futuristic integrated space and terrestrial networks. In *USENIX NSDI*, 2023.

[44] Vaibhav Bhosale, Ketan Bhardwaj, and Ada Gavrilovska. Toward loosely coupled orchestration for the LEO satellite edge. In *USENIX HotEdge 20*, 2020.

[45] Tobias Pfandzelter and David Bermbach. Celestial: Virtual software system testbeds for the leo edge. In *ACM/IFIP Middleware*, 2022.

[46] Zhiguo Liu, Jie Zhu, Jiangmei Zhang, and Qingli Liu. Routing algorithm design of satellite network architecture based on sdn and icn. *Int. J. Satell. Commun. Netw.*, 38(1):1–15, December 2019.

[47] Debopam Bhattacherjee and Ankit Singla. Network topology design at 27,000 km/hour. In *ACM CoNEXT*, 2019.

[48] Yuanjie Li, Hewu Li, Lixin Liu, Wei Liu, Jiayi Liu, Jianping Wu, Qian Wu, Jun Liu, and Zeqi Lai. "Internet in Space" for Terrestrial Users via Cyber-Physical Convergence. In *ACM HotNets*, 2021.

[49] Sebastian Sabogal, Alan George, and Christopher Wilson. Reconfigurable framework for environmentally adaptive resilience in hybrid space systems. *ACM Transactions on Reconfigurable Technology System*, 13(3), jul 2020.

[50] Abderezak Lashab, Mohammad Yaqoob, Yacine Terriche, Juan C. Vasquez, and Josep M. Guerrero. Space microgrids: New concepts on electric power systems for satellites. *IEEE Electrification Magazine*, 8(4):8–19, 2020.

[51] Qing Li, Shangguang Wang, Xiao Ma, Ao Zhou, Yue Wang, Gang Huang, and Xuanzhe Liu. Battery-aware energy optimization for satellite edge computing. *IEEE Transactions on Services Computing*, 17(2):437–451, 2024.

[52] Wikipedia. Markov chain. https://en.wikipedia.org/wiki/Markov_chain, 2024. Accessed: 2024-08-27.

[53] Quang Gao, Keith J Blow, David J Holding, Ian W Marshall, and XH Peng. Radio range adjustment for energy efficient wireless sensor networks. *Ad hoc networks*, 4(1):75–82, 2006.

[54] Y Li, Bertan Bakkaloglu, and Chaitali Chakrabarti. A comprehensive energy model and energy-quality evaluation of wireless transceiver front-ends. In *IEEE Workshop on Signal Processing Systems Design and Implementation, 2005.*, 2005.

[55] Heather Hanson, Stephen W. Keckler, Soraya Ghiasi, Karthick Rajamani, Freeman Rawson, and Juan Rubio. Thermal response to dvfs: Analysis with an intel pentium m. In *ACM/IEEE ISLPED*, 2007.

[56] Beomsik Kim and Hoeseok Yang. Temperature sensor assisted lifetime enhancement of satellite embedded systems via multi-core task mapping and dvfs. *Sensors*, 19(22), 2019.

[57] B. Yost. https://www.nasa.gov/sites/default/files/atoms/files/2022_soa_full_0.pdf, title=State-of-the-Art Small Spacecraft Technology, 2023.

[58] Solideworks. thermal analysis software. https://www.solidworks.com, 2024. Accessed: 2024-08-27.

[59] Adelina Ioana Ilieş, Rajmond Jánó, and Gabriel Chindriş. Thermal management of system-on-chip devices used in satellite systems under low temperature conditions. In *International Spring Seminar on Electronics Technology*, 2016.

[60] Casper Versteeg and David L Cotten. Preliminary thermal analysis of small satellites. *Small Satellite Research Laboratory, The University of Georgia, Athens, Georgia*, 30602, 2018.

[61] Franyelit Suárez-Carreño and Luis Rosales-Romero. Convergency and stability of explicit and implicit schemes in the simulation of the heat equation. *Applied Sciences*, 11(10):4468, 2021.

[62] Yolov3 algorithm. https://github.com/ultralytics/yolov3, 2024. Accessed: 2024-08-27.

[63] Yolov5-lite algorithm. https://github.com/ppogg/YOLOv5-Lite, 2024. Accessed: 2024-08-27.

[64] Yolo-fastest algorithm. https://github.com/dog-qiuqiu/Yolo-Fastest, 2024. Accessed: 2024-08-27.

[65] Simon Kassing, Debopam Bhattacherjee, André Baptista Águas, Jens Eirik Saethre, and Ankit Singla. Exploring the "internet from space" with hypatia. In *ACM IMC*, 2020.

[66] Bradley Denby and Brandon Lucia. Orbital edge computing: Machine inference in space. *IEEE Computer Architecture Letters*, 18(1), 2019.

[67] Inc. TE Technology. Temperature controllers. https://tetech.com/product-category/temperature-controllers, 2024. Accessed: 2024-09-05.

[68] Bingyang Wu, Zili Zhang, Zhihao Bai, Xuanzhe Liu, and Xin Jin. Transparent gpu sharing in container clouds for deep learning workloads. In *USENIX NSDI*, 2023.

[69] Cyber Peace Institute. Case Study ViaSat, 2022.

[70] Dwaine K Coates, Chris L Fox, DJ Standlee, and BK Grindstaff. Advanced nickel-hydrogen spacecraft battery development. In *NASA. Marshall Space Flight Center, The 1993 NASA Aerospace Battery Workshop*, 1994.

[71] Ashutosh Dutta and Yechiam Yemini. Power management of leos under bursty broadband traffic. In *17th AIAA international communications satellite systems conference and exhibit*, page 1317, 1998.

[72] Bill Tao, Maleeha Masood, Indranil Gupta, and Deepak Vasisht. Transmitting, fast and slow: Scheduling satellite traffic through space and time. In *ACM MobiCom*, 2023.

[73] Tobias Pfandzelter, Jonathan Hasenburg, and David Bermbach. Towards a computing platform for the leo edge. In *Proceedings of the International Workshop on Edge Systems, Analytics and Networking*, 2021.

[74] Pfandzelter, Tobias and Bermbach, David. Qos-aware resource placement for leo satellite edge computing. In *Proceedings of International Conference on Fog and Edge Computing*, 2022.

[75] Renchao Xie, Qinqin Tang, Qiuning Wang, Xu Liu, F Richard Yu, and Tao Huang. Satellite-terrestrial integrated edge computing networks: Architecture, challenges, and open issues. *IEEE Network*, 34(3):224–231, 2020.

[76] Minglei Tong, Xiaoxiang Wang, Song Li, and Liang Peng. Joint offloading decision and resource allocation in mobile edge computing-enabled satellite-terrestrial network. *Symmetry*, 14(3), 2022.

[77] Bill Tao, Maleeha Masood, Indranil Gupta, and Deepak Vasisht. Transmitting, fast and slow: Scheduling satellite traffic through space and time. In *ACM MobiCom*, 2023.

[78] Yuanjie Li, Hewu Li, Wei Liu, Lixin Liu, Wei Zhao, Yimei Chen, Jianping Wu, Qian Wu, Jun Liu, Zeqi Lai, and Han Qiu. A networking perspective on starlink's self-driving leo mega-constellation. In *ACM MobiCom*, 2023.

[79] Robin Bisping, Johannes Willbold, Martin Strohmeier, and Vincent Lenders. Wireless signal injection attacks on vsat satellite modems. In *USENIX Security*, 2024.

[80] Deepak Vasisht, Jayanth Shenoy, and Ranveer Chandra. L2d2: low latency distributed downlink for leo satellites. In *ACM SIGCOMM*, 2021.

[81] Debopam Bhattacherjee and Ankit Singla. Network topology design at 27,000 km/hour. In *ACM CoNEXT*, 2019.

[82] Mark Handley. Delay is not an option: Low latency routing in space. In *ACM HotNets*, 2018.

[83] Mark Handley. Using ground relays for low-latency wide-area routing in megaconstellations. In *ACM HotNets*, 2019.

[84] Xu Li, Feilong Tang, Jiacheng Liu, Laurence T. Yang, Luoyi Fu, and Long Chen. Auto: Adaptive congestion control based on Multi-Objective reinforcement learning for the Satellite-Ground integrated network. In *USENIX ATC*, 2021.

[85] Mohamed M. Kassem, Aravindh Raman, Diego Perino, and Nishanth Sastry. A browser-side view of starlink connectivity. In *ACM IMC*, 2022.

[86] Gianluca Giuffrida, Luca Fanucci, Gabriele Meoni, Matej Batič, Léonie Buckley, Aubrey Dunne, Chris van Dijk, Marco Esposito, John Hefele, Nathan Vercruyssen, Gianluca Furano, Massimiliano Pastena, and Josef Aschbacher. The ϕ-sat-1 mission: The first on-board deep neural network demonstrator for satellite earth observation. *IEEE Transactions on Geoscience and Remote Sensing*, 60:1–14, 2022.

[87] Ximing Yu, Yu Peng, and Liansheng Liu. Cloud detection using fully convolutional network with zynq soc for spaceborne application. In *IEEE ICSMD*, 2021.

[88] Muhammad Hasif bin Azami, Necmi Cihan Orger, Victor Hugo Schulz, Takashi Oshiro, and Mengu Cho. Earth observation mission of a 6u cubesat with a 5-meter resolution for wildfire image classification using convolution neural network approach. *Remote Sensing*, 14(8):1874, 2022.

[89] Yuan Yao, Zhiguo Jiang, Haopeng Zhang, and Yu Zhou. On-board ship detection in micro-nano satellite based on deep learning and cots component. *Remote Sensing*, 11(7):762, 2019.

Roaming Free in the VR World with MP^2

Yifei Xu[1], Xumiao Zhang[2,5], Yuning Chen[3], Pan Hu[4], Xuan Zeng[2], Zhilong Zheng[2], Xianshang Lin[2], Yanmei Liu[2], Songwu Lu[1], Z. Morley Mao[5], Wan Du[3], Dennis Cai[2], Ennan Zhai[2], Yunfei Ma[4]
[1]*University of California, Los Angeles*
[2]*Alibaba Cloud*
[3]*University of California, Merced*
[4]*Uber Technologies, Inc.*
[5]*University of Michigan*

Abstract

Free-roaming VR which allows a group of users to navigate in rooms and even buildings, enhances the VR experience by making it more immersive and interactive. Streaming VR videos over wireless enables unconstrained experiences but raises unprecedented requirements in mobility, efficiency, and scalability. Existing solutions fail in one or more of the following challenges: maintaining low latency during handover, balancing loads on different APs, and stabilizing bitrate for competing users, due to their decentralized nature where each user lacks information about others and makes locally optimal decisions. To address these problems, we present MP^2, a centralized VR streaming system that coordinates multiple Wi-Fi links and video bitrates among users for better QoE. A centralized controller collects cross-layer information from each user and makes better decisions based on global information. It achieves this in a timely manner through accurate modeling and the use of efficient pruning and partitioning algorithms. To our knowledge, MP^2 is the first centrally coordinated VR streaming system that supports multi-user free-roaming. Comprehensive benchmarks including real-world tests, large-scale emulation, and trace-driven user studies, confirm the effectiveness of MP^2 against state-of-the-art solutions. It achieves 35× improvement in tail latency, 1.56× in bitrate, and 1.86× in QoE over state-of-the-art baselines. MP^2 achieves up to a 99.1% improvement in mean opinion scores according to the user study.

1 Introduction

Among the rapidly growing Virtual Reality (VR) ecosystem, free-roaming VR has gained significant traction as it provides a more immersive and interactive experience. In contrast to traditional VR where users are constrained to a small room or seated position, free-roaming VR users can freely walk, run, and explore the virtual world in a much larger physical space. Notable examples include Zero Latency VR [18] with perceptual tricks that enable players to travel up to 1 km to journey through the virtual world within a 400 m^2 room, and Sandbox VR Game [12] that features a group of players collaborating with each other in a shared virtual adventure to defeat aliens, as illustrated in Figure 1. Besides gaming, free-roaming virtual reality has found its way to virtual tourism [17], training [15], and education [13].

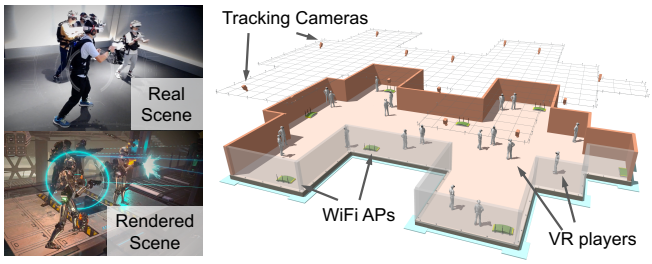
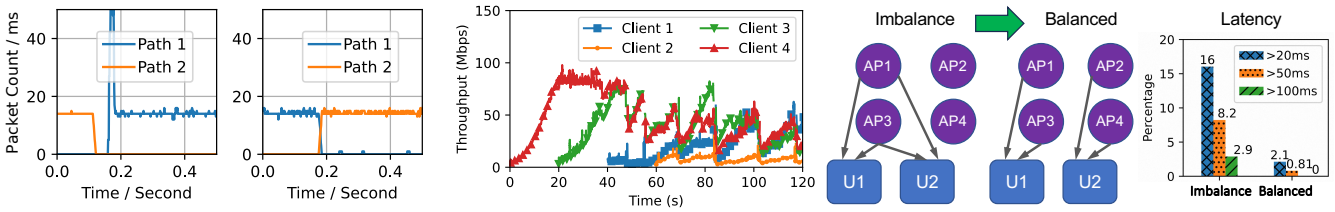


Figure 1: Real-world/rendered scenes and settings of a free-roaming VR scenario. Although existing commercial systems still require users to carry a heavy backpack (top left), a wireless streaming system would remove the burden.

Existing on-device VR rendering solutions necessitate that users either carry a heavy backpack or wear a bulky headset and battery pack for the rendering hardware, which limits immersion. Wireless streaming offloads rendering but needs to meet several requirements in free-roaming VR scenarios: *(1) Mobility*: The system must support wireless operation and allow users to roam freely across different access points (APs) without experiencing significant latency or disconnections, as a single AP cannot provide sufficient coverage and bandwidth. *(2) Scalability*: The system should be capable of supporting multiple users simultaneously, each with its own control loop, without compromising the quality of service. *(3) Efficiency*: It is crucial for the system to schedule available bandwidth resources efficiently to accommodate a large group of users.

It is not surprising to find that existing solutions often fail to meet the requirements of free-roaming VR as they are not specifically designed for such use cases. To better understand the challenges, we build a VR streaming system



(a) Large (≈50 ms) packet gap during handover (left) *vs.* seamless handover (right).

(b) Bitrate fluctuation of four competing streams.

(c) Example of uneven AP traffic load and how balancing improves latency.

Figure 2: Challenges of supporting free-roaming VR with state-of-the-art streaming system.

with XLINK [110], a state-of-the-art multipath transport, and ALVR [2], the most popular open-source VR streaming solution. The experiment results are shown in Figure 2. We summarize the challenges as follows:

• *High handover latency*: Handover between APs can lead to significant performance degradation [85]. While the baseline uses a multipath protocol designed to mitigate latency spikes during handover, it fails to account for the fact that Wi-Fi radios require time to wake up and achieve their optimal throughput. This oversight results in a substantial packet gap compared to the ideal seamless migration scenario, as illustrated in Figure 2a.

• *Unstable bitrate*: Deciding which encoding bitrate to use is already considered challenging over dynamic wireless links. This becomes more complex when coupled with multipath transport, which is known as the double-control loop problem [24,50]. The problem is further exacerbated in the context of multiple players competing for resources, resulting in unstable bitrates for players, as shown in Figure 2b.

• *Imbalance of loads*: Although the baseline solution can distribute traffic between different connected APs, it lacks control over which AP to connect to, nor does it have a global view. This results in an imbalance of loads with users concentrated on a subset of the APs, leading to sub-optimal performance, as shown in Figure 2c.

Similar to the baseline solution, current systems fall short in addressing one or more of the above challenges, as detailed in Table 1. We attribute the root cause to a lack of coordination in their architecture design, including 1) no horizontal coordination among competing users and APs; 2) no vertical coordination across different layers including link, transport, and application layers. Users have to rely on probing to make decisions in a decentralized manner, which is inefficient in free-roaming VR scenarios. While certain solutions manage to tackle some of the challenges, they often require kernel modifications or the use of special hardware, complicating the deployment, as shown in Table 1.

Our answer to these problems is **MP^2** (Multi-Path for Multi-Players), *a centralized overlay system that has a global view of the entire stack and coordinates link/path/bitrate decisions across free-roaming VR users on different APs*. MP^2 can be separated into a data plane and a control plane. The data plane of MP^2 is built around multipath QUIC tunnels that create a virtualized interface to route VR traffic according to the control plane's decisions. The control plane, on the other hand, obtains a global view of the system by gathering cross-layer information, including Wi-Fi PHY and VR application data from all users. By leveraging global knowledge, it orchestrates link-level AP association, transport-level path selection, and application-level bitrate adaptation to optimize Quality of Experience (QoE), and judiciously coordinates different streams to avoid performance degradation when enforcing the arrangement. MP^2 is practical to deploy as both its control and data planes operate in the user space thus requiring no modifications of the kernel, and can be implemented with low-cost commercial Wi-Fi hardware.

In addition to its unique centralized architecture, MP^2 incorporates two significant advancements to achieve its objective. The first is an efficient decision algorithm that optimizes global QoE for large-scale systems in a timely manner. We combine mathematical observations and algorithms optimization including Gaussian Mixture Modeling (GMM) of frame statistics, Modulation and Coding Scheme (MCS)-aware pruning, location-based partitioning, and adaptive topology stabilization. The second advancement is the coordinated seamless migration method which utilizes a combination of path warmup, bitrate guidance, and redundant transmission to smooth out latency spikes during migration.

We conduct comprehensive evaluations including real-world tests, large-scale emulation, and trace-driven user studies on commercial devices to benchmark and understand the performance gain of MP^2. Real-world tests confirm the effectiveness of seamless migration that reduces tail latency by more than an order of magnitude. Large-scale emulation results indicate that MP^2 consistently outperforms state-of-the-art solutions at all scales, achieving a 35× improvement in terms of tail latency, a 1.56× improvement in bitrate, and a 1.86× QoE improvement. The improvements in bitrate and latency translate into significant perceptual quality gain over different VR scenes, resulting in up to a 99.1% improvement in Mean Opinion Scores (MOS) according to our user study.

Our major contributions can be summarized as follows:

- We are the first to identify the requirements and challenges of wireless video streaming for free-roaming VR, and pro-

Table 1: Comparison of MP^2 to related video streaming systems.

Category	System	Application	Architecture	Implementation	Low-latency Handover	Bitrate Coordination	Cross-AP Load Balancing
Bitrate coordination	Habitus [105]	multi-path VR	◐ cross-AP cross-layer	○ special hardware	○	●	○
	Firefly [58]	multi-user VR	◐ cross-user cross-layer	◐ kernel-space	○	●	○
	Chen *et al.* [27]	multi-user VR	◐ cross-user cross-layer	○ special hardware	○	●	○
	Minerva [70]	multi-user VoD	◐ cross-user	● user-space	○	●	○
Multi-AP coordination	WGTT [84]	MU-MIMO	◐ cross-AP	○ special hardware	●	○	●
	ClientMarshal [23]	MU-MIMO	◐ cross-AP	○ special hardware	○	○	●
Reducing handover latency	Paasch *et al.* [74]	multi-path	◐ cross-AP	◐ kernel-space	◐	○	○
	ECF [56]	multi-path	◐ cross-AP	◐ kernel-space	◐	○	○
	XLINK [110]	multi-path VoD	◐ cross-AP cross-layer	● user-space	◐	○	○
This work	**MP^2**	free-roaming multi-user VR	● cross-user cross-AP cross-layer	● user-space	●	●	●

Legend: ●: full support; ◐: partial support; ○: no support.

pose the first centralized multipath multiplayer orchestration system as an alternative to existing decentralized solutions that fail to meet the needs.
- We present the design and implementation of MP^2's data and control planes, along with key enablers including an efficient scheduler and seamless migration mechanism to solve the challenges of mobility and scalability. MP^2 is implemented in the user space, ensuring easy deployment without the need for special hardware.
- We perform extensive evaluations on MP^2 through real-world testing, large-scale emulation, and trace-driven user studies to demonstrate its superior supports for mobility, scalability, and improvements in latency, bitrate, as well as perceptual quality over state-of-the-art baselines.

2 Background and Motivations

2.1 Requirements of Free-Roaming VR

QoE requirements: Creating life-like, immersive VR experiences is the holy grail for multimedia technology. Ideally, to match human perception, a resolution of 9720×8100 per eye is needed, assuming a 162 × 135 degree field of view and 60 pixels per degree, thus requiring a whopping 4 Gbps of bandwidth [30]. Although the peak/average bitrates are 140/80 Mbps on Oculus Pro and Quest 2, we anticipate higher bitrates with next-generation, higher-resolution headsets. Latency is equally important. The optimal motion-to-photon latency should be 20 ms or less [52, 89–91, 103], which poses significant challenges to the entire video pipeline, including rendering, encoding, transport, decoding, and display. As a result, we report percentages of frames with >20/50/100 ms as the latency metric, though MP^2 is designed to be flexible with different latency targets (§6.2.3). Ideally, all these percentages should be zero. We also include a user study for a more subjective understanding of the performance (§6.3).

Number of players and APs: An example setup of the free-roaming VR is shown in Figure 1. Current free-roaming VR supports 4 to 12 players per venue [12, 18], and there is a growing demand to support even more players [108] in the building scale. This not only accommodates larger groups, but also helps to increase sales per unit area. Considering the aforementioned bandwidth requirement for each user and the maximum throughput supported by modern Wi-Fi [7], we estimate that each AP can reliably serve up to three users. Taking a 48-user scenario as an example, this requires at least 16 APs, or more, to accommodate redundancy and account for varying and uneven user distribution. Given the number of users and APs, it is crucial to keep scalability in mind. MP^2 is designed to achieve this goal, and we present large-scale studies in §6.2.

2.2 Key Observations

The following key observations motivate the design of MP^2:

Observation I: Centralization beats decentralization. Existing decentralized systems cannot effectively manage the complex dynamics of free-roaming VR scenarios due to their inability to coordinate decisions across users and network layers. This can lead to high handover latency, unstable bitrates, and imbalanced loads, all of which degrade the quality of experience for VR users. In contrast, the centralized design of MP^2 not only solves these problems, but also simplifies deployment and maintenance.

Observation II: Single Wi-Fi is insufficient. Handover and reliability are limiting factors for a single Wi-Fi link in free-

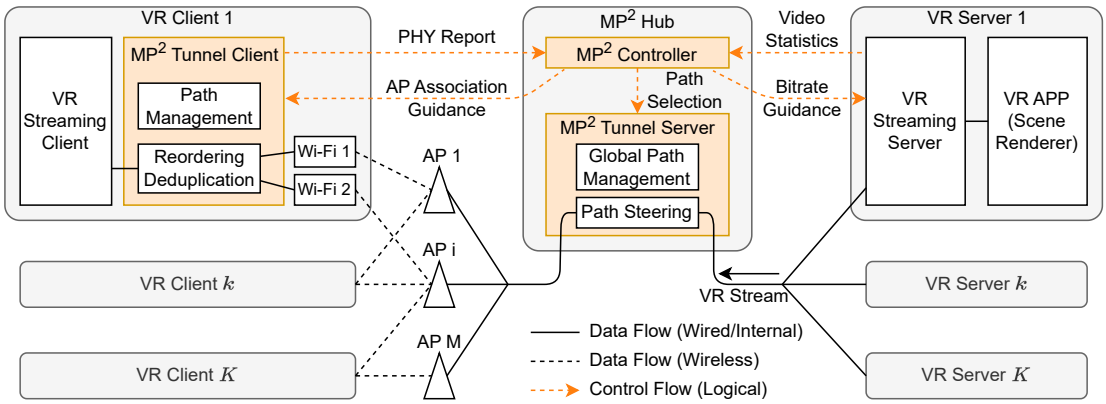


Figure 3: System architecture of MP^2 consists of multi-homed VR clients, MP^2 Hub, and VR servers.The data flows are represented in black lines, with solid lines for wired data, and dotted lines for wireless data. The logical control flows are indicated by colored, dotted arrows.

roaming VR. Wi-Fi handover involves hundred-ms level network outages even with fast handover mechanisms [21,35,88], which is not acceptable for the free-roaming VR experience. Another factor to consider is reliability. A single Wi-Fi link can be prone to interference, congestion, and signal degradation, leading to packet loss, jitter, and increased latency. Recent studies [33,41,46,65,79,80,110,112] show that combining multiple links provides better stability and quality for mobile users. We present a more detailed discussion in §8.

Observation III: User-space implementation simplifies deployment. Operating in user space allows MP^2 to be deployed without OS kernel modifications or special hardware, which significantly lowers the barrier to adoption (see Table 1). This also facilitates easier updates and maintenance allowing quick adaptation to evolving technologies and user needs. We explain the design of MP^2 in §3.1.

Inspired by these observations, we then explain how we build MP^2 to overcome challenges and meet the requirements of free-roaming VR in the following sections.

3 Overview of MP^2

MP^2 orchestrates VR streams for multi-homing users by collecting cross-layer and cross-user information, then performing path assignment and migration for optimal overall QoE. The system architecture, depicted in Figure 3, consists of endpoints including VR clients and servers, as well as network infrastructure components including Wi-Fi APs and Ethernet switches. MP^2 incorporates a centralized hub, namely MP^2 hub, to coordinate between multiple paths and users. Note that the MP^2 hub can run on one of the servers, eliminating the need for additional hardware. Also, we focus on downlink traffic (from server to client) as uplink traffic, which primarily comprises of VR tracking and control messages, requires minimal bandwidth.

The VR client (headset) is equipped with multiple Wi-Fi radios, runs a VR streaming client application, and integrates a tunnel client built on top of multipath QUIC (MPQUIC [59]). It sends PHY layer reports to the MP^2 controller running on the MP^2 hub, and receives AP association guidance from the controller. The VR servers run a VR scene renderer that generates raw video streams, a GPU/CPU video encoder that compresses the frames, and a VR streaming server that controls the sending rate. The video streams are sent to the tunnel server inside the MP^2 hub and then distributed to clients via Wi-Fi APs according to the path selection decision from the MP^2 controller. Similar to the client, the VR servers also send video statistics to the MP^2 controller which generates bitrate guidance based on the statistics.

The bitrate guidance works by setting a cap on top of the target bitrate input for codec while keeping the original Adaptive Bitrate (ABR) algorithm functioning if its suggested bitrate is lower. This provides two-fold benefits: 1) it guarantees that no stream will exceed its optimal bitrate (flattening the curve), and 2) it allows MP^2 to make use of any advanced ABR algorithm that may respond faster than the MP^2 control loop or even provide more advanced features [45,63,92,104].

We first introduce the MP^2 data plane (§3.1), including the tunnel server and clients that carry video traffic, and then present techniques for seamless connection migration in MP^2 (§3.2). We describe the MP^2 controller which makes centralized routing decisions according to cross-layer information in the next section (§4).

3.1 MP^2 Data Plane

The goal of the MP^2 tunnel is to provide the data plane for handling VR traffic while facilitating real-world deployment by masking the complexity of the underlying links. It consists of a tunnel server and client:

Tunnel Server: It uses a virtual tunnel interface to receive VR video streams and conduct path steering based on decisions from the MP^2 controller. It encapsulates all incoming packets with a tunnel header, assigning new IPs and ports for diverse

path forwarding. Additionally, it manages a global path table for tracking client-server mappings and IP/port details for each stream's paths.

Tunnel Client: It is positioned between the VR streaming client and the Wi-Fi radio on each VR client. On the control plane, the tunnel client is responsible for local path management by maintaining a local path table. It collects PHY layer information and transmits it to the MP^2 controller. Additionally, it relays the AP association guidance received from the MP^2 controller to the Wi-Fi radios to establish association on-demand. On the data plane, the tunnel client exposes a virtual tunnel interface to the VR streaming client as the egress port for video streams.

3.2 Coordinated Seamless Migration

The centralized architecture of MP^2 is required by the Client-Server model of the QUIC protocol, but it also provides a coordinated scheduler to maximize overall QoE as described in the previous section. In fact, we find that the benefit of a centralized architecture goes beyond that: it enables a coordinated workflow for seamless connection migration:

Path warmup: Before initiating a connection migration, the MP^2 controller sends a small amount of probing traffic (*e.g.,* one packet per 10 ms) over the target link for a short period to warm up the Wi-Fi radio. While the Wi-Fi beaconing mechanism may also help keep the radio active to some extent, it is primarily intended for network management and typically with over 100 ms intervals, which is insufficient for seamless handover (confirmed empirically by our observation in Figure 2a).

Redundant transmission: During connection migrations, we adopt a transient full-redundant transmission on both paths to minimize packet loss and latency. The path warmup and redundancy mechanisms introduce a fractional overhead, but they are only required for a short time window (100 ms for path warmup and 50 ms for redundant transmission based on our practice), and therefore negligible. Our evaluation in 6.2.1 also shows significant empirical gains from both.

Bitrate guidance: Migration can disrupt streams on an AP by breaking the original equilibrium. Instead of waiting for the underlying ABR algorithm to achieve the new equilibrium, which can take over 10 seconds according to our evaluation 8a, the MP^2 controller proactively issues bitrate guidance that enforces a maximum bitrate cap for all users on the AP. The guidance value is calculated by MP^2 controller (§4) to optimize global QoE. As a result, the new user quickly reaches the ideal bitrate, enabling higher overall bitrates along with lower latencies. Note that bitrate guidance functions as a cap, only intervening when the existing ABR attempts to use excessive bitrate. This allows MP^2 to coexist seamlessly with other traffic, as the ABR continues to adapt when bandwidth is occupied by competing traffic.

While not all these techniques are surprising, a combination of them forms the perfect recipe. To understand the performance of coordinated seamless migration, we present a comprehensive evaluation of seamless migration and ablation study in §6.1. Next, we introduce the MP^2 controller that performs all the heavy lifting behind the scenes.

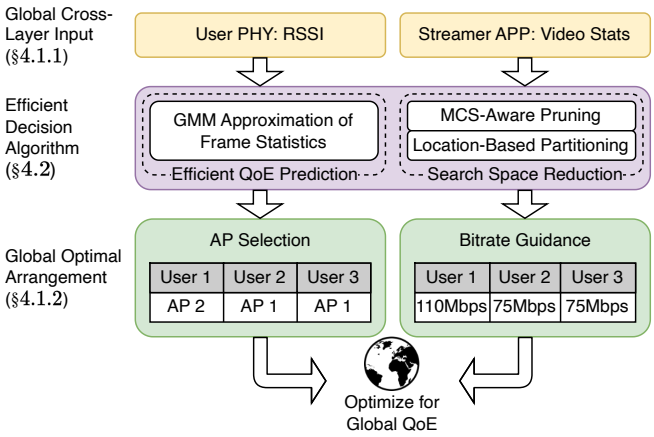


Figure 4: Overview of the MP^2 controller.

Table 2: Notations used in the controller design, in each timestamp.

Notation	Definition for user k
$\boldsymbol{L}_{k,m}$	Matrix of link rates from clients (Mbps).
$\boldsymbol{S}_{k,t}$	Matrix of frame sizes from clients (Byte).
$\boldsymbol{A}_{k,m}$	Binary matrix of user-AP connections.
$\boldsymbol{B}_{\boldsymbol{k}}$	List of bitrate guidance for each user.
$\boldsymbol{P}_{k,i}$	% of frames with latency $>$ threshold.
w_i	Weight for the corresponding P.

4 MP^2 Controller

The MP^2 controller acts as the brain of the control plane, positioned atop the tunnels within the data plane. Figure 4 shows an overview of the controller. We start with the design of MP^2 controller, including control objectives, inputs, and outputs. Then, we present an efficient decision algorithm enhanced through mathematical observations and algorithms including approximation, pruning, and partitioning.

4.1 Control Problem Formulation

In this section, we define the control input/output and optimization goal. The notations used can be found in Table 2.

4.1.1 Control Inputs

The MP^2 controller collects cross-layer information from both clients and servers during each decision-making round. It focuses on adapting to changes in user/AP topology rather than fine-grained Wi-Fi link fluctuations. Consequently, the decision interval need not be long and is adjusted based on mobility levels[1]. The collected data includes:

[1]In this paper, the control interval is set to a default of one second to accommodate walking or jogging users with speeds between 0 to 3 m/s.

Link rate from clients: The MP^2 tunnel client periodically collects Wi-Fi `SSID` and `RSSI` information using `iw dev wlan link` and `iw dev scan` for connected and nearby Wi-Fi APs. We then convert the RSSI to link capacity using standard MCS (Modulation and Coding Scheme) table [1, 7] and AP configurations such as generation (*e.g.,* Wi-Fi 5/6), bandwidth, number of spatial streams, and guard interval [2]. While raw CSI information or dedicated measurement streams could provide more accurate capacity estimates, these approaches are either impractical due to special firmware/driver requirements [39] or too resource-intensive for tight bandwidth budgets. RSSI strikes a balance, offering both practicality and decent estimation [102]. The MP^2 hub aggregates PHY reports from all clients as a link rate matrix $\boldsymbol{L}_{k,m}$ with values in Mbps where $k \in [1,K]$ and $m \in [1,M]$ for K clients and M APs. We fill $\boldsymbol{L}_{k,m}$ with zeros if AP m is out of range for client k.

Statistics of VR frames from servers: The actual frame size of the VR video stream varies from frame to frame, as shown in Figure 5. Simply dividing the bitrate by the frame rate can lead to significant inaccuracies [49], particularly when the QoE function is sensitive to transient or tail latency, as is often the case in VR streaming. To more accurately characterize VR traffic at the per-stream level, we collect recent T video frame size information from all users as $\boldsymbol{S}_{k,t}$ with values in Bytes where $k \in [1,K]$ and $t \in [1,T]$ for K clients and T samples. Instead of using raw frame size for scheduling, we use the Gaussian function to approximate the size, as also depicted in Figure 5. We will discuss Gaussian approximation in detail in §4.2. The system also includes an empirical preset to help cold-start the system.

4.1.2 Control outputs

The MP^2 controller generates the following control outputs:

Arrangement of Path/AP selection: As mentioned in §3.1, the tunnel server needs to select a path (AP) for each user denoted as a logical matrix $\boldsymbol{A}_{k,m}$ with values in the Boolean domain $\{0,1\}$, where 1 means establishing or maintaining a connection between user k and AP m. To ensure all users have exactly one active connection, $\sum_{m=1}^{M} \boldsymbol{A}_{k,m} = 1$ for all $k \in \{1,K\}$. There are two cases when executing $\boldsymbol{A}$: in the first case, all users have connections to APs ($\sum_{k=1}^{K}\sum_{m=1}^{M} \boldsymbol{A}_{k,m} = K$) so the data plane can switch the path directly; in the second case, a user k may not satisfy the aforementioned condition, thus requiring one of the Wi-Fi NICs on user k to associate with the AP first and then apply the path switch. As a result, MP^2 operates with a minimum number of two Wi-Fi radios[3], one for the active link that carries the VR stream, and the other for scanning AP downlink MCSs passively.

Bitrate guidance: With the centralized framework, MP^2 is able to steer the encoder target bitrate on each VR server by overriding ABR with a global bitrate guidance $\boldsymbol{B_k}$ for $k \in [1,K]$. This approach has several advantages: 1) Decentralized ABR tends to achieve fairness among streams passing through the same AP, while MP^2 can apply different bitrates on each stream for better overall QoE. 2) It is generally hard to tune a decentralized ABR algorithm to settle on the optimal bitrate for high-bandwidth, low-latency applications (as depicted in Figure 2b). 3) We can enable seamless path migration by coordinating bitrate between contending users, which we explain in detail in §3.2.

[2]Although it is possible to get AP to client MCS directly using the `iw/nmcli` command, we find RSSI tends to be more stable as it is resilient to whether the link is in use or not.

[3]We discuss the possibility of using a single Wi-Fi radio in §7.

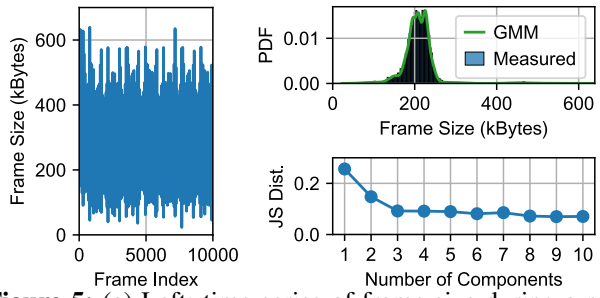


Figure 5: (a) Left: time series of frame size during a real VR session. (b) Top Right: GMM approximation. (c) Bottom Right: Jensen-Shannon distance of the approximation *vs.* number of components in GMM, where 0 means identical and 1 means completely different.

4.1.3 Optimization Goal

MP^2 uses a flexible QoE goal that can be customized for various VR scenarios from fast-paced first-person shooting games to slow-paced leisure games:

$$\boldsymbol{Q} = \sum_{k=1}^{K} \boldsymbol{B}_k * (1 - \sum_{i=1}^{3} w_i * \boldsymbol{P}_{k,i}) \quad (1)$$

where $\boldsymbol{P}_{k,1}, \boldsymbol{P}_{k,2}, \boldsymbol{P}_{k,3}$ represent the percentage of frames with latency $>$ 20/50/100 ms for user k, and w_i represent their respective weights that can be tailored for different needs[4].

4.2 The MP^2 Decision Algorithm

Optimizing the QoE goal shown in Equation 1 is non-trivial. A brute-force approach would require searching across all possible combinations of $(\boldsymbol{A},\boldsymbol{B})$, then running a simulator with recorded video traces to compute $\boldsymbol{P}$ by counting the delayed frames. The time complexity for K clients and M APs is $M^K \cdot B^K \cdot C$, where B is the number of discrete bitrate levels, and C is the time needed to run the simulator once. For example, with $M = 4$, $K = 12$, $B = 10$, and $C = 0.1$ seconds, the total running time would be 1.67×10^{18} seconds, making this approach computationally infeasible. The core issues here are the *lack of closed-form prediction for tail latency* and the *intractable search space*. To address this challenge, MP^2 incorporates two key components: Gaussian Mixture

[4]The frame rate is not considered part of the QoE function since most VR headsets are running at a constant, pre-configured frame rate.

Modeling (GMM) of frame statistics to provide a closed-form solution for calculating $\boldsymbol{P}$, and pruning and partitioning based on MCS and user locations to exponentially reduce the search space.

Gaussian Mixture Modeling (GMM) of frame statistics: The decision engine should be able to estimate tail latency ratio $\boldsymbol{P}$ for a given candidate arrangement and bitrate guidance $(\boldsymbol{A}, \boldsymbol{B})$. While this could theoretically be achieved with a simulator or a complex prediction model, the exponentially growing search space makes such non-closed-form predictions impractical. Therefore, an efficient yet effective prediction method with a closed-form solution is essential. Our intuition for solving the problem is to approximate the VR frame size statistics using Gaussian approximation. However, since videos usually contain different types of frames such as intra/inter frames with varying size distributions, a single Gaussian distribution may be insufficient. Instead, we adopt GMM [64] which is a mixture of several Gaussian distributions. We use the Expectation-Maximization (EM) algorithm [69] to find parameters in the GMM model and experiment with different numbers of components. The result shown in Figure 5 confirms our approach: the GMM closely tracks the distribution of measured frame size. We further calculate the distance of two distributions using Jensen-Shannon (JS) divergence [57] and experiment with different components of the GMM. From the figure, we conclude that a small number of components such as three, is sufficient to bring the JS distance below 0.10, which indicates accurate modeling.

We then realize a nice property of Gaussian modeling: a linear combination of Gaussian distribution is also Gaussian distributed, which can be demonstrated using convolutions or characteristic functions [16]. Such property helps to *solve two problems at once* of generating closed-form solutions for both $\boldsymbol{L}$ and $\boldsymbol{B}$, eliminating the need to run a simulator. Assuming that there are N users streaming through an AP and the total link bandwidth $\boldsymbol{W}$ (also assume fair time-sharing between Wi-Fi clients), the sum of distribution $\boldsymbol{S}$ would be:

$$\boldsymbol{B} = \alpha \boldsymbol{W} = N * (\sum_{k=1}^{N} L_k^{-1})^{-1}, \quad \boldsymbol{S} \sim \mathcal{N}(\sum_{k=1}^{N} \mu_k, \sum_{k=1}^{N} \sigma_k^2) \quad (2)$$

where α is a discounting variable $(0 < \alpha < 1)$ to trade-off between bitrate and latency. As a result, we can calculate $\boldsymbol{P}$ directly:

$$\boldsymbol{P}_{k,i} = \frac{1}{2}\left[1 - erf\left(\frac{x_i - \sum_{k=1}^{N} \mu_k}{\sqrt{2\sum_{k=1}^{N} \sigma_k^2}}\right)\right] \quad (3)$$

where erf is the Error function and x is the size of frames that take more than 20/50/100 ms to transmit, *e.g.,* $\boldsymbol{W} * 0.1$ for >100 ms latency. Using binary search or Newton's method we can find the best α that maximizes the QoE function.

Pruning and Partitioning: Despite the advantages of GMM, the problem remains challenging as there are still M^K possible arrangements. Our solution is two-fold: *(1) MCS-aware Pruning*: We sort the MCS for all users and apply pruning by skipping a certain percentage p of links with low MCS. The percentage p is adjusted according to computation resources. *(2) Location-based Partitioning*: We group co-located APs into cells. Handover between cells is handled in a way that is adapted from traditional cellular networks: if there exists an AP in another cell with MCS higher than the current AP by a certain threshold, MP^2 calculates the performance gain and initiates the cell switch only if there is significant QoE gain.

With these strategies, we reduce the time complexity to $E * (p * (M/E))^{K/E}$ where E is the number of cells. MP^2 is able to handle system as large as $M = 16$, $K = 48$, $E = 4$, $p = 0.6$ with a running time of less than 1 second. Assuming a fixed cell size with a fixed AP-to-client ratio of $M : K : E$, the complexity of the MP^2 decision algorithm grows linearly as the total number of cells/total users.

In practice, we also observe that link bandwidth fluctuates frequently. Blindly applying the optimal client configuration at each control interval can cause excessive link switching, leading to instability. To address this, MP^2 employs an adaptive topology stabilization mechanism based on a configurable threshold $V_{\text{thresh-qoe}}$, which prevents unnecessary topology changes that offer only marginal benefits but pose significant risks of service disruption. This mechanism acts effectively as a dual-threshold strategy: a handover is triggered only when the estimated global QoE gain exceeds $V_{\text{thresh-qoe}}$, ensuring that topology changes are made only when the benefit is substantial. In scenarios where clients frequently move between neighboring AP cells, this approach prevents oscillation by favoring stability unless a meaningful performance gain is predicted. The threshold is tunable based on user dynamics: a higher value improves stability in volatile scenarios, while a lower one allows more responsive adaptation to user movements. By default, the threshold is set proportional to the number of links being changed in a given attempt ($|A^{\text{cand}} - A^{\text{cur}}|$).

The complete decision algorithm of MP^2 is shown in Algorithm 1. By combining all the optimizations, the control loop takes less than one second to run, which is sufficient for the target scenario.

5 Implementation

It requires non-trivial effort to build a free-roaming VR system as it is relatively new. Most VR streaming systems operate on special hardware with constrained software and OS access, complicating the development. We describe the implementation details of MP^2 and highlight the major challenges we have encountered.

VR streaming platform: The platform should satisfy the following requirements: 1) hardware support for multiple Wi-Fi radios; 2) a software platform that runs MP^2; 3) detailed metrics from VR applications to facilitate evaluation of both the server and client sides. The Oculus ecosystem has the largest user base, but unfortunately, it does not satisfy any of

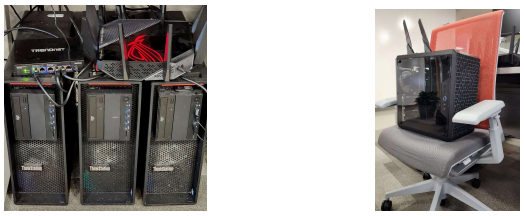
Algorithm 1: The MP^2 Decision Algorithm

Input: Link rates $\boldsymbol{L}$, approximated framesize $\boldsymbol{S}$, current AP selection $\boldsymbol{A}^{cur}$ and bitrate $\boldsymbol{B}^{cur}$.
Output: Arrangement of AP selection $\boldsymbol{A}$, bitrate guidance $\boldsymbol{B}$

```
/* Location-based partitioning                                   */
for each client k do
    if max(L_k) > A_k^cur + V_thresh-link then
        Join(k, argmax(L_k).partition)
for each partition e do
    U^e ← All clients in partition e
    /* Topology stabilization                                    */
    A^e, B^e, QoE^max ←
      A^cur, B^cur, QoE^cur + |A^cand − A^cur| × V_thresh−qoe
    /* MCS-aware pruning                                         */
    L^prune ← prune(argsort(L_k), p) for k in U^e
    candidates ← Cartesian_prod({L_u^pruned | u ∈ U^e})
    for A^cand ∈ candidates do
        QoE^cand ← 0
        for each AP a in partition e do
            U^a ← {u | A_u^cand = a}
            /* GMM-based approx. (S'_u obtained)                 */
            α, QoE^a ← Search towards maximizing:
              Σ_{u∈U^a} QoE(L_u, S'_u, α)
            for u ∈ U^a do
                B_u^cand ← α × (Σ(L_{u∈U^a})^{-1})^{-1}
            QoE^cand ← QoE^cand + QoE^a
        if QoE^cand > QoE^max then
            A^e, B^e, QoE^max ← A^cand, B^cand, QoE^cand
    A.merge(A^e)
    B.merge(B^e)
return A, B
```

these requirements. Instead, we choose to implement MP^2 with ALVR/ALXR [2] as it not only runs on Linux PCs as well as commercial headsets but also comes with source code that allows for the collection of all necessary metrics. On the server side, we have the Steam gaming platform [14] for VR scene rendering and ALVR for VR video streaming. On the client side, we run the headset application using Linux ALXR on top of Monado [8], an open-source XR runtime.

MP^2 tunnel and controller: We implement the MP^2 tunnel in C, following the IETF draft of QUIC [59] with 10k lines of code. It is an event-driven framework that responds to registered events such as read/write file descriptors and timers. We enable multithreading for packet forwarding with dedicated threads for logging and monitoring. The MP^2 system involves orchestrating multiple clients/servers on different layers of the stack. As a result, we choose to build the centralized controller and the VR client/GPU software around Redis [11], as it provides us with a robust, efficient, and scalable method to synchronize messages across devices with versatile interfaces to modules built in different languages. For example, we use



(a) GPU servers and AP/switch. **(b)** Mobile client.

Figure 6: Hardware testbed of the real-world test.

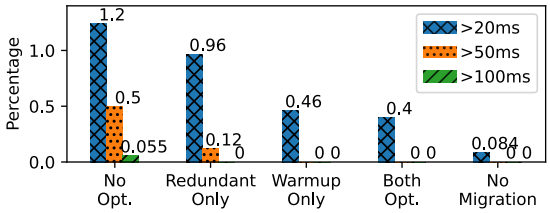


Figure 7: Lag rate comparison of different stream migration methods. MP^2 with both redundant transmission and path warm-up led to significant improvement in the >20 ms lag rate and eliminated >50/100 ms lag completely, which is very close to the upper bound, no migration scenario.

`hiredis` for the C binding with the tunnel, `tokio-rs` for the Rust binding with the ALVR application, and `redis-py` for the Python binding with Wi-Fi information collection and the centralized controller.

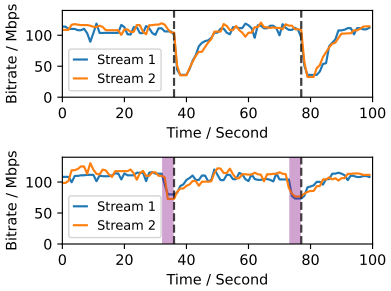
Hardware testbed: The streaming server is equipped with an Intel Xeon 18-core 36-thread CPU and NVIDIA RTX 3070 GPU. The MP^2 tunnel server and the streaming clients are each equipped with an Intel i5 6-core 12-thread CPU. The tunnel server is equipped with one Intel 2*10 Gbps NIC, and one Intel 4*1 Gbps NIC. We connect the tunnel server, and two Asus ROG Rapture GT-AXE11000 Wi-Fi 6 APs via a TEG-S762 six-port, 10 Gbps switch. The streaming clients each connect to the APs with two Intel AX211 Wi-Fi 6E radios and two external antennas, as shown in Figure 6[5].

6 Evaluation

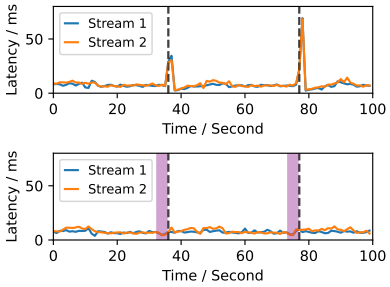
In this section, we present a comprehensive evaluation for MP^2 to answer the following questions:

- **How does MP^2 perform in real-world?** We evaluate the handover performance of MP^2 against state-of-the-art systems in the real world under single/multiuser scenarios.
- **How does MP^2 perform in large-scale emulation?** We perform Mininet emulations that run MP^2 data and control plane to evaluate MP^2 in large-scale deployments that are required to unleash the full potential of MP^2 as well as ensure repeatability and facilitate reproducibility.
- **How does MP^2's performance translate into real players' experience?** We put MP^2 to the ultimate test by conducting

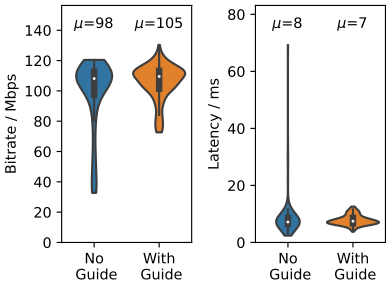
[5]Current mobile platforms offer limited support for dual Wi-Fi radios, so we run clients on standard desktops. They are powered by 100-foot extension cables and are made mobile by placing them on chairs.



(a) Bitrate of two streams without (top) and with (bottom) guidance.



(b) Latency of two streams without (top) and with (bottom) guidance.



(c) Distribution of bitrate (left) and latency (right).

Figure 8: Multi-user handover benchmarks. We indicate the time of migration with vertical dashed lines and color area with bitrate guidance in purple. MP^2 does not incur any noticeable latency increase with a higher average bitrate than XLINK + ALVR which does not have bitrate guidance.

a user study with VR headsets and emulation traces to understand how MP^2 performs from players' perspective.

Baselines. As we have mentioned in §5, it is not easy to build free-roaming VR systems that require setting up both application and transport software on different platforms. We intend to compare as many state-of-the-art solutions as possible, but unfortunately, many works are yet to be open-sourced, which prohibits faithful replication and greatly increases the development effort, or does not support the target platform, or require special hardware. By carefully evaluating recent works, we select the industry standard MPQUIC [32, 60] with four state-of-the-art schedulers as the transport layer for our baselines: minRTT (the default and wide adopted scheduler for MPTCP and MPQUIC) [78], fully redundant (RE) [36], Earliest Completion First (ECF) [56], and XLINK [110] (a recent advancement in MPQUIC optimized specifically for video streaming in mobile environments). The application layer of our baselines is ALVR, and we use high-end commercial Wi-Fi and compute hardware as described in §5. For a fair comparison, we have tailored and optimized these baselines to perform at their best.

6.1 Real-world Performance Test

We benchmark the real-world performance of MP^2 by initially testing with a single user and then introducing additional competing users. Through an ablation study and detailed trace analysis, we closely examine and explain the reasons for the performance improvement.

6.1.1 Handover Latency.

We conduct an ablation study to demonstrate the effectiveness of path warmup and redundant transmission in enhancing path migration latency in a single-user scenario.

In this experiment, we allow a client to move freely between two APs, which triggers a handover every 20 seconds, and record the frame latency for a total duration of 75 minutes. The findings of this study are presented in Figure 7. Compared with XLINK + ALVR (No Opt.), MP^2 with both optimizations could reduce the >20 ms lag rate (percentage of frames with over 20 ms latency) from 1.2% to 0.4% and eliminate >50/100 ms latency completely. The performance of applying both optimizations is the closest to the no migration baseline (achieving minimal latency by sacrificing bitrate) with normal Wi-Fi fluctuation while using a single optimization method leads to worse performance.

6.1.2 Bitrate Guidance.

We further demonstrate the performance of MP^2 in scenarios involving multiple users. We arrange for stream 1 to repeatedly join and depart from the AP with 20-second intervals, while stream 2 remains active. One representative period is depicted in Figure 8. It is important to note that the departure of stream 1 does not cause significant fluctuation. However, when we move stream 2 to the same AP as stream 1, the ABR algorithm responds drastically, reducing the bitrate by more than half and taking over 10 seconds to recover. This is evident in the upper half of Figure 8a.

In contrast, the bitrate guidance method proactively reduces the bitrate, thereby avoiding a drastic drop and recovering to the normal bitrate much quicker. As expected, this preemptive guidance also significantly reduces the latency during the switch, as shown in Figure 8b. This is because the traditional reactive method relies on a delay increase as the signal for bitrate adaptation, which is often too late.

Consequently, bitrate guidance contributes to a 120% increase in the minimum bitrate, from 32.7 to 72.7 Mbps, and a 4.5-fold reduction in the maximum latency, from 68.9 to 12.6 ms. These improvements are illustrated in Figure 8c.

6.2 Larger Scale Emulation

Mininet Wi-Fi Emulation Testbed: Our emulation testbed is based on Mininet Wi-Fi [37], which is a Wi-Fi extension of the network emulator Mininet [51]. It can create isolated containers for each server and mobile client and faithfully emulate the behavior of Wi-Fi. We choose Mininet-based

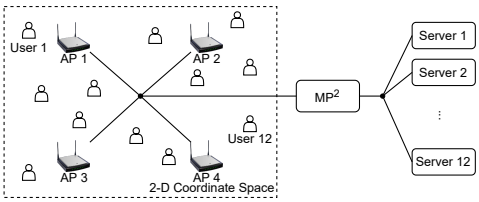


Figure 9: Topology of the large-scale Emulation with Mininet Wi-Fi, using 4 AP × 12 clients as an example.

emulation because MP² is designed as an end-to-end system that tightly integrates real-time components. Mininet allows us to run the MP² software stack without abstracting the core components to evaluate.We overcome two major hurdles to set up the testbed:

(a) VR streams: Due to the limitation of computation resources, it is not possible to start enough VR encoding sessions for the emulation. To maximize emulation fidelity, we record real VR packet traces and replay them using a UDP-based streamer, which implements the same ABR algorithm as ALVR.

(b) Wmediumd: `wmediumd` is the built-in physical layer emulator of Mininet Wi-Fi. Our test results show that its max throughput is less than 30Mbps, which is far away from the aggregated throughput of multiple VR streams. To solve the problem, we approximate a high bandwidth link by scaling the packet count by 40× to balance between performance and practicality that each frame still consists of a few packets. Furthermore, since the implementation of `wmediumd` cannot emulate the effects during handover, we focus solely on emulating the periods between handovers. Although this may understate the full advantages of MP², it will accurately demonstrate the contribution of the MP² controller.

Figure 9 shows an example of the Mininet Wi-Fi deployment featuring 4 APs and 12 clients. In this setup, the APs and their antennas remain stationary. To simulate player movements, we generate a set of random walk trajectories with random speeds between 0 to 3 m/s for each individual player, which. Both the traffic pattern and player movements are kept consistent across comparisons. We carry out the emulation for a cumulative total of 300 hours for all the experiments in this section.

6.2.1 Comprehensive Performance Evaluation.

In this section, we carry out an extensive and thorough performance evaluation involving 16 APs and 48 clients. The results are displayed in Figure 10. MP² consistently reduced latency throughout the tests, with only 0.26% of the time where latency exceeded 20 ms outperforming all baseline metrics by a large margin. It achieves an impressive reduction in the 20 ms lag rate by 97.3% to 98.9%. Moreover, it completely eliminates cases where the latency exceeds 50 ms.

MPQUIC + ALVR with RE scheduler exhibits the weakest performance, primarily due to its design of full redundancy. This design, while intended for occasional severe network disruptions, is not well-suited for multi-user scenarios. The

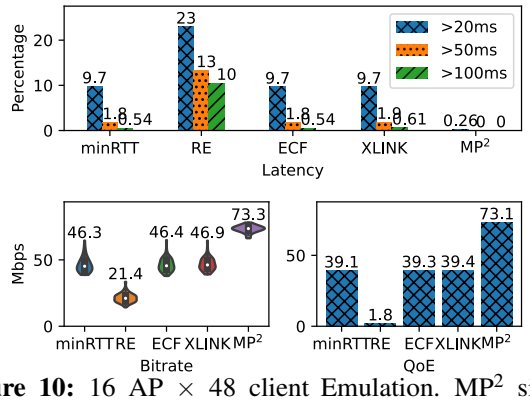


Figure 10: 16 AP × 48 client Emulation. MP² significantly outperforms XLINK + ALVR and different flavors of MPQUIC (MinRTT, RE, ECF) + ALVR on both latency (35× improvement over 2nd place), bitrate (1.56×), and QoE (1.86×).

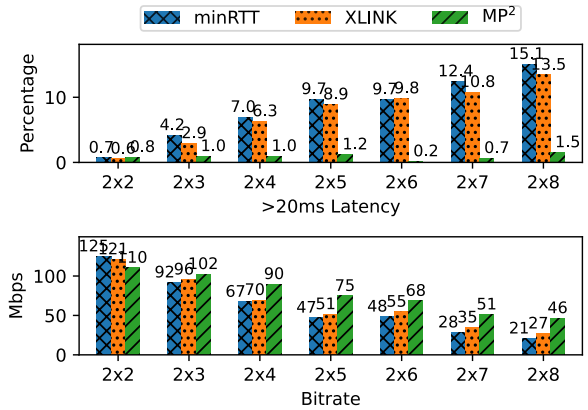


Figure 11: Benchmark of MPQUIC (minRTT) + ALVR, XLINK + ALVR and MP² under different AP×user ratio. MP² is able to maintain a steady low latency while baselines keep increasing as there are more users while keeping a high bitrate with up to 8 users.

unnecessary redundant packets congest the system, leading to an increased queue buildup and consequently compromising QoE for all players. XLINK and other MPQUIC schedulers including MinRTT, and ECF display similar latency patterns.

MP² achieves an average bitrate of 73.3 Mbps, notably improving the overall network capacity by 56.3% to 242.5%. We adopt a sample QoE with latency penalty weights $w_1, w_2, w_3 = 1, 2, 4$ in this experiment to show consideration of the combined effect of latency and bitrate. It is not surprising that MP² also achieves outstanding performance in terms of QoE, which is 1.86× the closest contender.

6.2.2 Scalability Evaluation.

In this experiment, we compare MP² with other competitors under different user/AP densities and scales to show its scalability advantages. Figure 11 shows the performance variation in a 2-AP setup where the number of users grows from 2 to 8. In 2 × 2 deployment, MP² performs similarly to its competitors. This is expected since most multipath schedulers are naturally optimized for such cases. When the user den-

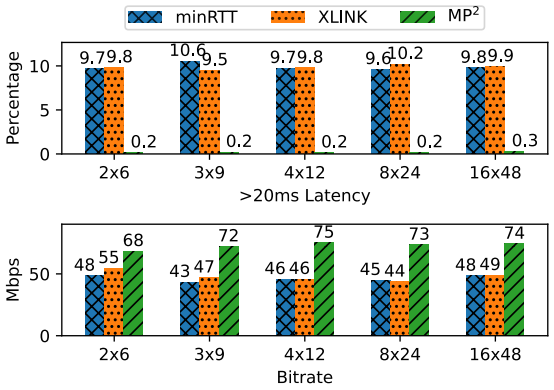


Figure 12: Benchmark of MPQUIC (minRTT) + ALVR, XLINK + ALVR, and MP^2 with the same AP to client ratio but different scales. MP^2 consistently outperforms baselines.

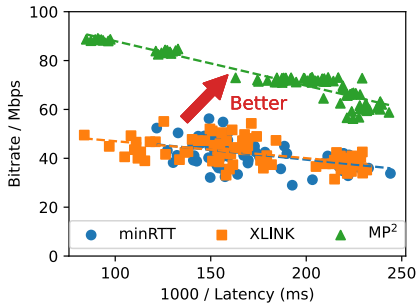


Figure 13: Latency *vs.* bitrate under different ABR algorithm targets. MP^2 achieves a clearly better trade-off than minRTT + ALVR and XLINK + ALVR.

sity grows, as expected, both latency and bitrate performance of both MPQUIC (minRTT) + ALVR and XLINK + ALVR degrade drastically due to unmannered competing for bandwidth. On the other hand, MP^2 successfully keeps the >20 ms lag rate below 1.5%, achieving up to 48× lag rate improvement over the 2nd place. In terms of bitrate, MP^{2}s able to maximize the utility of limited bandwidth and achieves 1.7 × improvement over the 2nd place.

Figure 12 illustrates the variations in latency and bitrate across different scales ranging from 2×6 to 16×48. Overall, MP^2 consistently delivers a substantial performance boost irrespective of the scale. The lag rate exceeding 20 ms remains below 0.6% across all scales, while the bitrate improvement ranges from 51.0% to 71.4% over the baseline. The performance of MP^2 remains relatively stable when the system scale grows, demonstrating its capability for VR streaming with a larger number of users.

6.2.3 Evaluation under Different ABR Latency Targets.

MP^2 is designed to cooperate with ABR algorithms, enabling flexible adjustments to meet various latency objectives. To understand its performance, we conducted experiments with latency target settings of 5/10/20 ms and compared the results with minRTT/XLINK + ALVR baselines. The experiments are performed 70 times in a 4 AP × 12 client setup. Figure 13 clearly illustrates the trade-off between median latency and bitrate for each of the three ABR targets. MP^2 ranges from 4 ms/61 Mbps to 12 ms/92 Mbps while minRTT and XLINK ranges from 4 ms/38 Mbps to 12 ms/50 Mbps. Regardless of the latency target, MP^2 consistently achieves a better Pareto frontier between latency and bitrate.

(a) Sample VR scenes (indoor and outdoor). **(b)** Oculus Pro VR headset and controllers.

Figure 14: Software/hardware setup of the user study.

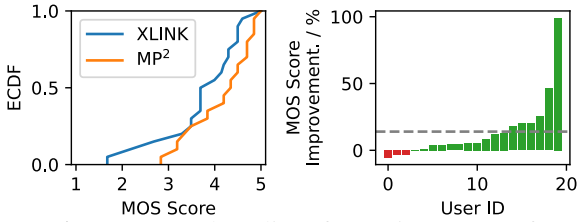


Figure 15: Perceptual quality of experience score for MP^2 and XLINK + ALVR, the higher the better. MP^2 consistently outperforms the baseline and is significantly preferred by the vast majority of users.

6.3 Trace-driven User Study

A fundamental limitation of current commercial VR headsets is their lack of support for additional network adapters, in both hardware and software, which constrains the evaluation of MP^2. In order to assess the benefits MP^2 could bring to the VR experience from the viewpoint of the players, we conduct a user study which is based on packet-level traces collected from the previous large-scale emulation, by randomly selecting client-side traces from both XLINK + ALVR and MP^2.

System Setup: The setup is shown in Figure 14. We construct the user study platform using the ALVR backend, the Steam VR gaming engine, and the Oculus Quest Pro, a high-end consumer VR headset. To emulate network impact, we use MahiMahi [71] to replay the collected packet-level traces between the headset and a container running Steam VR and the ALVR streamer on a GPU server.

Methodology: We recruit 20 unpaid volunteers, ranging in age from early 20s to 40s, with varied technical and VR experience. We distribute invitations for the user study via a public channel within our organization, and include all interested individuals in the experiment and results. Each participant is exposed to both the MP2 and XLINK + ALVR conditions in randomized order, across visually distinct indoor and outdoor scenes. In each setting, participants freely explore three

30-second segments per condition and rate their overall VR experience on a 1–5 scale (bad to excellent), yielding a total of 240 ratings. The evaluation is based on the question: "How would you rate your overall VR experience?" While the scale labels are explained qualitatively, no formal training or dimension-specific criteria are provided.

Figure 15 shows the distribution and improvements of Mean Opinion Scores (MOS). MP^2 consistently outperforms the baseline in terms of MOS distribution. Additionally, MP^2 is preferred by the vast majority of users, with an improvement of up to 99.1%. Even among users who do not perceive an improvement, MP^2 matches the baseline performance, with no more than 5.5% drawbacks.

7 Limitations and Discussion

Next, we discuss MP^2's limitations and future directions.

7.1 Practical Concerns

Larger scale deployment: Due to hardware (no support for multiple Wi-Fi interfaces) and software (unable to run a customized tunnel application) constraints (§5), in the real-world experiments, the client-side testbed for MP^2 is implemented on PC rather than actual headsets. These constraints are largely due to the relatively closed nature of current VR headset ecosystems. We look forward to a large-scale deployment with actual VR headsets. In fact, we anticipate that MP^2 can be easily deployed on any headset by simply connecting an extra Wi-Fi NIC and installing a user-plane application.

Hardware Gap: The only hardware gap between MP^2 and current commercial VR systems is the addition of a second Wi-Fi interface (with the corresponding antenna). However, we argue that this is practical: (1) Multiple Wi-Fi interfaces have long been commercially available on the AP side [22], and there are no technical difficulties on the client side either. In fact, the industry [5, 6, 9] is moving towards Dual Band Dual Concurrent (DBDC) Wi-Fi, which enables simultaneous operation on different bands; (2) Wi-Fi radios are relatively inexpensive compared to the cost of VR headsets, with entry-level models available for as little as USD $2 [10], and even advanced Wi-Fi radios can be obtained for $10 to $20 [3, 4].

Additional Power Consumption: While the second Wi-Fi interface introduces some power overhead, our estimation shows it is minimal, contributing less than 2% of the total headset power consumption. The interface remains idle most of the time (for PHY info collection) and activates only during migration periods, which occur for at most 100 ms per second (10% duty cycle). According to open measurements [68], it consumes up to 750 mW when active and 25 mW when idle. Based on this, the weighted average overhead is $\leq$97.5 mW. Given that typical VR headsets draw 5–13 W during operation [66, 67], the additional consumption remains within 0.75%–2%.

Scalability Limits: Scalability is an important consideration in MP^2's centralized architecture. To address this, MP^2 adopts a partitioning strategy (§4.2) in which the environment is divided into geographic cells. This reflects the observation that resource contention in multiplayer VR primarily occurs among users who are physically close. By optimizing QoE within each cell independently, global optimization is reduced to a set of localized sub-problems. Inter-cell handovers are handled using lightweight heuristics inspired by cellular network designs. As a result, the computational overhead of the MP^2 Controller scales linearly with the number of users and APs. Nonetheless, scalability within a single partition is bounded by the CPU and NIC capacity of the MP^2 Hub. Our prototype demonstrates support for up to 48 users per partition using a single server. Beyond this scale, the hub may become a bottleneck. While partitioning enhances scalability, it can weaken global optimality. Thus, partition size should be carefully selected based on server capability, or alternatively, multiple servers can be deployed to scale out the MP^2 Hub horizontally, managed by containerized cloud [28, 98, 99].

7.2 Future Works

Handling Failures: As a centralized system, MP^2 is built with failure resilience in mind. In the event of a data plane failure, such as a tunnel disconnection, clients will immediately fall back to a direct connection with VR servers. While AP and bitrate guidance remain functional, handovers are no longer seamless under this fallback mode. To detect control plane failures, clients periodically send heartbeat packets, allowing the system to revert to a baseline solution when needed. Partial failures may result in some clients/servers being unresponsive to AP association and bitrate guidance, leading to reduced performance for unaffected clients. However, addressing these failure scenarios comprehensively requires careful design and rigorous testing.

Full redundancy and network coding: MP^2 does not replicate streams unless during migration for better efficiency as we observe there is nontrivial overhead to do so, which may result in much worse performance, such as RE mode of MPQUIC in §6. This result aligns with previous works [81]. However, if bandwidth permits, network coding [38, 44, 48] based redundant transmission through two or more links may offer higher flexibility, especially for less stable links.

Experiment with other ABR algorithms: although MP^2 is designed to work with any ABR algorithm due to the fact that we impose a cap for the bitrate only, it may be interesting to see how it works with state-of-the-art real-time ABR algorithms [25, 34].

8 Related Work

While we have presented a comparison of MP^2 with related works in Table 1, we delve into them with greater detail:

Reducing latency during handover. Latency experienced during handover between APs represents a significant challenge [74] that attracted substantial attention: SCTP [86] dynamically updates a user's address list for handover from one interface to another. MPTCP [74, 78, 95] and MPQUIC [32, 94], multipath extensions of TCP and QUIC, can adapt to changing network conditions and provide smooth handover leveraging multiple channels. Efforts have been made to improve multipath transport [40, 55, 72, 73] and apply multipath for various cases, such as path dynamics, heterogeneity, and handovers [43, 53, 77, 110, 111]. These systems primarily focus on single-user perspectives while MP^2 targets a group of VR users sharing the common bottleneck and optimizes for overall performance. There are also multipath solutions designed for Wi-Fi networks. Sheriff *et al.* [83], Musher [81], MPWi-Fi [75] and DiversiFi [47] aim to exploit multiple Wi-Fi paths simultaneously for improved throughput. Croitoru *et al.* [29] connects mobile clients with all visible APs and splits traffic among them. Again, they do not address multiple contending users. Our approach optimizes global QoE for a group of users with coordination among different user streams.

Streaming optimizations for mobile VR. Collaborative VR is a real-time streaming application without an explicit playback buffer and thus cannot utilize existing buffer-aware ABR algorithms on video-on-demand (VoD) tasks [45, 63, 70, 100, 104]. Adaptive real-time streaming solutions like GCC [25] and Converge [34] optimize both latency and bitrate, but their applications (*e.g.,* video conferencing) do not involve user competition at the edges. Nevertheless, they are orthogonal and can be integrated with MP^2. Given the positional and link-quality dynamics of mobile VR users, VR-specific ABR algorithms and streaming strategies like Firefly [58], MoVR [19], M5 [107], Habitus [105], and CollaborativeVR [26] have been developed. However, they either focus on single-user scenarios, potentially compromising other users' performance, or are limited to single-AP scenarios that are unsuitable for free-roaming VR. Recent advancements including MuV2 [61], Dasari *et al.* [31] leverage the correlation across VR users to optimize rendering, which are also orthogonal and can be integrated with MP^2.

Multi-AP coordination for load-balancing. Multi-AP coordination techniques like C-OFDMA [93], coordinated spatial reuse [62], coordinated beamforming [20, 42], and joint transmission [106] have been explored. There is also a growing interest in using a centralized controller for WLAN management such as Smartassoc [97], NEMOx [109], SWAN [54], OpenSDWN [82], MUSE [87], BigAP [113], Wi-Fi Goes to Town [84], ClientMarshal [23], LATTE [76], LMA-ABC [96], and OpenCarrier [101]. However, these approaches often require either low-level access to Wi-Fi devices or customized hardware. Additionally, they may fail to meet the stringent demands for handover latency and bandwidth required by free-roaming VR.

9 Conclusion

We design, implement, and evaluate MP^2, a centralized overlay system with a global view of the entire stack that coordinates link/path/bitrate decisions across free-roaming VR users. With a carefully engineered data plane and an intelligent control plane, MP^2 orchestrates VR streams that compete for bandwidth to achieve global optimal QoE and steer traffic between paths seamlessly. Our real-world tests, large-scale simulations, and trace-driven user studies confirm the effectiveness of MP^2, with up to an order of magnitude improvement in latency metrics, as well as consistent QoE gain under different settings, making it a promising solution for future immersive VR experiences.

References

[1] 802.11ac-vht mcs, snr and rssi. https://d2cpnw0u24fjm4.cloudfront.net/wp-content/uploads/802.11ac-VHT-MCS-SNR-and-RSSI.pdf. Accessed: 2024-02-29.

[2] Alvr. https://alvr-org.github.io/. Accessed: 2023-09-01.

[3] Ax201ngw 2.4gbps 802.11ax wireless bluetooth 5.0 adapter. https://www.amazon.com/Wendry-Wireless-M-2-CNVio2-Interface-Bluetooth/dp/B083WFL2W3. Accessed: 2023-09-01.

[4] Intel ax210 ieee 802.11ax bluetooth 5.2 tri band wi-fi/bluetooth combo adapter. https://www.amazon.com/Intel-802-11ax-Blue-tooth-Adapter-Notebook/dp/B0B4T696W5. Accessed: 2023-09-01.

[5] Intel wi-fi 6e ax411. https://www.intel.com/content/www/us/en/products/sku/217242/intel-wifi-6e-ax411-gig/specifications.html. Accessed: 2023-09-01.

[6] Intel® killer™ wi-fi 6e ax1690. https://www.intel.com/content/www/us/en/products/sku/217241/intel-killer-wifi-6e-ax1690-is/specifications.html. Accessed: 2023-09-01.

[7] Mcs index, modulation and coding. https://mcsindex.net/. Accessed: 2023-09-01.

[8] Monado - open source xr platform. https://monado.dev/. Accessed: 2023-09-01.

[9] Qualcomm unleashes wi-fi gaming performance for windows 11 pcs. https://www.qualcomm.com/news/releases/2021/10/qualcomm-unleashes-wi-fi-gaming-performance-windows-11-pcs. Accessed: 2023-09-01.

[10] Realtek mini usb wireless 802.11b/g/n lan card wifi network adapter rtl8188. https://www.ebay.com/itm/284649544751. Accessed: 2023-09-01.

[11] Redis. https://redis.io/. Accessed: 2023-09-01.

[12] Sandbox vr. https://sandboxvr.com/. Accessed: 2023-09-01.

[13] Spree arena. https://jointhespree.com/solutions/spree-arena-vr-attraction/. Accessed: 2023-09-01.

[14] Steam. https://steampowered.com/. Accessed: 2023-09-01.

[15] Strivr enterprise vr training. https://www.strivr.com/. Accessed: 2023-09-01.

[16] Sum of normally distributed random variables. https://en.wikipedia.org/wiki/Sum_of_normally_distributed_random_variables. Accessed: 2023-09-01.

[17] The void vr. https://www.thevoid.com/. Accessed: 2023-09-01.

[18] Zero latency vr. https://zerolatencyvr.com/. Accessed: 2023-09-01.

[19] Omid Abari, Dinesh Bharadia, Austin Duffield, and Dina Katabi. Enabling high-quality untethered virtual reality. In *USENIX NSDI*, 2017.

[20] Omid Abari, Hariharan Rahul, Dina Katabi, and Mondira Pant. Airshare: Distributed coherent transmission made seamless. In *2015 IEEE Conference on Computer Communications (INFOCOM)*, pages 1742–1750. IEEE, 2015.

[21] Hassan Ahmed and Hossam Hassanein. A performance study of roaming in wireless local area networks based on ieee 802.11 r. In *2008 24th Biennial Symposium on Communications*. IEEE, 2008.

[22] ASUS ROG. Rog rapture gt-axe11000. https://rog.asus.com/networking/rog-rapture-gt-axe11000-model/. Accessed: 2025-05-20.

[23] Apurv Bhartia, Bo Chen, Derrick Pallas, and Waldin Stone. Clientmarshal: Regaining control from wireless clients for better experience. In *The 25th Annual International Conference on Mobile Computing and Networking*, pages 1–16, 2019.

[24] Divyashri Bhat, Amr Rizk, and Michael Zink. Not so quic: A performance study of dash over quic. In *Proceedings of the 27th workshop on network and operating systems support for digital audio and video*, pages 13–18, 2017.

[25] Gaetano Carlucci, Luca De Cicco, Stefan Holmer, and Saverio Mascolo. Analysis and design of the google congestion control for web real-time communication (webrtc). In *ACM MMSys*, 2016.

[26] Jiangong Chen, Feng Qian, and Bin Li. Enhancing quality of experience for collaborative virtual reality with commodity mobile devices. In *2022 IEEE 42nd International Conference on Distributed Computing Systems (ICDCS)*, pages 1018–1028. IEEE, 2022.

[27] Ying Chen, Hojung Kwon, Hazer Inaltekin, and Maria Gorlatova. Vr viewport pose model for quantifying and exploiting frame correlations. In *IEEE INFOCOM*, 2022.

[28] Yuning Chen, Kang Yang, Zhiyu An, Brady Holder, Luke Paloutzian, Khaled M Bali, and Wan Du. Marlp: Time-series forecasting control for agricultural managed aquifer recharge. In *ACM KDD*, 2024.

[29] Andrei Croitoru, Dragos Niculescu, and Costin Raiciu. Towards wifi mobility without fast handover. In *USENIX NSDI*, 2015.

[30] Eduardo Cuervo, Krishna Chintalapudi, and Manikanta Kotaru. Creating the perfect illusion: What will it take to create life-like virtual reality headsets? In *Proceedings of the 19th International Workshop on Mobile Computing Systems & Applications*, 2018.

[31] Mallesham Dasari, Edward Lu, Michael W Farb, Nuno Pereira, Ivan Liang, and Anthony Rowe. Scaling vr video conferencing. In *2023 IEEE Conference Virtual Reality and 3D User Interfaces (VR)*, pages 648–657. IEEE, 2023.

[32] Quentin De Coninck and Olivier Bonaventure. Multipath quic: Design and evaluation. In *ACM CoNEXT*, 2017.

[33] Quentin De Coninck and Olivier Bonaventure. Multiflow quic: a generic multipath transport protocol. *IEEE Communications Magazine*, 59(5):108–113, 2021.

[34] Sandesh Dhawaskar Sathyanarayana, Kyunghan Lee, Dirk Grunwald, and Sangtae Ha. Converge: Qoe-driven multipath video conferencing over webrtc. In *ACM SIGCOMM*, 2023.

[35] Stefan Feirer and Thilo Sauter. Seamless handover in industrial wlan using ieee 802.11 k. In *2017 IEEE 26th international symposium on industrial electronics (ISIE)*, pages 1234–1239. IEEE, 2017.

[36] Tobias Flach, Nandita Dukkipati, Andreas Terzis, Barath Raghavan, Neal Cardwell, Yuchung Cheng, Ankur Jain, Shuai Hao, Ethan Katz-Bassett, and Ramesh Govindan. Reducing web latency: the virtue of gentle aggression. In *ACM SIGCOMM*, 2013.

[37] Ramon R Fontes, Samira Afzal, Samuel HB Brito, Mateus AS Santos, and Christian Esteve Rothenberg. Mininet-wifi: Emulating software-defined wireless networks. In *2015 11th International Conference on Network and Service Management (CNSM)*. IEEE, 2015.

[38] Christos Gkantsidis and Pablo Rodriguez Rodriguez. Network coding for large scale content distribution. In *Proceedings IEEE 24th Annual Joint Conference of the IEEE Computer and Communications Societies.*, volume 4, pages 2235–2245. IEEE, 2005.

[39] Francesco Gringoli, Matthias Schulz, Jakob Link, and Matthias Hollick. Free your csi: A channel state information extraction platform for modern wi-fi chipsets. In *Proceedings of the 13th International Workshop on Wireless Network Testbeds, Experimental Evaluation & Characterization*, pages 21–28, 2019.

[40] Yihua Ethan Guo, Ashkan Nikravesh, Z Morley Mao, Feng Qian, and Subhabrata Sen. Accelerating multipath transport through balanced subflow completion. In *ACM MobiCom*, 2017.

[41] Habtegebreil Haile, Karl-Johan Grinnemo, Simone Ferlin, Per Hurtig, and Anna Brunstrom. Performance of quic congestion control algorithms in 5g networks. In *Proceedings of the ACM SIGCOMM Workshop on 5G and Beyond Network Measurements, Modeling, and Use Cases*, pages 15–21, 2022.

[42] Ezzeldin Hamed, Hariharan Rahul, Mohammed A Abdelghany, and Dina Katabi. Real-time distributed mimo systems. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 412–425, 2016.

[43] Bo Han, Feng Qian, Lusheng Ji, and Vijay Gopalakrishnan. Mp-dash: Adaptive video streaming over preference-aware multipath. In *ACM CoNEXT*, 2016.

[44] Tracey Ho, Muriel Médard, Ralf Koetter, David R Karger, Michelle Effros, Jun Shi, and Ben Leong. A random linear network coding approach to multicast. *IEEE Transactions on information theory*, 52(10):4413–4430, 2006.

[45] Te-Yuan Huang, Ramesh Johari, Nick McKeown, Matthew Trunnell, and Mark Watson. A buffer-based approach to rate adaptation: Evidence from a large video streaming service. In *Proceedings of the 2014 ACM conference on SIGCOMM*, pages 187–198, 2014.

[46] Arash Molavi Kakhki, Samuel Jero, David Choffnes, Cristina Nita-Rotaru, and Alan Mislove. Taking a long look at quic: an approach for rigorous evaluation of rapidly evolving transport protocols. In *ACM IMC*, 2017.

[47] Rajat Kateja, Nimantha Baranasuriya, Vishnu Navda, and Venkata N Padmanabhan. Diversifi: Robust multilink interactive streaming. In *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies*, 2015.

[48] Sachin Katti, Hariharan Rahul, Wenjun Hu, Dina Katabi, Muriel Médard, and Jon Crowcroft. Xors in the air: Practical wireless network coding. In *Proceedings of the 2006 conference on Applications, technologies, architectures, and protocols for computer communications*, 2006.

[49] H Koumaras, C Skianis, G Gardikis, and A Kourtis. Analysis of h. 264 video encoded traffic. In *Proceedings of the 5th International Network Conference (INC2005)*, pages 441–448, 2005.

[50] Jonathan Kua, Grenville Armitage, and Philip Branch. A survey of rate adaptation techniques for dynamic adaptive streaming over http. *IEEE Communications Surveys & Tutorials*, 19(3):1842–1866, 2017.

[51] Bob Lantz, Brandon Heller, and Nick McKeown. A network in a laptop: rapid prototyping for software-defined networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, 2010.

[52] Steven M LaValle, Anna Yershova, Max Katsev, and Michael Antonov. Head tracking for the oculus rift. In *IEEE ICRA*, 2014.

[53] HyunJong Lee, Jason Flinn, and Basavaraj Tonshal. Raven: Improving interactive latency for the connected car. In *ACM MobiCom*, 2018.

[54] Tao Lei, Zhaoming Lu, Xiangming Wen, Xing Zhao, and Luhan Wang. Swan: An sdn based campus wlan framework. In *2014 4th International Conference on Wireless Communications, Vehicular Technology, Information Theory and Aerospace & Electronic Systems (VITAE)*, pages 1–5. IEEE, 2014.

[55] Yeon-sup Lim, Yung-Chih Chen, Erich M Nahum, Don Towsley, Richard J Gibbens, and Emmanuel Cecchet. Design, implementation, and evaluation of energy-aware multi-path tcp. In *ACM CoNEXT*, 2015.

[56] Yeon-sup Lim, Erich M Nahum, Don Towsley, and Richard J Gibbens. Ecf: An mptcp path scheduler to manage heterogeneous paths. In *Proceedings of the 13th international conference on emerging networking experiments and technologies*, pages 147–159, 2017.

[57] Jianhua Lin. Divergence measures based on the shannon entropy. *IEEE Transactions on Information theory*, 37(1):145–151, 1991.

[58] Xing Liu, Christina Vlachou, Feng Qian, Chendong Wang, and Kyu-Han Kim. Firefly: Untethered multi-user vr for commodity mobile devices. In *Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference*, pages 943–657, 2020.

[59] Yanmei Liu, Yunfei Ma, Quentin De Coninck, Olivier Bonaventure, Christian Huitema, and Mirja Kühlewind. Multipath Extension for QUIC. Internet-Draft draft-ietf-quic-multipath-02, Internet Engineering Task Force, July 2022. Work in Progress.

[60] Yanmei Liu, Yunfei Ma, Quentin De Coninck, Olivier Bonaventure, Christian Huitema, and Mirja Kühlewind. Multipath Extension for QUIC. Internet-Draft draft-ietf-quic-multipath-10, Internet Engineering Task Force, July 2024. Work in Progress.

[61] Yu Liu, Puqi Zhou, Zejun Zhang, Anlan Zhang, Bo Han, Zhenhua Li, and Feng Qian. Muv2: Scaling up multi-user mobile volumetric video streaming via content hybridization and sharing. In *ACM MobiCom*, 2024.

[62] David López-Pérez, Xiaoli Chu, Athanasios V Vasilakos, and Holger Claussen. On distributed and coordinated resource allocation for interference mitigation in self-organizing lte networks. *IEEE/ACM Transactions on Networking*, 21(4):1145–1158, 2012.

[63] Hongzi Mao, Ravi Netravali, and Mohammad Alizadeh. Neural adaptive video streaming with pensieve. In *Proceedings of the conference of the ACM special interest group on data communication*, pages 197–210, 2017.

[64] Geoffrey J McLachlan, Sharon X Lee, and Suren I Rathnayake. Finite mixture models. *Annual review of statistics and its application*, 6:355–378, 2019.

[65] Péter Megyesi, Zsolt Krämer, and Sándor Molnár. How quick is quic? In *IEEE ICC*, 2016.

[66] Meta. Meta quest 3 — specs. https://www.meta.com/ie/quest/quest-3/#specs. Accessed: 2025-04-09.

[67] Meta. Product information sheet – meta quest 3. https://www.meta.com/ie/legal/quest/product-information-sheet-meta-quest-3/. Accessed: 2025-04-09.

[68] Microsoft. Wi-fi power management for modern standby platforms. https://learn.microsoft.com/en-us/windows-hardware/design/device-experiences/wi-fi-power-management-for-modern-standby-platforms. Accessed: 2025-04-09.

[69] Todd K Moon. The expectation-maximization algorithm. *IEEE Signal processing magazine*, 13(6):47–60, 1996.

[70] Vikram Nathan, Vibhaalakshmi Sivaraman, Ravichandra Addanki, Mehrdad Khani, Prateesh Goyal, and Mohammad Alizadeh. End-to-end transport for video qoe fairness. In *Proceedings of the ACM Special Interest Group on Data Communication*, pages 408–423. 2019.

[71] Ravi Netravali, Anirudh Sivaraman, Somak Das, Ameesh Goyal, Keith Winstein, James Mickens, and Hari Balakrishnan. Mahimahi: accurate {Record-and-Replay} for {HTTP}. In *USENIX ATC*, 2015.

[72] Yunzhe Ni, Feng Qian, Taide Liu, Yihua Cheng, Zhiyao Ma, Jing Wang, Zhongfeng Wang, Gang Huang, Xuanzhe Liu, and Chenren Xu. {POLYCORN}: Data-driven cross-layer multipath networking for high-speed railway through composable schedulerlets. In *USENIX NSDI*, 2023.

[73] Ashkan Nikravesh, Yihua Guo, Feng Qian, Z Morley Mao, and Subhabrata Sen. An in-depth understanding of multipath tcp on mobile devices: Measurement and system design. In *ACM MobiCom*, 2016.

[74] Christoph Paasch, Gregory Detal, Fabien Duchene, Costin Raiciu, and Olivier Bonaventure. Exploring mobile/wifi handover with multipath tcp. In *Proceedings of the 2012 ACM SIGCOMM workshop on Cellular networks: operations, challenges, and future design*, pages 31–36, 2012.

[75] Mijanur Rahaman Palash and Kang Chen. Mpwifi: Synergizing mptcp based simultaneous multipath access and wifi network performance. *IEEE Transactions on Mobile Computing*, 19(1):142–158, 2018.

[76] Hannaneh Barahouei Pasandi, Tamer Nadeem, Hadi Amirpour, and Christian Timmerer. A cross-layer approach for supporting real-time multi-user video streaming over wlans. In *Proceedings of the 27th Annual International Conference on Mobile Computing and Networking*, pages 849–851, 2021.

[77] Costin Raiciu, Sebastien Barre, Christopher Pluntke, Adam Greenhalgh, Damon Wischik, and Mark Handley. Improving datacenter performance and robustness with multipath tcp. *ACM SIGCOMM Computer Communication Review*, 41(4):266–277, 2011.

[78] Costin Raiciu, Christoph Paasch, Sebastien Barre, Alan Ford, Michio Honda, Fabien Duchene, Olivier Bonaventure, and Mark Handley. How hard can it be? designing and implementing a deployable multipath {TCP}. In *9th USENIX symposium on networked systems design and implementation (NSDI 12)*, pages 399–412, 2012.

[79] Jan Rüth, Ingmar Poese, Christoph Dietzel, and Oliver Hohlfeld. A first look at quic in the wild. In *International Conference on Passive and Active Network Measurement*, pages 255–268. Springer, 2018.

[80] Jan Rüth, Konrad Wolsing, Klaus Wehrle, and Oliver Hohlfeld. Perceiving quic: Do users notice or even care? In *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies*, 2019.

[81] Swetank Kumar Saha, Shivang Aggarwal, Rohan Pathak, Dimitrios Koutsonikolas, and Joerg Widmer. Musher: An agile multipath-tcp scheduler for dual-band 802.11 ad/ac wireless lans. In *The 25th Annual International Conference on Mobile Computing and Networking*, pages 1–16, 2019.

[82] Julius Schulz-Zander, Carlos Mayer, Bogdan Ciobotaru, Stefan Schmid, and Anja Feldmann. Opensdwn: Programmatic control over home and enterprise wifi. In *Proceedings of the 1st ACM SIGCOMM symposium on software defined networking research*, pages 1–12, 2015.

[83] Irfan Sheriff and Elizabeth Belding-Royer. Multipath selection in multi-radio mesh networks. In *2006 3rd International Conference on Broadband Communications, Networks and Systems*, pages 1–11. IEEE, 2006.

[84] Zhenyu Song, Longfei Shangguan, and Kyle Jamieson. Wi-fi goes to town: Rapid picocell switching for wireless transit networks. In *ACM SIGCOMM*, 2017.

[85] Hamed Soroush, Peter Gilbert, Nilanjan Banerjee, Brian Neil Levine, Mark Corner, and Landon Cox. Concurrent wi-fi for mobile users: analysis and measurements. In *Proceedings of the Seventh COnference on emerging Networking EXperiments and Technologies*, pages 1–12, 2011.

[86] Randall Stewart, Qiaobing Xie, Michael Tuexen, Shin Maruyama, and Masahiro Kozuka. Stream control transmission protocol (sctp) dynamic address reconfiguration. Technical report, 2007.

[87] Sanjib Sur, Ioannis Pefkianakis, Xinyu Zhang, and Kyu-Han Kim. Practical mu-mimo user selection on 802.11 ac commodity networks. In *Proceedings of the 22nd Annual International Conference on Mobile Computing and Networking*, pages 122–134, 2016.

[88] Ahmad Ali Tabassam, Henning Trsek, Stefan Heiss, and Jürgen Jasperneite. Fast and seamless handover for secure mobile industrial applications with 802.11 r. In *2009 IEEE 34th Conference on Local Computer Networks*. IEEE, 2009.

[89] Zhaowei Tan, Yuanjie Li, Qianru Li, Zhehui Zhang, Zhehan Li, and Songwu Lu. Supporting mobile vr in lte networks: How close are we? *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 2(1):1–31, 2018.

[90] Zhaowei Tan, Jinghao Zhao, Yuanjie Li, Yifei Xu, Yunqi Guo, and Songwu Lu. Ldrp: Device-centric latency diagnostic and reduction for cellular networks without root. *IEEE Transactions on Mobile Computing*, 23(4):2748–2764, 2023.

[91] Zhaowei Tan, Jinghao Zhao, Yuanjie Li, Yifei Xu, and Songwu Lu. {Device-Based}{LTE} latency reduction at the application layer. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 471–486, 2021.

[92] Farzad Tashtarian, Abdelhak Bentaleb, Hadi Amirpour, Sergey Gorinsky, Junchen Jiang, Hermann Hellwagner, Christian Timmerer, et al. Artemis: Adaptive bitrate ladder optimization for live video streaming. In *USENIX Symposium on Networked Systems Design and Implementation*, pages 1–21, 2024.

[93] Luca Venturino, Alessio Zappone, Chiara Risi, and Stefano Buzzi. Energy-efficient scheduling and power allocation in downlink ofdma networks with base station coordination. *IEEE transactions on wireless communications*, 14(1):1–14, 2014.

[94] Tobias Viernickel, Alexander Froemmgen, Amr Rizk, Boris Koldehofe, and Ralf Steinmetz. Multipath quic: A deployable multipath transport protocol. In *IEEE ICC*, 2018.

[95] Damon Wischik, Costin Raiciu, Adam Greenhalgh, and Mark Handley. Design, implementation and evaluation of congestion control for multipath tcp. In *NSDI*, 2011.

[96] Huaqing Wu, Jiayin Chen, Conghao Zhou, Feng Lyu, Ning Zhang, Li Wang, and Xuemin Sherman Shen. Load-and mobility-aware cooperative content delivery in sag integrated vehicular networks. In *ICC 2021-IEEE International Conference on Communications*, pages 1–6. IEEE, 2021.

[97] Fengyuan Xu, Xiaojun Zhu, Chiu C Tan, Qun Li, Guanhua Yan, and Jie Wu. Smartassoc: Decentralized access point selection algorithm to improve throughput. *IEEE transactions on Parallel and distributed systems*, 24(12):2482–2491, 2013.

[98] Yifei Xu, Yuning Chen, Xumiao Zhang, Xianshang Lin, Pan Hu, Yunfei Ma, Songwu Lu, Wan Du, Z Morley Mao, Ennan Zhai, et al. Cloudeval-yaml: A realistic and scalable benchmark for cloud configuration generation. 2023.

[99] Yifei Xu, Yuning Chen, Xumiao Zhang, Xianshang Lin, Pan Hu, Yunfei Ma, Songwu Lu, Wan Du, Zhuoqing Mao, Ennan Zhai, et al. Cloudeval-yaml: A practical benchmark for cloud configuration generation. *Proceedings of Machine Learning and Systems*, 6:173–195, 2024.

[100] Francis Y Yan, Hudson Ayers, Chenzhi Zhu, Sadjad Fouladi, James Hong, Keyi Zhang, Philip Levis, and Keith Winstein. Learning in situ: a randomized experiment in video streaming. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 495–511, 2020.

[101] Yubo Yan, Panlong Yang, Jie Xiong, and Xiang-Yang Li. Opencarrier: Breaking the user limit for uplink mu-mimo transmissions with coordinated aps. *ACM Transactions on Sensor Networks (TOSN)*, 18(2):1–21, 2022.

[102] Kang Yang, Yuning Chen, and Wan Du. Gwrf: A generalizable wireless radiance field for wireless signal propagation modeling. *arXiv preprint arXiv:2502.05708*, 2025.

[103] Richard Yao, Tom Heath, Aaron Davies, Tom Forsyth, Nate Mitchell, and Perry Hoberman. Oculus vr best practices guide. *Oculus VR*, 4:27–35, 2014.

[104] Xiaoqi Yin, Abhishek Jindal, Vyas Sekar, and Bruno Sinopoli. A control-theoretic approach for dynamic adaptive video streaming over http. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, pages 325–338, 2015.

[105] Anlan Zhang, Chendong Wang, Yuming Hu, Ahmad Hassan, Zejun Zhang, Bo Han, Feng Qian, and Shichang Xu. Habitus: Boosting mobile immersive content delivery through full-body pose tracking and multipath networking. In *USENIX NSDI*, 2024.

[106] Ding Zhang, Mihir Garude, and Parth H Pathak. Mmchoir: Exploiting joint transmissions for reliable 60ghz mmwave wlans. In *Proceedings of the Eighteenth ACM International Symposium on Mobile Ad Hoc Networking and Computing*, pages 251–260, 2018.

[107] Ding Zhang, Puqi Zhou, Bo Han, and Parth Pathak. M5: Facilitating multi-user volumetric content delivery with multi-lobe multicast over mmwave. In *Proceedings of the 20th ACM Conference on Embedded Networked Sensor Systems*, pages 31–46, 2022.

[108] Wenxiao Zhang, Bo Han, and Pan Hui. Sear: Scaling experiences in multi-user augmented reality. *IEEE Transactions on Visualization and Computer Graphics*, 28(5):1982–1992, 2022.

[109] Xinyu Zhang, Karthikeyan Sundaresan, Mohammad A Khojastepour, Sampath Rangarajan, and Kang G Shin. Nemox: Scalable network mimo for wireless networks. In *Proceedings of the 19th annual international conference on Mobile computing & networking*, pages 453–464, 2013.

[110] Zhilong Zheng, Yunfei Ma, Yanmei Liu, Furong Yang, Zhenyu Li, Yuanbo Zhang, Jiuhai Zhang, Wei Shi, Wentao Chen, Ding Li, et al. Xlink: Qoe-driven multi-path quic transport in large-scale video services. In *ACM SIGCOMM*, 2021.

[111] Xiao Zhu, Jiachen Sun, Xumiao Zhang, Y Ethan Guo, Feng Qian, and Z Morley Mao. Mpbond: efficient network-level collaboration among personal mobile devices. In *ACM MobiSys*, 2020.

[112] Johannes Zirngibl, Philippe Buschmann, Patrick Sattler, Benedikt Jaeger, Juliane Aulbach, and Georg Carle. It's over 9000: analyzing early quic deployments with the standardization on the horizon. In *ACM IMC*, 2021.

[113] Anatolij Zubow, Sven Zehl, and Adam Wolisz. Bigap—seamless handover in high performance enterprise ieee 802.11 networks. In *NOMS 2016-2016 IEEE/IFIP Network Operations and Management Symposium*, pages 445–453. IEEE, 2016.

STORM: a Multipath QUIC Scheduler for Quick Streaming Media Transport under Unstable Mobile Networks

Liekun Hu[1] *and Changlong Li*[1,2,3]*
[1]*School of Computer Science and Technology, East China Normal University*
[2]*Jianghuai Advance Technology Center, Hefei 230026, China*
[3]*MoE Engineering Research Center of Hardware/Software Co-Design Technology and Application*

Abstract

The rapid proliferation of streaming media applications has driven the need for multipath transport on mobile devices. While multipath techniques successfully improve throughput by exploiting multiple network interfaces, our study reveals that path instability leads to excessive end-to-end latency. This paper analyzes the data path of multipath networks and observes that the high latency is always caused by the "last mile" wireless link, instead of the core network. Additionally, unlike traditional scenarios, both reliable and unreliable data are transmitted across these paths. However, existing multipath schedulers did not fully account for the reliability characteristics in the design. To address this gap, this paper proposes STORM, a novel multipath scheduler that aims to ensure low latency under unstable mobile networks. We integrate STORM with the mobile device's wireless modules (e.g., WiFi and 5G). STORM differentiates between reliable and unreliable traffic. This approach prevents retransmissions from hindering critical data flows. Our evaluation on real devices shows that STORM reduces tail packet delay by 98.2% and improves the frame rate of streaming media by 1.95x under unstable networks, compared to the state-of-the-art.

1 Introduction

Recent years have witnessed the popular trend of streaming-media-based applications, such as real-time footage of unmanned aerial vehicles (UAVs) [45], 360° panoramic videos of virtual/mixed reality (VR/MR) headsets [30], ultra-high definition (UHD) videos [4], and cloud gaming [44]. With their increased demand for high-quality content, multipath transport is increasingly important for mobile devices. By making use of multiple network interfaces (e.g., WiFi, cellular network, and satellite communication) on mobile devices, the throughput of streaming-media transport can be enhanced as the data is split over various sub-flows and transmitted in parallel.

*Changlong Li is the corresponding author, email: clli@cs.ecnu.edu.cn.

TCP-based multipath transport (MPTCP [31]) is the most famous protocol, establishing a subflow over each TCP path. MPTCP is naturally unsuitable for streaming media data transmission for its strict reliability and retransmission mechanism [29]. Differently, QUIC supports unreliable transport as it is developed based on UDP, which is more friendly for transmitting streaming media data. Hence, the industry has been moving toward QUIC in many emerging scenarios, and as its extension, multipath QUIC (abbreviated as MPQUIC) has been widely studied recently [21, 22, 26, 49, 51].

However, most bandwidth-hungry applications are also latency-sensitive. For example, the latency of VR visuals should be less than 130 ms, or people may feel dizzy during usage [27]. Even though MPQUIC enhances throughput, this paper shows that the end-to-end latency is unexpectedly high if the signal of one path becomes unstable. Our evaluations in a moving vehicle illustrate that the tail latency is up to 4.59s when enabling MPQUIC, which is 6.38x higher than the original system, even though the average throughput is effectively enhanced. Unfortunately, with the increasing mobility, such as high-speed railways [25], vehicles [16, 26], and the Electric Vertical Takeoff and Landing (eVTOL [1]) in the near future, the instability of wireless networks is an increasingly common phenomenon. Enhanced mobility introduces more frequent and abrupt cellular network handovers and WiFi roaming et al. Therefore, ensuring quick streaming media transport on multipath when the network becomes unstable, instead of the normal stable case, is crucial in practice.

As the core element of MPQUIC, the scheduler determines when and how to distribute packets to each sub-flow. This paper shows that existing schedulers cannot address the latency issue of MPQUIC from neither '*when*' nor '*how*' aspects.

First, existing schedulers' response speed on network path deterioration is slow, which cannot meet the low-latency demand of streaming media services. Indeed, the mobility-induced instability problem has been considered in recent studies [5, 22, 49]. However, existing solutions estimate the path status based on the congestion status on a path within a certain period or the already appeared user experience phe-

nomena. Such a design philosophy tends to lag, especially for wireless links that experience unexpected outages accompanied by high burst loss. We find that the scheduler does nothing at the moment that the signal of one path already becomes weak, and start to reduce the data transmission proportion on a certain path while the signal of this path is already recoverd. In this paper, we make an interesting observation that the transmission latency bottleneck lies in the "last mile wireless link" (e.g., link from end devices to the base station or access point) rather than the core network. By co-designing with the end device's WiFi and 4G/5G module, the scheduler has the potential to be aware of unstable wireless link conditions and do scheduling instantly.

Second, reliable and unreliable packets are allowed to be transported on QUIC paths simultaneously [28]. Existing schedulers did not take the packet reliability into account. When one path faces burst loss, our evaluation in Section 2 will show that the retransmission of reliable packets blocks the unreliable ones, and the unreliable packets further block many reliable packets. Specifically, the scheduler consistently prioritizes retransmitted reliable packets over normal packets in MPQUIC. During severe burst loss, a substantial volume of reliable packets requiring retransmission monopolizes bandwidth resources, leading to a backlog of unreliable packets. Since the existing scheduler treats reliable and unreliable packets without differentiation, the subsequent reliable packets have to wait for all accumulated unreliable packets to be sent.

Inspired by the observations, this paper proposes STORM, a **M**ultipath QUIC **S**cheduler for quick st**r**eaming-media **T**ransport under m**o**bile networks. STORM aims to ensure low latency of streaming media applications when the multipath suffers "network storm". To achieve this goal, two novel schemes are further proposed: a signal-watermark mechanism (**SWM**) and a Reliability-aware scheduling (**RAS**) strategy. The SWM implementation relies on tightly coupling the MPQUIC software stack with the wireless network modules on the end device. Like the memory-watermark mechanism in the Linux kernel [17], SWM maintains a signal-watermark to estimate the path state. Compared to existing solutions, the lightweight SWM can be aware of the network storm more timely and accurately. Based on that, RAS realizes reliability differentiated scheduling by managing them as blocks and streams separately.

We have developed STORM based on the MPQUIC protocol and implemented it on real devices. Mobile systems can use STORM without rooting the device or recompiling the kernel. Experimental results show that the tail packet delay decreases by 98.2% when enabling STORM, and the average retransmission ratio is reduced by 12.8%. With STORM, the user experience is effectively improved. Specifically, the frame rate of streaming media data is improved by 1.95x when under unstable networks, compared to the state-of-the-art [21, 51].

The contributions of this paper are summarized as follows:

- STORM is the first effort to address the latency issue of MPQUIC under unstable mobile networks. By making use of the characteristics of QUIC protocol and streaming media data, this paper demonstrates that the transport latency with MPQUIC has the potential to be further reduced.
- We breakdown the data path of multipath networks and observe that the bottleneck is located at the "last mile" wireless link, instead of the core network. Inspired by this observation, this paper shows that the multipath scheduler can be designed in a lightweight but more practical approach.
- This paper is the first to propose a scheduler that distinguishes reliable and unreliable data on multipath. In this approach, the streaming media data on emerging scenarios can be transported with minimized retransmission, which is critical to reducing the end-to-end latency.
- We have implemented STORM on real-life mobile devices. Experimental results show that tail packet delay with STORM can be reduced by 98.2% and the frame rate of streaming media data is improved by 1.95x when under unstable networks, compared to the state-of-the-art solutions.

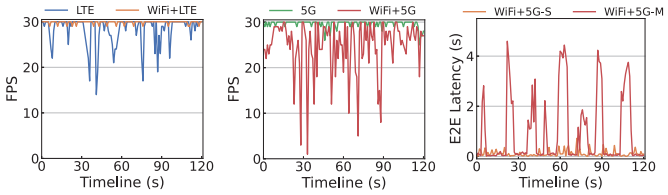
2 Background and Motivation

2.1 Trends in Mobile Applications

The development of STORM is motivated by a series of trends in mobile application scenarios.

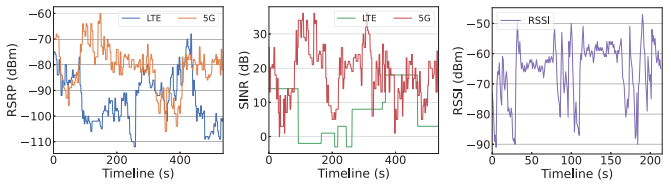
Emerging Scenarios and Demands. There are diversified streaming media-based usage scenarios (e.g., video conferencing [5], AR/VR [7], teleoperated driving [6], and cloud gaming [44]) nowadays that significantly change people's lives. For example, using Apple Vision Pro [11], the data from a neighbor Macbook can be presented on an MR headset in a 3D approach. More people prefer to have conferences and buy online, which spawns a wave of Internet celebrity economy [19]. The COVID-19 pandemic is further accelerating such transitions in consumers' habits. In addition, modern operation systems (e.g., Fuchsia [12] and Harmony [13]) are trying to connect various mobile devices as a unit. They expect to let consumers use remote devices as if in local [3, 18, 40]. These emerging scenarios need high network bandwidth support. It motivates the development of multipath transport techniques. However, it imposes more stringent user experience requirements because viewers are less tolerant of slow response speed [39]. Besides improving throughput, the transport latency should also be considered in the scheduler design.

Enhanced Mobility. Different from traditional computers, mobile devices always move. People may use their phones



(a) Stationary Case (b) Moving Case (c) E2E latency

Figure 1: Frame rate in stationary and moving cases.



(a) LTE and 5G fluctuation (RSRP + SINR) (b) WIFI RSSI fluctuation

Figure 2: Signal strength fluctuation during moving.

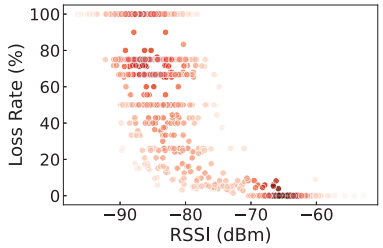


Figure 3: Impact of signal strength on loss rate.

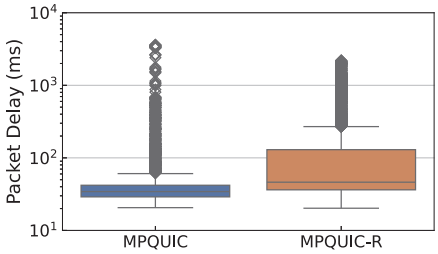


Figure 4: Packet delay in MPQUIC (Reliable & Unreliable) and MPQUIC-R (Fully reliable) mode.

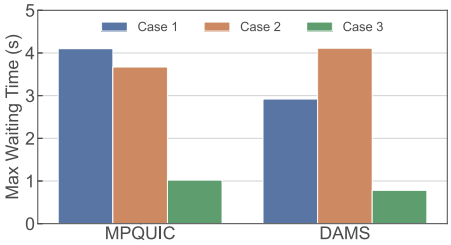


Figure 5: Maximum waiting time of critical data in different cases.

when walking or on the high-speed railway. Video streaming from vehicles unlocks a host of novel applications, like in-vehicle entertainment and gaming [9]. The mobility can lead to sudden signal changes on the end device side, caused by cellular network handover or weak WiFi signal et al. With the development of the next-generation cellular network, mobile devices will suffer more handover in the future because the coverage area of a single 5G/6G base station is smaller than that of the 4G LTE base station [24]. The enhanced mobility introduces more frequent and sudden signal fluctuates.

Reliability Differentiated Transmission. The importance of transmitted data is different. For example, video streams have I-frames and P-frames [48]. I-frames are independent and serve as the start of a GOP (Group of Pictures), while P-frames depend on the previous frame. The loss of an I-frame renders all dependent data useless, but existing systems (e.g., SVC extensions [36]) allow skipping P-frames without affecting the decoding of subsequent frames. Transporting all the data reliably prolongs the latency. MPQUIC supports reliability differentiated transmission: critical data is transmitted reliably while uncritical data is not.

2.2 Effect of MPQUIC on QoE

To explore the effect of modern MPQUIC on the Quality of Experience (QoE), this paper conducts a series of evaluations on a Xiaomi 12S Pro with LTE, 5G, and WiFi network interfaces. We ported the open-source Ringmaster [33, 34] video conference platform to Android, enabling two conference endpoints to communicate via MPQUIC. We adopt frame rate as the metric to quantify the smooth experience and collect the FPS (frame-per-second). Furthermore, the per-frame end-to-end latency is measured.

2.2.1 Effect on FPS

We first evaluate the FPS of UHD video conferences under LTE. Fig. 1(a) shows that single-path transport results in frame rate fluctuations. Specifically, the worst frame rate is only 14fps. Since LTE bandwidth is insufficient to support UHD video conferences, the delivery interval between frames becomes prolonged, causing a decrease in frame rate. Also in this position, enabling multipath (LTE+WiFi) effectively enhances FPS, with all rates exceeding 29fps. This shows that the multipath technique improves bandwidth-hungry apps.

When moving on campus, the signal strength changes. For multipath transport, the frame rate significantly degrades if one path's signal is poor. FPS is even worse than in single-path transport. As shown in Fig. 1(b), average FPS with 5G is 29.6fps, while reduced by 18.4% with 5G+WiFi. In the worst case, single-path 5G has a frame rate of 25fps when moving, while multipath drops to 10fps. That is, even though multipath transport is necessary for streaming media-based applications, its effect is negative instead of positive in some cases.

2.2.2 Per-frame E2E Latency

This paper further explores the end-to-end (E2E) latency of each frame. For a high user experience, the latency should be minimized. As shown in Fig. 1(c), 92.3% of the latency for multipath transmission is less than 100ms. However, when moving, the tail latency reaches 4.59 seconds, which is 5.37x of the stationary case. Such high latency makes real-time communication for video conference attendees impossible, causing frequent frame drops and video artifacts on the screen.

2.3 Root Cause Analysis

2.3.1 Mobility-induced Signal Changes

To explore the reasons, the signal strength during evaluation is analyzed. We collect three types of signal strength indicators at the physical layer through the OS's API: Received Signal Received Power (RSRP), Signal to Interference plus Noise Ratio (SINR), and Received Signal Strength Indicator (RSSI) [15, 32, 47]. At the transport layer, the packet loss rate is recorded. Fig. 2 shows that the wireless signal strength changes dramatically when the device moves around campus. In some areas, the WiFi signal drops, while the cellular network signal drops suddenly in other places. For example, we find that public WiFi signals are stronger in office buildings, while cellular network signals tend to be weaker indoors compared to outdoors.

We further explore the effect of such signal fluctuations on the transmission. Fig. 3 shows the relationship between RSSI (lower dBm values mean worse signal) and loss rate. In the figure, each point represents an RSSI-loss measurement, with darker red shades indicating higher point concentration. Specifically, we can see all measured loss rates remain under 10% at -70dBm. As the RSSI drops to -80dBm, a significant cluster emerges between 60% and 80% loss, and in some cases reaches 100%. When the value of RSSI is reduced, the loss rate tends to be higher. This phenomenon is widely observed in the evaluation. As loss rates rise, more retransmissions occur, consuming bandwidth that could serve subsequent data.

The results demonstrate that the scheduler cannot respond to these rapid signal changes. This is due to the "visibility gap" and unawareness of link outages. Our log analysis indicates that the scheduler selects and schedules according to the path bandwidth and RTT obtained by the congestion control mechanism. As a result, severe burst loss can last several seconds. Due to dependencies between packets on the two paths, overall performance is reduced. The delay in data arrival prevents rendering the next frame, causing video stutter and video artifacts.

2.3.2 Scheduling without Reliability Differentiation

Keyframes (e.g., I-frames) are crucial for video smoothness, as non-keyframes depend on them for decoding [50]. Without keyframes, non-keyframes are useless. MPQUIC provides reliable transmission for these critical data, while other data are transmitted unreliably. Theoretically, this scheme can effectively reduce the additional delay due to lost retransmissions. However, the practice performance is not as expected. Fig. 4 shows that MPQUIC's median and third quartile packet delays improve by 26.6% and 67.55% compared to MPQUIC-R. However, MPQUIC has a higher maximum packet delay outlier (rhombic scatter) than MPQUIC-R, reaching up to 1.65x.

We further explore the cause of this phenomenon by calculating the waiting time of keyframes in the video conference. In our study, waiting time refers to the delay from when the application hands data to MPQUIC until MPQUIC has finished transmitting the data. We compare the default MPQUIC scheduler (minRTT [21]) with the state-of-the-art deadline-aware scheduler (DAMS [51]). The deadline is set to 300ms[1].

We evaluate the scheduler in three cases. (1) Both critical and uncritical data are treated as reliable and scheduled without differentiation. (2) Critical data are treated as reliable, and uncritical data as unreliable, flagged in QUIC as `STREAM` and `DATAGRAM` frames [28], respectively. (3) Similar to Case-2, with modified scheduling: unreliable data waits if any reliable data are in the sending queue.

Fig. 5 shows the critical data, maximum waiting time in three cases when moving around the campus. In Case-1, the maximum waiting time is 4.1s for MPQUIC and 2.92s for DAMS. When distinguishing unreliable packets, the maximum waiting time for MPQUIC was only reduced by 11.79%. Unexpectedly, the maximum waiting time for DAMS increased by 39.93%. Comparing Case-2 and Case-3, we observe a significant reduction in maximum waiting time.

Burst loss from poor signals triggers extensive retransmissions of reliable data, which are prioritized over all normal traffic. Consequently, unreliable data remains queued until retransmissions are complete. Afterward, queued unreliable data floods the network, delaying subsequent reliable data. This mutual blockage increases latency, resulting in worse tail packet delay for MPQUIC than for MPQUIC-R. Ideally, reliable data should not be so heavily affected by unreliable data. Case-3 shows that prioritizing reliable data does help. A scheduler that better coordinates both data types is needed.

3 Design

3.1 STORM Overview

This paper proposes STORM, an MPQUIC scheduler for quick streaming media transport under mobile networks. STORM aims to ensure low latency when the multipath suffers unstable networks.

3.1.1 Key Components

See Fig. 6, to achieve STORM, two core components are maintained: a signal-watermark mechanism (SWM) and a reliability-aware scheduling (RAS) strategy.

Signal-watermark mechanism. As analyzed, link outages during device movement and the sender scheduler's unawareness result in a vicious cycle of packet loss and retransmissions. This phenomenon motivates us to propose a memory-watermark-like mechanism for the mobile devices' MPQUIC.

[1]We set the deadline as 300ms by default since delivering latency over this value will cause significant delays and disruptions [20].

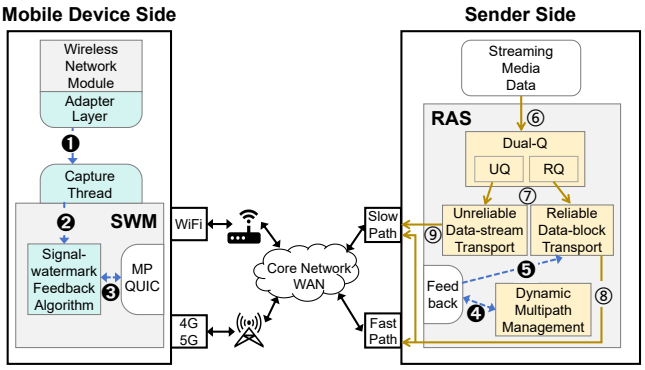


Figure 6: Architecture overview.

The basic idea is to quantify the path's signal strength, compare it with a predefined watermark, and then send differentiated feedback based on the comparison. There are many challenges to tackle in SWM design, for example, how to quantify multiple signal strength indicators into a signal quality index, and how to ensure the flexibility and applicability of the watermark. We will discuss this in detail in Section 3.2.

Reliability-aware scheduling. Based on SWM, this paper further proposes an RAS strategy on the sender side. RAS treats reliable and unreliable data differently. Specifically, reliable data is organized as blocks and unreliable data is organized as streams. The details and advantages of such an approach will be discussed in Section 3.3.

3.1.2 Workflow

Also see Fig. 6, STORM's workflow consists of a feedback flow and a scheduling flow, forming a scheduling system that strings together the two components. Specifically, the blue arrows in the figure are feedback flows (❶ - ❺), while the brown arrows are scheduling flows (⑥ - ⑨).

Feedback flow begins at the adapter layer encapsulated in the wireless network module of the mobile device. (1) A dedicated thread collects signal strength indicators from this adapter layer. (2) These indicators are input to the Signal-Watermark feedback algorithm. (3) The algorithm sends feedback to the sender via MPQUIC and adjusts the watermark based on subsequent results. (4) Upon receiving the feedback, the sender notifies the Dynamic Multipath Management module in the RAS component to decide whether to activate or deactivate a path. (5) The Reliable Data-block Transport module also uses this feedback to adjust its path allocation ratio. After these steps, the feedback flow completes, and scheduling can proceed using the updated signal conditions. (6) The scheduling flow starts once streaming media data arrives at the RAS component, which places the data into Dual-Q based on reliability requirements. (7) Dual-Q maintains separate queues for different reliability classes, each governed by its own scheduling strategy. (8) The Reliable Data-block Transport module distributes reliable data across two paths to maximize aggregate bandwidth, reducing allocations on any path identified (via SWM feedback) as having poor signal. (9) The Unreliable Data-stream Transport module then injects data into idle times on the slower path after each reliable data send.

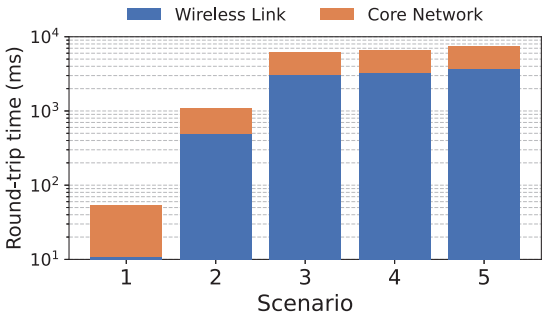


Figure 7: Breakdown of End-to-End RTT: Wireless Link vs. Core Network RTT (Scenario 1: strong signal location for stationary; Scenario 2: underground mall; Scenario 3: subway; Scenario 4: hotel room; Scenario 5: high-speed railroad).

3.2 Signal-watermark Mechanism

3.2.1 Data Path Breakdown

Extensive research has examined whether wireless links (so called the "last mile") act as a bottleneck for network throughput [2, 37]. However, the latency bottleneck on the data path has not been thoroughly analyzed. In this paper, we breakdown the data path to understand where the latency bottleneck locates on the network path.

To accurately obtain the round trip time (RTT) between the base station and the device, we innovatively utilize the first hop IP address after entering the core network from the base station. Specifically, we employ a traceroute-like function to identify this first-hop IP address. Since the latency between the base station and its first hop is typically under 5ms, measuring the RTT of this address effectively represents the wireless link's latency.

Fig. 7 shows results from various scenarios, including poor RSRP (low coverage) and poor SINR (high interference). We observe that poor RSRP (Scenario 2 and 4) leads to more frequent failed handovers compared to poor SINR (Scenario 3). The severity of packet loss and wireless link latency due to handover failure intensifies whenever either RSRP or SINR deteriorates. In the worst case, the E2E RTT reaches 3,781 ms, with the wireless link RTT at 3,750 ms. This trend is consistent across all mobile scenarios, indicating that despite advancements in wireless technology claiming low latency, the wireless link remains the latency bottleneck in mobile environments. Therefore, designing SWM to monitor signal conditions and provide feedback is essential for quickly reducing latency.

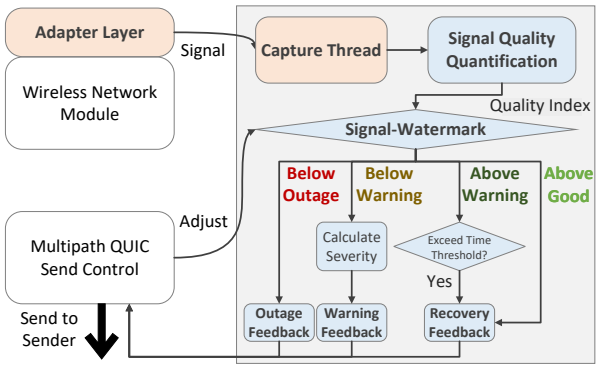


Figure 8: The workflow of SWM. The gray box works on the MPQUIC protocol stack.

3.2.2 Co-Design with the Wireless Network Module

As shown in Fig. 8, to enable SWM's signal-awareness, we co-designed MPQUIC with the wireless network module. Our goals were to provide a unified way to retrieve signal strength across different platforms and to maintain MPQUIC's performance by avoiding blocking operations.

Originally, the mobile device's wireless module ran independently of MPQUIC. To allow seamless access to signal strength indicators, we extended it with an adapter layer that standardizes retrieval across platforms (see the orange box on the left in the figure). MPQUIC then obtains signal strength through a single, unified API.

MPQUIC uses a non-blocking I/O event loop for multipath transfers. Directly polling signal strength in this loop could block it, degrading performance. To avoid this, we delegate signal-strength capture to a dedicated thread (see the orange box on the right in the figure). This thread periodically calls the adapter layer's API and updates atomic variables in SWM. By employing lock-free programming, we minimize overhead on MPQUIC's main event loop.

3.2.3 Signal-watermark Feedback Algorithm

Upon receiving signal strength indicators from the capture thread, the Signal-Watermark feedback algorithm begins execution. Its workflow is shown in the blue box of Fig. 8.

Signal Quality Quantification. Signal strength indicators such as SINR and RSRP vary in scales and units, influencing signal quality from distinct dimensions—coverage and interference. To coherently quantify their combined impact, we normalize these indicators into a unified interval for direct comparison. Since SINR and RSRP inherently exhibit multiplicative and proportional characteristics, we adopt a weighted geometric mean rather than a linear combination. The weights tune the relative influence of each indicator, recognizing that SINR and RSRP affect signal quality to different extents (see Section 3.2.1). As shown in Eq. 1, the base quality Q_{base} is defined as:

$$Q_{\text{base}} = \left((SINR_{norm})^{w_{sinr}} \times (RSRP_{norm})^{w_{rsrp}}\right)^{\frac{1}{w_{sinr}+w_{rsrp}}} \quad (1)$$

where w_{rsrp} and w_{sinr} are adjustable weights in [0,1]. Requiring the sum of weights to exceed 1 imposes a compressive effect, preventing large jumps in Q_{base} from minor improvements and pulling Q_{base} toward moderate values when indicators are extremely low, maintaining stability and avoiding excessive dispersion. Note that for WiFi, which only uses RSSI, Q_{base} equals $RSSI_{norm}$.

To further refine this measure, we introduce a modulation factor that reacts to packet loss trends. The packet loss rate is a critical indicator of whether current feedback decisions are effective: a persistent rise in loss ($\Delta loss > 0$) shows that the signal-degrading path is still over-utilised. Accordingly, the modulation factor amplifies the negative adjustment, aggressively reducing that path's utilization and achieving the desired effect more quickly. Conversely, when $\Delta loss \leq 0$, the factor does not raise the quality score, because temporary loss reductions do not yet signal genuine recovery. The modulation factor $H(\Delta loss)$ is defined as:

$$H(\Delta loss) = \begin{cases} 1, & \Delta loss \leq 0 \\ \exp(-\gamma \cdot \Delta loss), & \Delta loss > 0 \end{cases} \quad (2)$$

Here, $\gamma \in [0,1)$ controls sensitivity to increasing loss: a larger γ sharply reduces quality when loss rises, while a smaller γ allows for subtle adjustments. This keeps the base quality as the primary determinant, with $\Delta loss$ as a secondary cue.

The final quality index $Q \in [0,1]$ is:

$$Q = Q_{\text{base}} \times H(\Delta loss) \quad (3)$$

As shown in the blue box at the top right of Fig. 8, the final quality index will be passed to the Signal-watermark for comparison.

Adaptive Signal-watermark Feedback. After obtaining the quality index, it is compared against three watermarks: *Good*, *Warning*, and *Outage*. If the index falls below the *Warning* watermark, a `warning` feedback is generated, indicating the gap between the current index and the *Warning* watermark, to help adjust path allocation ratios. Dropping below the *Outage* watermark triggers high-priority `outage` feedback, instructing the sender to stop scheduling on that path. When the index exceeds the *Warning* watermark, SWM will consider generating `recovery` feedback.

To prevent frequent feedback caused by the unstable quality index, a timing threshold mechanism with Fast Suppression/Slow Recovery is implemented. Feedback is promptly generated when the index falls below *Warning* or *Outage*. However, `recovery` feedback is only immediately issued when the index recovers from below *Outage* to above *Good*. If the index rises to between *Outage* and *Warning*, no action is taken. When the index surpasses *Warning*, a timer starts, and `recovery` feedback is generated only if the index remains above *Warning* for a threshold time T_{rec}; otherwise, the timer resets. This approach ensures that signal improvements are sustained.

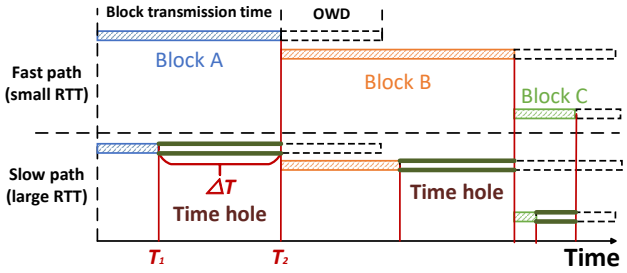


Figure 9: Slow path time hole.

However, rapid fluctuations in the quality index can lead to a worst-case scenario where the corresponding path remains inactive for an extended period. Specifically, if the index continuously and quickly oscillates across multiple watermarks, it may never stabilize above the *Warning* watermark for the entire recovery feedback period (T_{rec}), thus failing to generate any `recovery` feedback. Even if it briefly exceeds the *Good* watermark, it will be deactivated again upon falling below the *Outage* watermark. In fact, utilizing paths with severe quality fluctuations offers no performance advantage and considerably impairs user experience. Hence, sustained deactivation of such unstable paths is beneficial.

The default watermark settings were based on extensive experiments (see Fig. 3), but device-specific conditions, network technologies, and environments can affect effectiveness. To accommodate this variability, watermarks are adaptive. Specifically, the SWM evaluates the effects of previous feedback through MPQUIC send control. If the *Outage* watermark fails to prevent high burst loss, it is increased until the loss is curtailed. Similarly, if the burst loss exceeds 10% when below the *Warning* watermark, it is raised by a smaller step for finer granularity. To prevent aggressive path deactivation and bandwidth waste, the *Outage* watermark decreases if burst loss above 50% does not occur, ensuring alignment with actual link outages. These adaptive adjustments keep the watermarks calibrated to real-world link conditions.

3.3 Reliability-aware Scheduling

To enable the sender to fully utilize SWM feedback, a new scheduling scheme must be devised. Merely adjusting multipath scheduling allocation ratios and deactivating link outages path may improve performance to some extent, but does not address the root cause. Although SWM can mitigate high burst loss, it cannot achieve a loss-free environment in real wireless networks. As analyzed in Section 2.3.2, loss cause reliable and unreliable data to block one another, making coordinated scheduling of both essential for further improving latency.

Dual-Q for Reliable and Unreliable Data. To coordinate the scheduling of data with different reliability requirements, we manage them using two separate queues. Specifically, we revise the MPQUIC protocol to adopt a Dual-Q architecture, consisting of an RQ for reliable data blocks and a UQ for unreliable data streams. In the RQ, reliable data are handled as individual blocks, each with its own priority and deadline set by the application. The scheduler establishes a subflow for each path and distributes these blocks accordingly. Meanwhile, in the UQ, unreliable data are sorted by their deadlines. if a loss occurs, RAS discards the affected data and proceeds to the next.

Maintaining two distinct data types, including blocks for reliable data and streams for unreliable data, offers two key advantages. First, by representing reliable data as discrete blocks, each block is fully delivered as a unit, meeting the requirement for complete, in-order delivery. Second, treating unreliable data as streams enables more flexible use of idle bandwidth without the constraints imposed by fixed block sizes. We now introduce the scheduling strategy in detail.

Reliable Data-block Transport. For reliable data blocks, RAS aims to deliver as many as possible before the deadline. Aligning arrival rather than sending times maximizes the faster path's potential [8]. To ensure packets belonging to one block can arrive at a similar time, the data size allocated on the paths ($Size_1$ and $Size_2$) follows the constraint:

$$\frac{Size_1 \times bw_{agg}}{bw_1 \times F_1} + OWD_1 = \frac{Size_2 \times bw_{agg}}{bw_2 \times F_2} + OWD_2 \quad (4)$$

Here, bw_i represents the current path bandwidth, and OWD_i refers to the one-way delay on the corresponding paths. Path bandwidth is estimated based on the congestion control algorithm employed by the MPQUIC. To prevent the division result from approaching zero, we normalize with the aggregated bandwidth bw_{agg} (the sum of the two paths' bandwidths). F_i is determined by the mobile device side SWM's feedback: more severe `warning` feedback lower the F_i value, indicating a higher potential loss rate. Fewer data allocations are made to paths with lower F_i values to avoid burst loss. Upon receiving a `recovery` feedback, the F_i value is restored to one.

RAS calculates the residual time to send at that moment for each subflow. For subflow on path i, its residual time is:

$$Trsd_i = deadline_i - Tpsd_i - OWD_i \quad (5)$$

Here, $Tpsd_i$ represents the time already passed before processing. If less than 0, the data is proven to be expired. RAS cancels the sending operation and evicts it from RQ.

The urgency of the deadline depends on both residual time and block size. A large block with a distant deadline can still be urgent. Based on Eq. 5, we calculate the gap between the current bandwidth (Cbw) and the minimal required bandwidth (Mbw), as shown in Eq. 6. Mbw corresponds to $\frac{Size_i}{Trsd_i}$. If not processed immediately, the required bandwidth increases as the block nears its deadline.

Blocks with smaller Gbw values are more urgent. If the gap is less than zero, the data cannot be delivered within the remaining time under current conditions. In this case, RAS

delays processing this block to prioritize others.

$$Gbw = \min_{i \in \{1,2\}} \left(Cbw_i - \frac{Size_i}{Trsd_i} \right) \quad (6)$$

Ultimately, RAS determines weights based on the bandwidth gap and block priority. Blocks with lower weight are more likely to be scheduled.

We normalize the bandwidth gap and block priority to calculate the final weights. The gap is divided by bw_{agg}, and the priority is divided by the current maximum priority value (P_{max}). Parameter α adjusts the importance of each in the weight calculation. To avoid unfairness to smaller blocks, we multiply the gap by parameter R, representing the unsent data ratio in a block. This adjustment ensures blocks that have already sent most of their data receive a lower weight (higher priority).

$$weight = \alpha \times \frac{Gbw}{bw_{agg}} \times R + (1-\alpha) \times \frac{priority}{P_{max}} \quad (7)$$

Unreliable Data-stream Transport. After achieving efficient scheduling for reliable data blocks, we must consider how to handle unreliable streaming data. If we simply allocate them across two paths in the same manner as reliable data, the loss affecting reliable data will continue to cause mutual blockage. Furthermore, as shown in Fig. 9, block A finishes on two paths at T_1 and T_2. The idle period ΔT $(T_2 - T_1)$ on the slow path creates a *time hole*. This *time hole* reduces throughput, and when combined with mutual blockage, it becomes nearly impossible for both data types to meet their deadlines.

RAS manages unreliable data on UQ as a stream arranged by deadlines, with their weight set higher than P_{max} for differentiation. During ΔT, RAS schedules UQ data to the slow path. Because unreliable data is injected only during time hole, any blockage lasts only for a single retransmission. Once that retransmission completes, the next reliable block creates another time hole, allowing the scheduling of unreliable data in this new hole. This approach effectively alleviates mutual blockage and further improves overall delivery speed.

Dynamic Multipath Management. Another challenge is sudden link outages. Upon receiving `outage` feedback from the mobile device, the affected path is quickly deactivated to minimize the impact of burst loss. With only a single path remaining, Eq. 4 is no longer applicable. Instead, RAS employs subsequent equations to calculate each block's weight based on the full allocation to the remaining path. Due to bandwidth constraints, the unreliable stream is scheduled by the deadline only after all reliable blocks have been transmitted and there is available congestion window capacity. When `recovery` feedback is received, the path is immediately reactivated to maximize bandwidth utilization.

Even though subsequent data will not be scheduled to the deactivated path, prior scheduling decisions cannot be undone. If not processed, these data may be delivered late. To solve this, RAS employs deadline-aware reinjection. RAS calculates if unacknowledged data can meet the deadline with the current available path's bandwidth. If feasible, it reinjects the data; otherwise discarding them. Reinjection occurs after the scheduling round to avoid delaying normally scheduled data. Unreliable data are discarded without reinjection.

4 Implement STORM with MPQUIC

STORM is implemented based on the QUIC protocol. Specifically, we developed STORM based on Alibaba's XQUIC [10]. Features of QUIC, like the unreliable QUIC-Datagram based on RFC9221 [28] and multipath functionality incorporated into the IETF WG Draft [21], are used by STORM. Based on that, we modify QUIC with 1,342 lines of C language code, excluding the libraries.

Wireless Network Module Adapter Layer. MPQUIC uses abstract paths to adapt to various mobile network interfaces, preventing direct access to signal strength indicators. We introduce an adapter layer to the wireless network module. On Android, we implemented Java logic to monitor wireless network signal changes. These indicators are stored as atomic variables and accessed by SWM via Java Native Interface (JNI) calls every 100 milliseconds.

The adapter layer implementation introduces minimal modifications to user equipment. Unlike previous approaches that required hardware adjustments or system-level root privileges, our adapter-based method efficiently retrieves cellular signal strength indicators by utilizing standard Android classes (e.g., WifiManager [42] and TelephonyManager [41]). Specifically, the adapter achieves this with only 102 lines of Java code, significantly simplifying developers' workload and seamlessly integrating into existing systems.

Reliable Data-block Transport. MPQUIC supports various frame types to handle different data reliability requirements. Specifically, `STREAM` frames carry reliable data, while `DATAGRAM` frames carry unreliable data [14, 28]. STORM extends this capability by mapping different reliability data to different frame types.

MPQUIC packetizes data into a doubly linked list for write streams, complicating fine-grained scheduling. To minimize overhead, STORM schedules data based on block metadata and caches reliable blocks in a vector.

Unreliable Data-stream Transport. MPQUIC's stream management supports unreliable data streams but faces synchronization challenges: data must be packetized pre-injection into the slow path, complicating alignment between unreliable data on the slow path and reliable data on the fast path. Fixed packet sizes worsen timing mismatches, leading to inconsistent finishes of unreliable transmissions relative to reliable blocks under varying bandwidth conditions.

To solve these issues, STORM introduces a novel scheme. Unreliable data is placed in a doubly linked send queue organized by deadlines, where each node holds an entire data

segment rather than individual packets. This approach defers packetization until injecting data into the slow path, thereby reducing scheduling overhead and improving synchronization with reliable data.

Dynamic Multipath Management. MPQUIC manages multipath usage by sending `PATH_STANDBY` and `PATH_AVAILABLE` frames to notify peers of path states. A STANDBY path does not schedule packets unless the AVAILABLE path is unavailable [21]. However, this mechanism is inadequate for signal-aware path management due to two issues: (1) In-flight packets on a STANDBY path remain until lost, while buffered data continues to transmit, increasing latency and packet loss. (2) If the AVAILABLE path's congestion window is exhausted, new packets are sent through the STANDBY path. These behaviors conflict with the deactivation of paths experiencing link outages, highlighting a limitation in MPQUIC's scheduling.

STORM introduces a new frame type, `PATH_ALERT`, containing feedback from the SWM. Upon receiving a `PATH_ALERT` frame, the sender manages multipath according to RAS's design.

Abstraction and API. STORM encapsulates the process and offers an API to the upper layer. Applications just need to call the *STORM_send* and provide parameters like deadline and priority. Specifically, STORM categorizes application data into Reliable Transport Blocks and Unreliable Transport Streams, executing their logic.

STORM does not use a proxy [25, 26] in its implementation. While proxies reduce application modifications, they only intercept the payload, losing important metadata. This limits QUIC's cross-layer benefits, complicating fine-grained scheduling. Instead, data can be sent directly through the API provided by STORM, reducing code changes.

Evl. Type		Dev.	Equipment		
			NET	MEM	OS
Control Lab		PCs	Ethernet	94GB	Ubuntu 20.04
APP	**Video Conf.**	Phone	WiFi-6, LTE/5G	12GB	Android 14
	Live Stream.	Phone	WiFi-6, LTE/5G	12GB	Android 14
	360° Video	Laptop	WiFi-6	16GB	Ubuntu 18.04

Table 1: Equipment details of the evaluation platforms.

5 Evaluation

5.1 Experiment Setup

This section provides the evaluation from two aspects:

- **Control Network Environment.** We test STORM in a controlled lab environment. The evaluation focuses on three main aspects: microbenchmarks, the performance effects of individual designs, and the performance effects of parameter settings.
- **Applications in the Wild.** Furthermore, we developed three apps for typical scenarios: video conferencing, low-latency live streaming, and 360° video. These applications were evaluated in real-world mobile Internet environments, and the QoE was analyzed.

Parameter Configurations. The parameters in the evaluation are listed in Table 2. The default signal-watermark was set empirically. w_{rsrp} and w_{sinr} are set to 0.8 and 0.6, respectively, because poor RSRP values are usually more strongly associated with failure of handover. Set γ to 0.01 to avoid overly penalizing the common random losses (under 10%) typical of wireless links, while still enabling a rapid score reduction once $\Delta loss$ exceeds 50. T_{rec} is set to 400 ms. α is set to 0.5.

Symbols	Semantics	Setting
Good	High-watermark that represent strong signal	0.45
Warning	Medium-watermark that represent weak signal	0.32
Outage	Low-watermark that represent link outage	0.21
w_{rsrp}	RSRP weight coefficient	0.8
w_{sinr}	SINR weight coefficient	0.6
γ	Modulation factor sensitivity parameter	0.01
T_{rec}	Time threshold for recovery feedback	400 ms
α	Weight coefficient for reliable data transport	0.5

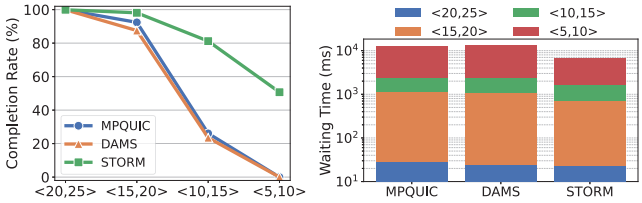
Table 2: Summary of parameters used by STORM.

5.2 Microbenchmark

We developed a toolkit to evaluate STORM's microbenchmark performance under various network conditions. The tool sends six blocks of varying priorities and sizes every 100ms at 30Mbps, each with a 300ms deadline. This setup emulates transmitting a Group of Pictures (GOP) in streaming applications at 60 fps. Specifically, the highest-priority block represents an I frame, three lower-priority blocks represent P frames, and two lowest-priority blocks (transmitted unreliably) represent B frames. Note that STORM supports deadline setting well below 300 ms, discussed in Appendix A.1. Two PCs (see Table 1) are connected to an OpenWrt router with traffic control (*tc*). Ethernet simulates WiFi and cellular networks, avoiding interference. We tested the completion rate and waiting time of high-priority blocks. DAMS [51] and MPQUIC were compared with STORM. Since Ethernet does not reflect wireless link conditions, we only examined STORM performance under a strong signal.

5.2.1 Bandwidth Variation

We evaluated STORM's performance under bandwidth fluctuations using various combinations, denoted in Mbps. Although more combinations were tested and exhibited similar trends, this paper only presents four representative cases: <20, 25> (adequate bandwidth), <15, 20> (nearly adequate), <10, 15> (slightly inadequate), and <5, 10> (severely inadequate). These combinations clearly illustrate STORM's per-



(a) Completion Rate (b) Waiting Time Distribution

Figure 10: Block completion rate and waiting time distribution under various bandwidth combinations.

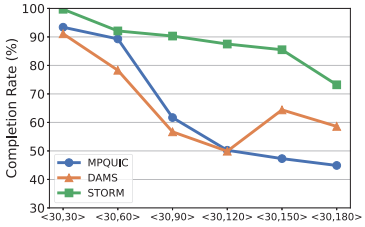


Figure 11: Block completion rate under various RTT combinations.

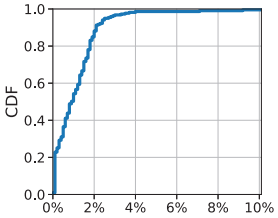
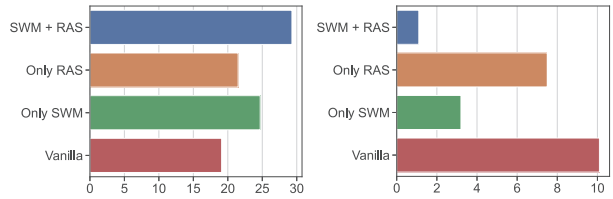


Figure 12: STORM's signal status sync bias rate.



(a) Goodput (Mbps) (b) Rebuffering Time (s)

Figure 13: Ablation study of STORM's design.

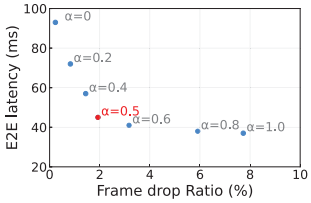


Figure 14: Weight parameter α.

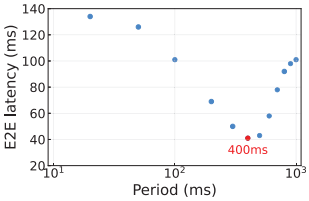


Figure 15: Recovery feedback period T_{rec}.

formance under bandwidth conditions ranging from sufficient to severely insufficient. The RTT for the two paths was set to 30ms and 50ms, with a loss rate of 0.01%.

Completion Rate. Fig. 10(a) shows that with <10, 15> bandwidth combinations, the completion rate[2] of MPQUIC drops from 99.9% to 26.1%. The completion rate of DAMS is reduced by 76.4% compared to the sufficient-bandwidth case. With a bandwidth combination of <5, 10>, both DAMS and MPQUIC nearly reach zero completion rates, unable to deliver on time with only half the required bandwidth. In contrast, STORM's completion rate only decreases by an average of 49.3%.

As Table 3 shows, STORM schedules 66.5% of reliable data to the fast path, while unreliable data is all scheduled on the slow path. In MPQUIC, all data competes equally for the fast path. DAMS, however, utilizes both paths for reliable data and defaults to MPQUIC's approach for unreliable data. As a result, unreliable data consumes 17.4% more bandwidth on the fast path than in MPQUIC. STORM fully utilizes the fast path without wasting the slow path, significantly increasing the completion rate of reliable blocks.

	Reliable data		Unreliable data	
	fast path	slow path	fast path	slow path
MPQUIC	75.1%	24.9%	71.7%	28.3%
DAMS	54.8%	45.2%	89.1%	10.9%
RAS	66.5%	33.5%	0%	100%

Table 3: The ratio of (un)reliable data on the two paths.

Efficiency to Transport High-priority Data. We further verify the advantages of scheduling reliable data more on the fast path. Fig. 10(b) shows the waiting time distribution for high-priority data. The top of each color bar represents the maximum waiting time for that bandwidth combination. As bandwidth decreases, the maximum waiting time for MPQUIC and DAMS increases significantly (by 44.3x and 54.5x, respectively). In the worst case, the time reaches 11 seconds. In contrast, STORM performs better. For example, with the <5,10> bandwidth combination, STORM's maximum waiting time is 49.6% of MPQUIC's and 45.3% of DAMS's.

[2]The completion rate is the ratio of packets successfully delivered before the deadline to the total number of packets.

5.2.2 RTT Variation

We further explore the effect of RTT variation. In the evaluation, one path's RTT is fixed at 30ms, while the other varies from 30ms to 180ms. The bandwidths of the two paths are set to 20Mbps and 14Mbps. As shown in Fig. 11, two phenomena are observed: First, STORM consistently outperforms DAMS and MPQUIC in the completion rate across various RTT combinations. With the <30,120> RTT combination, STORM's completion rate increased by 37.3% and 37.6% over MPQUIC and DAMS, respectively. Second, we observed DAMS's completion rate rising to 64.4% with the RTT combination <30, 150>. Due to excessive RTT differences, DAMS can hardly use the slow path to accelerate delivery, scheduling more data to the fast path. The fast path's 20Mbps bandwidth is sufficient for this higher completion rate. MPQUIC, however, allocates data to the slow path when the fast path's congestion window is exhausted, increasing missed deadlines.

5.3 STORM Deep Dive

In this section, we replay user walking network traces to investigate how each design affects STORM's performance. We modified saturatr [46] to trace when the mobile device detects a signal strength change, providing these records to SWM for simulating signal strength capture. We set up a mpshell [23]

testbed with a sender application and a virtual low-latency live streaming player. The player focuses on network transmission without actual decoding or rendering.

Signal State Sync Bias. Studies show that reporting signal strength to user space can be delayed by up to 200 ms [25], causing synchronization bias between signal change and feedback receipt. Fig. 12 shows that the 'captured vs. actual' signal strength differs minimally: about 1.14% on average and 0.95% at the median. With a capture-to-feedback delay of at most 300 ms, large spatial shifts are unlikely. Even at high speeds, STORM quickly reacts to signal degradation, then waits for stabilization before readjusting.

Ablation Study. We conducted an ablation study to investigate the impact of the Signal-Watermark Mechanism (SWM) and Reliability-Aware Scheduling (RAS) components. Fig. 13 shows that the complete design (SWM+RAS) is crucial for STORM to optimize user experience during signal fluctuations. Disabling SWM reduces goodput by 26.6% and increases rebuffering time by 6.8x. Disabling RAS reduces goodput by 15.7% and increases rebuffering time by 2.9x.

Computational Overhead. STORM is a lightweight multipath scheduler integrated into MPQUIC with negligible computational overhead. RAS's average processing time for data in Dual-Q is 5.03 ms, only 7.4% of the total transmission time (68.1 ms). The feedback algorithms in SWM have $O(1)$ complexity, incurring only μs-level overhead. With our implementation, the number of Java Native Interface (JNI) calls is minimized to further reduce overhead. Specifically, the most time-consuming JNI call introduces an average overhead of 78 μs per invocation. Given that this call occurs once every 100 ms, it results in merely 780 μs of additional CPU time per second, leading to negligible extra CPU usage.

5.4 Sensitive Study

We in this part test the sensitivity of STORM performance to different parameter settings.

Parameter α Tuning. A larger value of α in Eq. 7 leads to STORM scheduling the more urgent packets first, which decreases its consideration of their priority. We evaluate STORM with α values from 0 to 1, shown in Fig. 14. As α increases, E2E latency decreases as the scheduler favors urgent packets for timely delivery, but this raises the frame drop ratio. If critical decoding data is delayed by urgent packets, the player favors received data, compromising smoothness and quality despite lower latency. Thus, a suitable STORM balance is $\alpha = 0.5$.

Recovery Feedback Period. To find the optimal period T_{rec} for SWM to generate recovery feedback, we test a range of values in [20, 50, 100, 200, 300, 400, 500, 600, 700, 800, 900, 1000]ms. Fig. 15 shows the E2E latency for these period settings. Periods under 100ms produce overly frequent recovery feedback. This excessive frequency can disrupt transmission and cause traffic fluctuations. Periods over 800ms waste bandwidth and increase end-to-end latency. We select 400ms as the period for STORM. Note that as mobility increases, this period should be extended to ensure the signal stabilizes.

5.5 Real-world Applications Evaluation

We further explore STORM's effect on user QoE in the real-world mobile network. Three typical streaming media applications were tested on a Xiaomi 12S Pro phone, communicating with Tencent Cloud servers located in Asia (see Table 1).

Video Conference. We prepared three video streams to transport from the server to the client. The server encodes them into VP9 format frames using ffmpeg and libvpx. One stream is a real-time conference, serving as the mainstream. The other two streams are workloads. Mainstream keyframes have the highest priority. The non-keyframes of the other streams are set to unreliable data. Resolution is 1280x720, with a bitrate of 30 Mbps and a frame rate of 30 fps.

Low-latency Live Streaming. We simulate a live pull stream. We encoded a 1080p video at 30Mbps using ffmpeg. I and P frames are high and medium-priority blocks, respectively. And B frames are an unreliable stream. The video plays continuously for two minutes at 30 fps.

360° Video. We developed a 360° video application. The video is delivered as tiles, with tiles near the center of the user's viewport given higher priority. Unreliable masking streams prevent tile skipping. Due to the complexity of implementing 360° video on an Android phone, the client was implemented on a laptop with two wireless cards.

5.5.1 Benefit on QoE

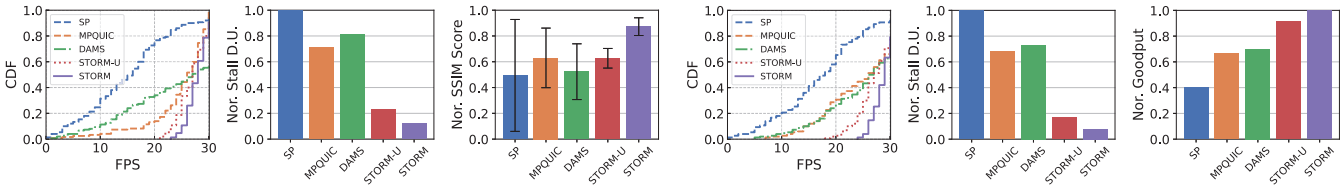
Our evaluation includes different QoE metrics. The following metrics were used:

- **Video Conference:** Frame rate distribution, normalized stall duration, and normalized SSIM score.
- **Low-latency Live Streaming:** Frame rate distribution, normalized stall duration, and normalized video goodput.
- **360° Video:** Peak Signal-to-Noise Ratio (PSNR) and User Perceived Ratio (the ratio of actual bitrate received by the user to the video bitrate).

To explore the real-world performance impact of SWM, we implemented a variant of STORM, STORM-U, using RAS without SWM feedback. Single path (SP) performance is also considered.

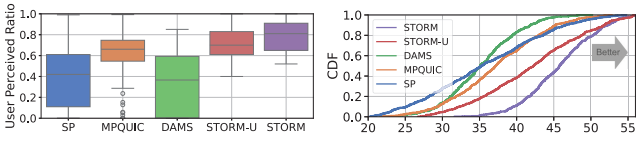
We conducted field tests at five real-world locations, such as the parking lot and the library. These locations are covered by WiFi and cellular networks, with the phone connected to both. To examine mobility's effect on multipath transmission performance, we divided the tests into two scenarios.

In Changing Signal Strength. In this case the user runs three applications while walking within a given area and



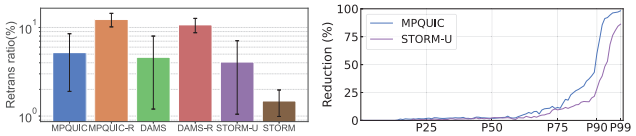
(a) Video conference's QoE (b) Low-latency live streaming's QoE

Figure 16: The video conference and low-latency live streaming's QoE.



(a) User perceived ratio (b) PSNR (dB) distribution

Figure 17: The 360° video's QoE



(a) Retransmission ratio (b) Packet delay reduction

Figure 18: Transport improvement.

APP	Metrics	MPQUIC	DAMS	STORM-U	STORM
1	**FPS**	29.2	29.4	29.4	29.6
	Stall Dur.	1	0.97	0.96	0.96
	SSIM	0.99	0.99	0.99	0.99
2	**FPS**	29.1	29.2	29.3	29.5
	Stall Dur.	1	0.97	0.97	0.96
	Goodput	0.84	0.90	0.93	1
3	**UPR**	0.92	0.95	0.94	0.98
	Median PSNR	47.2dB	47.4dB	47.8dB	47.9dB

Table 4: The QoE of (1) video conference, (2) low latency live streaming, and (3) 360° video applications under a stable signal strength.

experiences sudden link outages both indoors and outdoors. We measured how these rapid signal fluctuations affect each solution.

Fig. 16 shows DAMS's performance declines significantly with signal fluctuations, more so than MPQUIC. In contrast, STORM maintains good performance. A frame rate of 24 fps is generally considered the minimum for motion to appear natural to the human eye. Lower rates produce perceptibly unsmooth motion. For video conference, STORM's unsmooth FPS (below 24 fps) ratio improved by 95.1% compared to MPQUIC and 96.8% compared to DAMS. STORM also excels in stall duration, outperforming MPQUIC by 83.0% and DAMS by 85.2%. SP struggled with tests due to link outages, causing performance fluctuations. Lower FPS and longer stalls lead to more video stutters, while STORM effectively reduces both during signal fluctuations, enhancing user experience.

STORM quickly deactivates the path during link outages, minimizing loss and latency. STORM-U, however, cannot sense sudden changes. Delayed data leads to decoding failures, which in turn produce video artifacts and degrade video quality. For 360° video, STORM's median user-perceived ratio and PSNR are 0.81dB and 46.1dB (see Fig. 17), improving by 11.1% and 7.7% over STORM-U, resulting in the best video quality.

In Stable Signal Strength. In this case, the user runs three applications while staying in a position with a good signal. We begin by showing the QoE in the video conference. Table 4 shows that STORM performed the best. With sufficient bandwidth and low latency in stationary conditions, the performance differences among individual solutions are minimal. Similar results are observed in low-latency live streaming and 360° video.

5.5.2 Transport Improvement

We finally explored how STORM can improve transport performance in real-world applications. MPQUIC-R and DAMS-R are included for comparison.

Fig. 18(a) shows that STORM achieves the lowest retransmission ratio. MPQUIC-R and DAMS-R suffer severe retransmissions during sudden link outages. In contrast, DAMS and MPQUIC perform better in retransmission ratio by delivering non-critical data over unreliable transmissions. Despite its efficient scheduling, STORM-U still encounters an average retransmission ratio of 4.1% due to its inability to sense the wireless link. By quickly deactivating the broken path, STORM reduces the average retransmission ratio to 1.5%. Fig. 18(b) shows the packet delay reduction of STORM compared to other solutions at different percentiles. We selected the optimal MPQUIC and STORM-U solutions for comparison. The results show a 98.2% decrease in packet delay at the 99th percentile with MPQUIC, demonstrating its strong advantage in improving tail packet delay. Although STORM-U outperforms MPQUIC in other aspects, its delayed response to signal changes still results in high tail delays.

6 Related Works

Multipath Transport for Streaming Media. The growing demand for high-throughput and low-latency video streaming has led to much work on multipath transport. MPTCP [31] introduced the concept of establishing subflows across multiple TCP connections to use interfaces such as Wi-Fi and

cellular, but its strict reliability can degrade real-time streaming [29]. Even works designed to mitigate HOL blocking [8, 35, 38, 43] often remain suboptimal when mobility causes rapid link fluctuations. By contrast, QUIC, operating over UDP, is better suited for real-time data since it supports both reliable and unreliable modes [28]. Based on this, MPQUIC [21,22,26,49,51] aggregates throughput across multiple paths but, as our experiments show, still suffers high latency when one path's signal drop suddenly. STORM overcomes this issue by making use of the characteristics of QUIC protocol and streaming media data.

MPQUIC Scheduling. A key element of MPQUIC is the scheduler, which allocates packets across subflows. Existing mechanisms, such as minRTT [21], or QoE-based feedback [22, 49], rely on historical metrics that fail to react quickly to sudden signal drops. Although these approaches boost throughput, they often overlook stringent latency requirements demanded by real-time streaming (e.g., VR or cloud gaming). Consequently, severe tail latencies persist under unstable networks. STORM addresses this gap by offering fast signal-awareness and scheduling decisions that reduce latency without sacrificing throughput.

7 Conclusion

This paper proposed STORM, an MPQUIC scheduler for quick streaming media transport under mobile networks. Based on the signal-watermark mechanism and reliability-aware scheduling strategy, STORM ensure low latency of streaming media applications when the multipath suffers "network storm". We have implemented STORM on real-life devices. Our evaluation shows that STORM reduces tail packet delay by 98.2% and improves the frame rate of streaming media by 1.95x under unstable networks when enabling STORM, compared to the state-of-the-art.

Acknowledgments

We thank USENIX ATC 2025 reviewers for their feedback and their help in improving the paper. This work was supported in part by the National Key R&D Program of China (2024YFB4504400), Dreams Foundation of Jianghuai Advance Technology Center (No. 2023-ZM01Z011), NSFC under Grant No. 62302169 and No. 62372184, Major Key Project of PCL under Grant PCL2024A06 and PCL2022A05, and Shenzhen Science and Technology Program under Grant RCJC20231211085918010.

References

[1] Ishan Aryendu, Sudhanshu Arya, and Ying Wang. Minimizing age of information: Adaptive spectrum sharing in ultra-reliable and low-latency evtol communications. In *2024 IEEE 25th International Symposium on a World of Wireless, Mobile and Multimedia Networks (WoWMoM)*, pages 57–63. IEEE, 2024.

[2] Arjun Balasingam, Manu Bansal, Rakesh Misra, Kanthi Nagaraj, Rahul Tandra, Sachin Katti, and Aaron Schulman. Detecting if lte is the bottleneck with bursttracker. In *The 25th Annual International Conference on Mobile Computing and Networking*, pages 1–15, 2019.

[3] E. Bandara, X. Liang, S. Shetty, R. Mukkamala, P. Foytik, N. Ranasinghe, and K. D. Zoysa. Octopus: privacy preserving peer-to-peer transactions system with interplanetary file system (ipfs). In *17th Annual International Conference on Mobile Systems, Applications, and Services*, pages 591–609, 2023.

[4] Qiang Chen and Changlong Li. Argus: Real-time hq video decoding with cpu coordinating on consumer devices. In *2024 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 2024.

[5] Sandesh Dhawaskar Sathyanarayana, Kyunghan Lee, Dirk Grunwald, and Sangtae Ha. Converge: Qoe-driven multipath video conferencing over webrtc. In *Proceedings of the ACM SIGCOMM 2023 Conference*, page 637–653. Association for Computing Machinery, 2023.

[6] Oussama El Marai and Tarik Taleb. Smooth and low latency video streaming for autonomous cars during handover. *Ieee Network*, 34(6):302–309, 2020.

[7] Yongjie Guan, Xueyu Hou, Nan Wu, Bo Han, and Tao Han. Metastream: Live volumetric content capture, creation, delivery, and rendering in real time. In *Proceedings of the 29th Annual International Conference on Mobile Computing and Networking*, 2023.

[8] Yihua Ethan Guo, Ashkan Nikravesh, Z Morley Mao, Feng Qian, and Subhabrata Sen. Accelerating multipath transport through balanced subflow completion. In *Proceedings of the 23rd Annual International Conference on Mobile Computing and Networking*, pages 141–153, 2017.

[9] Philipp Hock, Sebastian Benedikter, Jan Gugenheimer, and Enrico Rukzio. Carvr: Enabling in-car virtual reality entertainment. In *CHI conference on human factors in computing systems*, pages 4034–4044, 2017.

[10] Alibaba Inc. Xquic. https://github.com/alibaba/xquic, 2023.

[11] Apple Inc. Introducing Apple Vision Pro: Apple's first spatial computer. https://www.apple.com/newsroom/2023/06/introducing-apple-vision-pro/, 2023.

[12] Google Inc. Fuchsia os. https://fuchsia.dev/, 2022.

[13] HUAWEI Inc. Harmony os. https://consumer.huawei.com/en/harmonyos/, 2024.

[14] Jana Iyengar and Martin Thomson. QUIC: A UDP-Based Multiplexed and Secure Transport. https://www.rfc-editor.org/rfc/rfc9000.html, 2021.

[15] Han-Shin Jo, Young Jin Sang, Ping Xia, and Jeffrey G Andrews. Heterogeneous cellular networks with flexible cell association: A comprehensive downlink sinr analysis. *IEEE Transactions on Wireless Communications*, 11(10):3484–3495, 2012.

[16] HyunJong Lee, Jason Flinn, and Basavaraj Tonshal. Raven: Improving interactive latency for the connected car. In *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking*, pages 557–572, 2018.

[17] Changlong Li, Yu Liang, Rachata Ausavarungnirun, Zongwei Zhu, Liang Shi, and Chuan Jason Xue. Ice: Collaborating memory and process management for user experience on resource-limited mobile devices. In *Proceedings of the Eighteenth European Conference on Computer Systems*, pages 79–93, 2023.

[18] Changlong Li, Liang Shi, and Chun Jason Xue. Mobileswap: Cross-device memory swapping for mobile devices. In *58th ACM/IEEE Design Automation Conference (DAC)*, 2021.

[19] LinkedIn. Internet celebrity economy market size, research. https://www.linkedin.com/pulse/internet-celebrity-economy-market-size-research-1v3zf, 2023.

[20] LinkedIn. Video conference: what best practices reducing latency video conferencing. https://www.linkedin.com/advice/3/what-best-practices-reducing-latency-video-conferencing, 2023.

[21] Yanmei Liu, Yunfei Ma, Quentin De Coninck, Olivier Bonaventure, Christian Huitema, and Mirja Kühlewind. Multipath Extension for QUIC. https://datatracker.ietf.org/doc/draft-ietf-quic-multipath/05/, 2023.

[22] Gerui Lv, Qinghua Wu, Yanmei Liu, Zhenyu Li, Qingyue Tan, Furong Yang, Wentao Chen, Yunfei Ma, Hongyu Guo, Ying Chen, et al. Chorus: Coordinating mobile multipath scheduling and adaptive video streaming. In *30th Annual International Conference on Mobile Computing and Networking*, pages 246–262, 2024.

[23] Mpshell. https://github.com/ravinet/mahimahi/tree/old/mpshell_scripted, 2023.

[24] Arvind Narayanan, Eman Ramadan, Jason Carpenter, Qingxu Liu, Yu Liu, Feng Qian, and Zhi-Li Zhang. A first look at commercial 5g performance on smartphones. In *Proceedings of The Web Conference 2020*, pages 894–905, 2020.

[25] Yunzhe Ni, Feng Qian, Taide Liu, Yihua Cheng, Zhiyao Ma, Jing Wang, Zhongfeng Wang, Gang Huang, Xuanzhe Liu, and Chenren Xu. {POLYCORN}: Data-driven cross-layer multipath networking for high-speed railway through composable schedulerlets. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 1325–1340, 2023.

[26] Yunzhe Ni, Zhilong Zheng, Xianshang Lin, Fengyu Gao, Xuan Zeng, Yirui Liu, Tao Xu, Hua Wang, Zhidong Zhang, Senlang Du, et al. Cellfusion: Multipath vehicle-to-cloud video streaming with network coding in the wild. In *Proceedings of the ACM SIGCOMM 2023 Conference*, pages 668–683, 2023.

[27] Stephen Palmisano, Luke Szalla, and Juno Kim. Monocular viewing protects against cybersickness produced by head movements in the oculus rift. In *Proceedings of the 25th ACM Symposium on Virtual Reality Software and Technology*, pages 1–2, 2019.

[28] Tommy Pauly, Eric Kinnear, and David Schinazi. An Unreliable Datagram Extension to QUIC. https://www.rfc-editor.org/rfc/rfc9221.html, 2022.

[29] Michele Polese, Federico Chiariotti, Elia Bonetto, Filippo Rigotto, Andrea Zanella, and Michele Zorzi. A survey on recent advances in transport layer protocols. *IEEE Communications Surveys & Tutorials*, 21(4):3584–3608, 2019.

[30] Feng Qian, Bo Han, Qingyang Xiao, and Vijay Gopalakrishnan. Flare: Practical viewport-adaptive 360-degree video streaming for mobile devices. In *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking*, pages 99–114, 2018.

[31] Costin Raiciu, Christoph Paasch, Sebastien Barre, Alan Ford, Michio Honda, Fabien Duchene, Olivier Bonaventure, and Mark Handley. How hard can it be? designing and implementing a deployable multipath TCP. In *Symposium on Networked Systems Design and Implementation (NSDI)*, pages 399–412, 2012.

[32] Elizaveta Rastorgueva-Foi, Mário Costa, Mike Koivisto, Kari Leppänen, and Mikko Valkama. User positioning in mmw 5g networks using beam-rsrp measurements and kalman filtering. In *2018 21st International Conference on Information Fusion (FUSION)*, pages 1–7, 2018.

[33] Ringmaster. https://github.com/microsoft/ringmaster?tab=readme-ov-file, 2023.

[34] Michael Rudow, Francis Y Yan, Abhishek Kumar, Ganesh Ananthanarayanan, Martin Ellis, and KV Rashmi. Tambur: Efficient loss recovery for videoconferencing via streaming codes. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 953–971, 2023.

[35] Swetank Kumar Saha, Shivang Aggarwal, Rohan Pathak, Dimitrios Koutsonikolas, and Joerg Widmer. Musher: An agile multipath-tcp scheduler for dual-band 802.11 ad/ac wireless lans. In *The 25th Annual International Conference on Mobile Computing and Networking*, pages 1–16, 2019.

[36] Heiko Schwarz, Detlev Marpe, and Thomas Wiegand. Overview of the scalable video coding extension of the h. 264/avc standard. *IEEE Transactions on circuits and systems for video technology*, pages 1103–1120, 2007.

[37] Ranya Sharma, Nick Feamster, and Marc Richardson. A longitudinal study of the prevalence of wifi bottlenecks in home access networks. In *Proceedings of the 2024 ACM on Internet Measurement Conference*, pages 44–50, 2024.

[38] Hang Shi, Yong Cui, Xin Wang, Yuming Hu, Minglong Dai, Fanzhao Wang, and Kai Zheng. {STMS}: Improving {MPTCP} throughput under heterogeneous networks. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 719–730, 2018.

[39] Sam Son, Seung Yul Lee, Yunho Jin, Jonghyun Bae, Jinkyu Jeong, Tae Jun Ham, Jae W. Lee, and Hongil Yoon. Asap: Fast mobile application switch via adaptive prepaging. In *USENIX Annual Technical Conference (USENIX ATC)*, pages 365–380, 2021.

[40] Sandeep Tata, Alexandrin Popescul, Marc Najork, Mike Colagrosso, Julian Gibbons, Alan Green, Alexandre Mah, et al. Quick access: building a smart experience for google drive. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1643–1651, 2017.

[41] Android Team. Telephonymanager. https://developer.android.google.cn/reference/android/telephony/TelephonyManager, 2022.

[42] Android Team. Wifimanager. https://developer.android.google.cn/reference/android/net/wifi/WifiManager, 2022.

[43] Chengke Wang, Hao Wang, Feng Qian, Kai Zheng, Chenglu Wang, Fangzhu Mao, Xingmin Guo, and Chenren Xu. Experience: A three-year retrospective of large-scale multipath transport deployment for mobile applications. In *Proceedings of the 29th Annual International Conference on Mobile Computing and Networking*, pages 1–15, 2023.

[44] Shibo Wang, Shusen Yang, Xiao Kong, Chenglei Wu, Longwei Jiang, Chenren Xu, Cong Zhao, Xuesong Yang, Jianjun Xiao, Xin Liu, et al. Pudica: Toward near-zero queuing delay in congestion control for cloud gaming. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2024.

[45] Yuxin Wang, Zunlei Feng, Haofei Zhang, Yang Gao, Jie Lei, Li Sun, and Mingli Song. Angle robustness unmanned aerial vehicle navigation in gnss-denied scenarios. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pages 10386–10394, 2024.

[46] Keith Winstein, Anirudh Sivaraman, and Hari Balakrishnan. Stochastic forecasts achieve high throughput and low delay over cellular networks. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 459–471, 2013.

[47] Weixing Xue, Weining Qiu, Xianghong Hua, and Kegen Yu. Improved wi-fi rssi measurement for indoor localization. *IEEE Sensors Journal*, 17(7):2224–2230, 2017.

[48] Xin-Wei Yao, Wan-Liang Wang, Shuang-Hua Yang, Yue-Feng Cen, Xiao-Min Yao, and Tie-Qiang Pan. Ipb-frame adaptive mapping mechanism for video transmission over ieee 802.11 e wlans. *ACM SIGCOMM Computer Communication Review*, 44(2):5–12, 2014.

[49] Zhilong Zheng, Yunfei Ma, Yanmei Liu, Furong Yang, Zhenyu Li, Yuanbo Zhang, Jiuhai Zhang, Wei Shi, Wentao Chen, Ding Li, Qing An, Hai Hong, Hongqiang Harry Liu, and Ming Zhang. Xlink: Qoe-driven multi-path quic transport in large-scale video services. In *ACM SIGCOMM*, page 418–432, 2021.

[50] Meng Zili, Wang Tingfeng, Shen Yixin, Wang Bo, Xu Mingwei, Han Rui, Liu Honghao, Arun Venkat, Hu Hongxin, and Wei Xue. Enabling high quality real-time communications with adaptive frame-rate. In *20th USENIX Symposium on Networked Systems Design and Implementation*, 2023.

[51] Xutong Zuo, Yong Cui, Xin Wang, and Jiayu Yang. Deadline-aware multipath transmission for streaming blocks. In *IEEE INFOCOM 2022-IEEE Conference on Computer Communications*, pages 2178–2187, 2022.

A Appendix

A.1 Deadline Setting Based on Application Real-Time Requirements

The microbenchmark experiments in Section 5.2 use a 300 ms end-to-end deadline. This value does not restrict STORM to traffic that must meet precisely 300ms. It simply reflects the minimum latency budget frequently cited for interactive video conferencing [20], as already noted in the footnote of Section 2.3.2. In practice, every unit of data that an application hands to STORM carries its own deadline in the metadata. The API function *STORM_send* exported by STORM therefore includes an explicit field, `deadline_us`, whose value is filled in by the application on a per-message basis and expressed in microseconds. Because the STORM reads this field directly, developers can tighten or relax the requirement for each video frame, audio chunk, or control message at run time without recompiling or reconfiguring the STORM.

Section 5.5 demonstrates this flexibility by evaluating three representative applications, each configured with a deadline that matches its real-time requirements. The video-conference prototype operates under a strict 150ms budget. The low-latency live-streaming scenario employs a 500ms budget, reflecting industry practice for end-to-end latency. The immersive 360° video workload targets an 80ms motion-to-photon limit. Whether a batch of packets meets the deadline in practice depends on the bandwidth and one-way delay of the active paths. STORM optimizes scheduling based on the current path conditions and operates within each path's inherent upper limits.

STORM is therefore deadline-agnostic. The 300ms setting in the microbenchmark experiments serves only as a conservative experimental baseline. When an application supplies a tighter or looser deadline, the same scheduling strategy remains valid and optimises scheduling under the newly specified deadline.

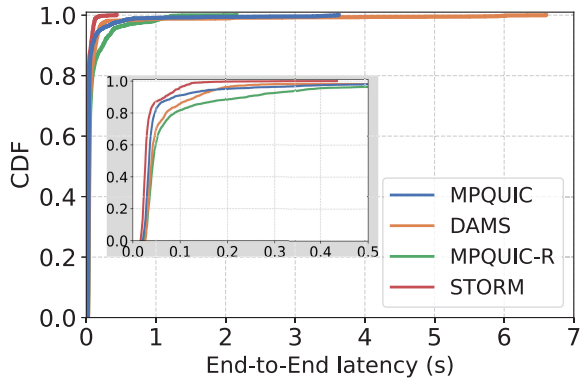


Figure 19: End-to-End latency distribution

A.2 Absolute End-to-End Latency Distribution

Section 5.5.2 evaluates the packet delay reduction achieved by STORM at selected percentile points. To obtain a more detailed view, we also collected absolute end-to-end latency samples under the same testbed configuration and plotted their cumulative distribution functions (CDFs). For clarity the figure retains only four representative solutions: STORM, MPQUIC, MPQUIC-R, and DAMS. Curves for the remaining solutions overlap and are therefore omitted.

Figure 19 indicates that 90% of STORM's latency measurements fall below 68.2ms and 99% remain under 129.8ms. MPQUIC, the best of the other solutions, records 90% of latencies under 89.8ms and 99% under 758.6ms. Hence, STORM significantly reduces both median and tail end-to-end latency and yields a much tighter distribution, which benefits real-time media streams.

Figure 19 also reveals a contrasting outcome for MPQUIC-R. Among the four evaluated solutions, MPQUIC-R exhibits the highest median end-to-end latency, clearly highlighting the latency cost introduced by its fully reliable delivery strategy. However, its tail end-to-end latency stops at 2.15s, well below the 3.62s of MPQUIC and the 6.60s of DAMS. This observation confirms that mutual blockage between data with different reliability requirements imposes a severe performance penalty, a problem that STORM mitigates through its RAS component.

Internet Connection Splitting: What's Old is New Again

Gina Yuan
Stanford University

Thea Rossman
Stanford University

Keith Winstein
Stanford University

Abstract

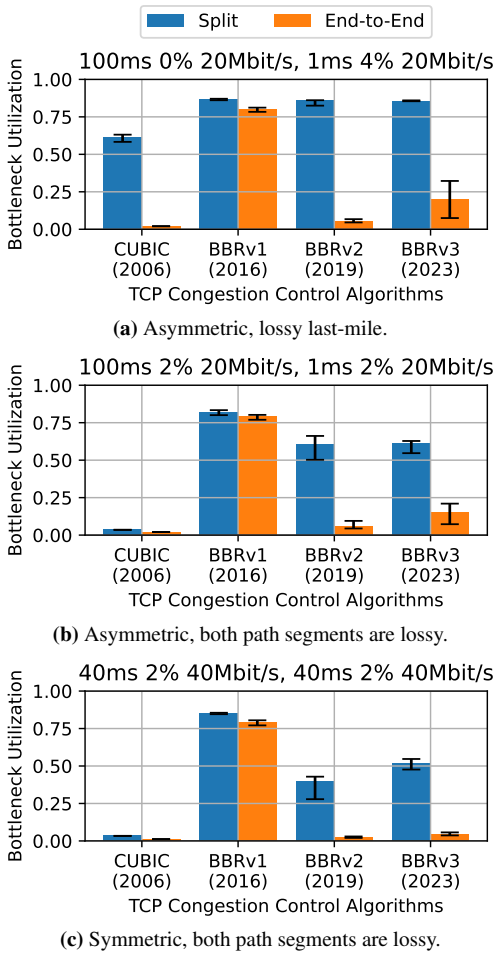
In the 1990s, many networks deployed performance-enhancing proxies (PEPs) that transparently split TCP connections to aid performance, especially over lossy, long-delay paths. Two recent developments have cast doubts on their relevance: the BBR congestion-control algorithm, which de-emphasizes loss as a congestion signal, and the QUIC transport protocol, which prevents transparent connection-splitting yet empirically matches or exceeds TCP's performance in wide deployment, using the same congestion control.

In light of this, are PEPs obsolete? This paper presents a range of emulation measurements indicating: "probably not." While BBR's original 2016 version didn't benefit markedly from connection-splitting, more recent versions of BBR *do* and, in some cases, even *more* so than earlier "loss-based" congestion-control algorithms. We also find that QUIC implementations of the "same" congestion-control algorithms vary dramatically and further differ from those of Linux TCP—frustrating head-to-head comparisons. Notwithstanding their controversial nature, our results suggest that PEPs remain relevant to Internet performance for the foreseeable future.

1 Introduction

A stylized history of connection-splitting over the Internet could go roughly as follows: In the 1970s, TCP was designed as a host-to-host transport for flow-controlled reliable byte streams. In the late 1980s, TCP implementations added congestion control [24] to respond to network-path limitations and contention, with indications of packet loss as the primary congestion signals. As Internet use exploded in the 1990s, it became apparent that contemporary TCP implementations performed poorly over long paths with packet loss. By the late 1990s, many network operators deployed commercial "TCP accelerators," also known as "performance-enhancing proxies" [17, 21], to improve customers' TCP throughput. These proxies transparently split a TCP connection in two, providing an intermediate point of acknowledgment and retransmission and transforming an end-to-end TCP connection into two independent connections.

PEPs were marketed as improving users' throughput, especially where a PEP splits a high-delay, lossy path into two segments, each with less delay or less loss. Past surveys found connection-splitting PEPs deployed on large numbers of wireless and satellite networks [17, 21]. But for almost as long as these PEPs have existed, purists have criticized their alleged benefits as exaggerated or unfair, sensitive to the details of an endpoint's congestion-control scheme, contrary to end-to-end arguments [42], and likely to lead to ossification [13, 38].



(a) Asymmetric, lossy last-mile.

(b) Asymmetric, both path segments are lossy.

(c) Symmetric, both path segments are lossy.

Figure 1: Three classes of network settings in which the throughput of a long-lived data transfer using BBRv3 benefits significantly from connection-splitting. While the throughput of TCP using BBRv1 may not have benefitted from a PEP, as the BBR algorithm has evolved over time, it has also made splitting pertinent again. In two of these classes, split BBRv3 even achieves high bottleneck link rate utilization where split CUBIC does not. IQR of $n = 20$.

Two more-recent developments have put PEPs on the back foot. The QUIC transport protocol, standardized in 2021 [23], is encrypted end-to-end and, unlike TCP, its connections can't be split by a transparent proxy—but its wide deployment

by major traffic sources (particularly Google) in the 2010s didn't produce a measured reduction in performance, compared with TCP connections that do benefit from PEPs in the field [31]. If PEPs had really been aiding TCP connections all this time, QUIC could have been expected to experience lower sustained throughput over such paths (compared with TCP connections between the same endpoints, using the same congestion-control schemes).

Furthermore, the BBR congestion-control scheme [8], first released in 2016, now controls a large percentage of Internet traffic (both TCP and QUIC) [51] and puts less emphasis on packet loss as a congestion signal than older schemes in wide deployment, such as CUBIC. Because much of PEPs' benefit comes from splitting up long, lossy paths for the benefit of "loss-based" schemes, it has been suggested that BBR has led to a sharp decline in the use of PEPs [25].

Do BBR and QUIC make connection-splitting obsolete? In this paper, we present findings that suggest that neither of these developments should be regarded as reliable; Connection-splitting may yet play a performance-enhancing role on the Internet, despite its controversial nature.

We conduct a measurement study in emulation to investigate what kinds of congestion control schemes, network scenarios, and transport protocols may still experience increased sustained throughput from connection-splitting. We aid our analysis with a *split throughput heuristic* for predicting the throughput of a split connection, based on the measured end-to-end throughput of each segment on the split path. We aim to (1) explore a wide parameter space of network settings and congestion control schemes and (2) reason about encrypted transport protocols (QUIC) that we cannot directly measure.

We present three major findings:

Finding 1: Splitting has become significantly more beneficial to TCP BBR since it was initially released in 2016. We confirm that BBR's initial "v1" release does not benefit markedly from splitting, unlike earlier congestion-control schemes. But as the BBR algorithm has evolved into BBRv2 in 2019 and now BBRv3 in 2023, it has also evolved to behave more conventionally in the sense that it benefits from being split—just like more traditional, loss-based congestion control schemes such as CUBIC (Fig. 1).

The regression in split throughput follows directly from the regression in end-to-end throughput, via the heuristic. While BBRv1 nearly always achieves full utilization, BBRv3 sees lower utilization in lossy settings, leaving more to gain from splitting. The end-to-end regression is because BBRv3 has increased its loss sensitivity to coexist fairly with "loss-based" schemes such as Reno and CUBIC [6, 50, 54]. Thus, we find that BBR and its split behavior continue to evolve, and that the model-based BBR algorithm has *not* made the performance enhancements of connection-splitting obsolete.

Finding 2: There exist classes of network paths where TCP BBR would significantly benefit from splitting but TCP CUBIC would not. Splitting not only benefits BBRv3 in all the same scenarios as CUBIC (in terms of long-lived throughput), it also benefits BBRv3 in *new* scenarios. In particular, there exist classes of network settings where the CCA achieves a very low bottleneck link rate utilization *without* a PEP, but a high bottleneck link rate utilization *with* a PEP.

More specifically, in addition to edge deployments with a lossy last-mile (Fig. 1a), BBRv3 can also benefit from splitting in scenarios where there is loss on both path segments (Fig. 1b), and when the connection-splitter is located farther from the edge (Fig. 1c). This suggests that in these network classes, simple split BBRv3 can replace the proprietary protocols that have traditionally been used to address middle-mile loss, particularly in satellite and wireless ad-hoc networks [2, 17, 39]. We should continue to evaluate the behavior of split CCAs in new types of networks, especially in space.

Finding 3: Implementations of the "same" congestion-control schemes in QUIC vary significantly, and further differ those of Linux TCP—frustrating attempts to directly compare performance between QUIC and TCP. In an initial study of three QUIC implementations, we find that the end-to-end behaviors of the same congestion control schemes vary significantly by implementation. The implementations vary in both baseline performance and sensitivity to loss and delay. By applying the split throughput heuristic, we find that some CUBIC implementations can actually benefit in the same classes of network paths as TCP BBRv3. We also find that BBR is challenging to implement, with various BBRv3 implementations exhibiting non-uniform end-to-end behavior and thus no clear trend in split behavior. We argue that the benefits of in-network assistance should be considered along with not just a CCA, but their specific implementations.

Our initial evaluation suggests that the performance improvements of connection-splitting remain relevant in the context of today's model-based congestion control algorithms, encrypted transport protocols, and satellite and wireless ad-hoc network settings. As congestion-control schemes continue to evolve, we urge researchers to refer to them by algorithm/implementation/version, not just "BBR" or even "QUIC BBRv1". We urge the community to create a performance test suite for an implementation to be able to claim that it conforms to a particular congestion-control standard. Finally, we hope that this work motivates the community to pursue protocol-agnostic PEPs that achieve the performance benefits of in-network assistance without the ossification downsides.

2 Background

In this section, we provide additional context on some concepts that we refer to throughout the text.

Performance-enhancing proxies. The idea of in-network assistance has long brought pain to the hearts of Internet researchers, protocol designers, and network operators. In

particular, the word "middleboxes" is often associated with NATs, firewalls, and other policy enforcers that modify or inspect data in ways that break end-to-end behavior. A substantial percentage of Internet paths are affected by feature or protocol-breaking policies of middleboxes [13].

In contrast, we refer specifically to the type of assistance provided by performance-enhancing "proxies", or PEPs, that transparently split a TCP connection in two [17]. While PEPs still interfere with end-to-end transport mechanisms, their goal is to enhance the user experience by maximizing performance, as opposed to enforce security or routing policies.

Connection-splitting PEPs can enhance performance in several ways. By providing an intermediate point of acknowledgment and retransmission, PEPs reduce the length of the feedback loop for signals of loss and congestion. The PEP also allows each side of the split connection to better optimize congestion control and flow control for network conditions local to the path segment. We can expect PEPs to impact a variety of performance metrics, including startup time, packet jitter, retransmission overheads, and more. In this paper, we focus only on the impact of connection-splitting on the *sustained throughput* of long-lived data transfers.

Congestion control. Congestion control was originally deployed in the 1980s to manage the explosive growth of the Internet [24]. Foundational algorithms such as slow start and fast retransmit eventually became part of the Tahoe congestion control scheme. Variants such as NewReno and CUBIC emerged over time to improve performance with fairness [18]. CUBIC is the default congestion control module for TCP in the Linux kernel, and widely deployed. It is the primary "loss-based" congestion control scheme that we evaluate.

The word "TCP" is sometimes used synonomously with the congestion control algorithms that it implements. In reality, the two are distinct, though deeply intertwined. For example, the QUIC transport protocol implements many of the same algorithms as TCP to manage network traffic, even though it runs over UDP [23]. In this paper, we refer to "TCP" and "QUIC" as the transport protocols, "Linux TCP" as the implementation of TCP in the Linux kernel, and congestion control schemes as the particular manifestations of a congestion control algorithm in a transport protocol implementation.

BBR. Today, the focus on congestion control has shifted towards BBR. BBR is sometimes referred to as a "model-based" congestion control scheme because it relies on a fundamentally different approach of modeling network bandwidth and round-trip time. It does not use packet loss as the primary signal for congestion. Google initially presented BBR in 2016 as addressing the problems of CUBIC in high-speed networks [7].

Over time, BBR encountered controversy for its unfairness towards loss-based schemes [4, 40, 50]. In response, "v1" of BBR has now evolved into BBRv2 in 2019 and BBRv3 in 2023, driven by efforts at Google. Today, Google recommends that BBRv1 and BBRv2 be deprecated in favor of BBRv3 [6].

BBR is also widely deployed [15, 51]. Google has contributed BBR implementations to the Linux kernel, and Amazon, Akamai, Meta, and Cloudflare CDNs all use BBRv1 for TCP. At Google, BBRv3 is used in all google.com and YouTube public Internet traffic with TCP, and is currently being A/B tested against BBRv1 in Google QUIC traffic [6]. It is estimated that 40% of traffic used BBR in 2019 [37].

Since 2017, there have been ongoing efforts to standardize BBR in the IETF [9]. However, progress has been slow. The Internet draft goes years or months at a time without an update at the mercy of Google contributors. While companies have been quick to adopt BBR in Linux TCP, adoption in QUIC has been slower. Anecdotally, some have resorted to reverse engineering the Linux implementation, and some such as Cloudflare, Meta, and Cisco are actively experimenting with variants of BBRv2+ in their QUIC stacks [15].

QUIC. The QUIC transport protocol was originally developed by Google in 2012 to improve performance for HTTPS traffic and to enable the rapid evolution of transport mechanisms. QUIC is implemented over UDP, and moves congestion control development from kernel space to user space.

As of 2021, QUIC is now a standards-track RFC [23] with widespread deployment and interoperability testing [46]. It is estimated that 8.4% of global websites support QUIC [49], many of them major traffic sources such as Cloudflare. Still, many high-speed flows remain on TCP due to better TCP hardware capabilities and "per-origin" caching, which complicates multi-CDN deployments that mix protocols.

Evaluations of QUIC at the Internet scale generally show improved performance compared to TCP, except in satellite networks. Features like zero-RTT connection establishment and stream multiplexing enable measured improvements in search latency, rebuffer rate, and other video playback metrics at Google [31]. Satellite network operators, on the other hand, experience strife when it comes to QUIC. Several studies have measured the negative impact of encryption on performance in split satellite environments [2, 26, 30, 32].

3 The Split Throughput Heuristic

When initially considering which network scenarios to evaluate in our emulation study, we discovered it to be non-trivial to pick scenarios that would lead to a general understanding of the throughput of a congestion control scheme in a split setting. It is challenging to select which network settings and combinations of path segments to evaluate, as the parameter space for combinations of path segments can be quite large. Also, not all CCA implementations can be trivially split.

For this paper, we refer to *throughput* as the throughput of a long-lived data transfer in an emulated network, using a single flow. We do not evaluate other metrics such as startup time, retransmission overheads, or short-lived data transfers, which have been of interest to PEPs and could also be impacted by connection-splitting. When discussing throughput, we refer

to *end-to-end throughput* as the throughput of an end-to-end connection, and *split throughput* as the throughput of the same connection when there is a connection-splitting PEP.

We realized that while split throughput is not necessarily well understood for any combination of CCA, transport protocol, and network setting, the end-to-end throughput often *is*. We make the following insight based on an ideal setting:

> **Split Throughput Heuristic:** We can estimate the long-lived "split throughput" of a connection by measuring the long-lived "end-to-end throughput" on each segment of the split path and taking the minimum.

Figure 2: A heuristic for evaluating split behavior from end-to-end behavior in an emulated network.

Thus if we know the throughput of the much smaller parameter space of end-to-end connections, we can easily derive:

1. The split throughput of a CCA on a network path composed of two path segments,
2. The end-to-end throughput on the network path,
3. The expected throughput benefit of using a connection-splitting PEP on the network path.

Furthermore, it is impossible to quantitatively evaluate the split throughput of encrypted transport protocols without creating a custom and explicit connection-splitting PEP. As a result, it is challenging to establish a baseline for the throughput that is potentially achievable by QUIC with in-network assistance, leaving many studies to compare QUIC only to split TCP [2, 48, 52]. The split throughput heuristic allows us to estimate the throughput of encrypted transport protocols using knowledge about its end-to-end throughput, without actually splitting the connection.

In §6, we discuss potential sources of error in the heuristic, including the lack of consideration of the queue configuration on the proxy or the placement of the bottleneck link relative to the data sender. However, we believe there is a design tradeoff in accuracy versus simplicity.

In the rest of this section, we first describe how to use this heuristic to analyze the split throughput benefit in a single network setting, without having to run an emulation with a connection splitter. Then we describe how we cache emulation results for an end-to-end parameter space to enable a more open-ended analysis of CCAs over a variety of networks. Finally, we discuss the limitations of our methodology using the heuristic, and propose possible extensions.

3.1 Analyzing Split Throughput Benefit in a Single Network Setting

Given a network model of the two path segments that compose an end-to-end network path, we would like to be able to estimate the expected throughput benefit of using a connection-splitting PEP between the two segments. In our emulation study, our simplified network model consists of three parameters: delay, bandwidth, and loss.

```
class NetworkModel:
    def __init__(self, delay, bw, loss):
        self.delay = delay
        self.bw = bw
        self.loss = loss

def compose(s1: NetworkModel,
            s2: NetworkModel) -> NetworkModel:
    delay = s1.delay + s2.delay
    bw = min(s1.bw, s2.bw)
    loss = s1.loss * (1-s1.loss)*s2.loss
    return NetworkModel(delay, bw, loss)

def throughput(s: NetworkModel) -> float:
    return run_emulation(s)
```

Listing 1: An interface for modeling a network path and estimating throughput. The `compose()` function models an end-to-end network path from two path segments. The `throughput()` function obtains a throughput measurement for a path segment.

```
def pred_split_throughput(s1: NetworkModel,
                          s2: NetworkModel) -> float:
    return min(throughput(s1), throughput(s2))

def pred_e2e_throughput(s1: NetworkModel,
                        s2: NetworkModel) -> float:
    s = compose(s1, s2)
    return throughput(s)
```

Listing 2: Functions that apply our simplified network model and the split throughput heuristic (Fig. 2) to predict split and end-to-end throughput, using the interface in Listing 1.

We explicitly define the interface we use to query split and end-to-end throughput in Listings 1 and 2. In addition to the network model, we implement functions to `compose()` a model of an end-to-end network path from two path segments and to query the end-to-end `throughput()` of a segment.

The `compose()` function. Since our network model consists of three parameters, we describe how to compose each of these parameters to model the end-to-end network path. Bandwidth is the minimum of the two bandwidths, which is the bottleneck. (We may also refer to "bandwidth" as "link rate.") Delay is just additive. If we think of loss as independent random loss, then the composition of the two is $loss = loss1 + (1 - loss1) \cdot loss2 = loss1 + loss2 - loss1 \cdot loss2$.

Note that many combinations of path segments can compose to the same end-to-end network path. This makes sense because regardless of where on the network path loss occurs or which path segment has the bottleneck bandwidth, from an end-to-end perspective, the network properties look the same.

Parameter	Values	Unit
Bandwidth	[10, 20, 30, 40, 50]	Mbit/s
Delay	[1, 20, 40, 60, 80, 100]	ms
Loss	[0, 1, 2, 3, 4]	%

Table 1: Network parameter space for the $5 \cdot 6 \cdot 5 = 150$ combinations of network settings that we analyze in §5. These values attempt to realistically reflect network settings in which we'd see a PEP.

The `throughput()` function. This is the throughput of a long-running HTTPS connection in a `mininet` emulation. The emulated network is parameterized by the network model of a single path segment, and the HTTPS implementation uses the congestion control scheme of interest. We describe additional details about the emulation in §4.

Calculating the split improvement. Listing 2 demonstrates how we can estimate the split throughput benefit using the interface in Listing 1. We `pred_split_throughput()` by taking the minimum measured throughput on each network path segment, and `pred_e2e_throughput()` by composing the path segments into an end-to-end network path and using the measured throughput of that network path. The split throughput benefit is just how much the split throughput has improved (if it has) relative to the end-to-end throughput.

3.2 Caching Measurements for Analysis

While we can now predict split throughput for congestion control schemes which are not easily splittable, evaluating split throughput on a variety of network settings still requires running emulations for each query. In order to speed up our analysis, we select a parameter space of delay, bandwidth, and loss within our network model and cache the end-to-end emulation results for this parameter space. We describe our choice of parameter space and other modifications as follows:

Selecting a parameter space. We select a range of values we thought would realistically reflect network settings in which we'd see a PEP (Table 1). We select propagation delays from 1 ms for a Wi-Fi last hop to 100 ms to reflect the longer RTTs of a satellite connection. Bottleneck bandwidths range from 10 to 50 Mbit/s for a single connection, and are within the CPU constraints of emulation. Random loss rates range from 0% for a stable connection to 4% for random loss caused by e.g., wireless interference and extraterrestrial weather.

Modifying the `compose()` function. We make some subtle changes to the compose function to keep composed network models within our parameter space. In some sense, each parameter e.g., delay, can be thought of as an algebraic group where the operator function is defined in `compose()`.

For path segments with small delays (1 ms), we consider the composed path segment to just have the delay of the longer segment, so we can have segments with 1 ms delay. For loss, note that when loss rates are small, $loss1 \cdot loss2$ is negligibly small. In our case, the loss is always below 4%, so we omit the term in the composition and loss becomes additive.

Collecting data efficiently. For some CCAs, we expect a large number of network settings in the parameter space to have extremely low throughput. Thus we set a utilization threshold of 5% of the link rate below which we are not interested in the exact throughput of that network setting.

To more efficiently build the cache, we run a search algorithm on the parameter space to explore faster network settings first. The initial network setting we evaluate has the lowest delay, bandwidth, and loss. We explore adjacent data points if and only if the current data point has not timed out.

3.3 Limitations

Although our methodology is more efficient for evaluating many network settings and enables the exploration of theoretical scenarios, caching measurements in emulation still takes non-negligible time. A much faster method to estimate sustained throughput would be to use a theoretical model, such as the TCP macroscopic model for AIMD schemes [36]. However, this method does not reflect the nuances of implementation, and as Mathis states himself, new models are needed for BBR and modern paradigms [34, 35].

Another limitation is our simplified network model and emulation testbed. One could incorporate more parameters into the network model, such as the fluctuating bandwidths of cellular networks [20], or how BBR incorporates ACK aggregation [5]. Another idea is to generate end-to-end measurements from real testbeds instead of emulation. We believe these challenges are not limited to emulation studies on split performance, and hope to use the much more well-understood space of end-to-end behavior to extrapolate split behavior.

We acknowledge that we only test connections in a single-flow environment. This limits what we can say about inter-flow fairness, but we can still infer some aspects from how congestion-control schemes respond to loss and delay in isolation, a classical concept known as "TCP friendliness" [19]. While split connections reduce retransmissions by buffering on the path, their high throughput may also make them unfair. There may also be different fairness implications when combining split with end-to-end connections. If every connection is split, then we refer to other studies for understanding fairness at scale for each side of the concatenation [40].

We also acknowledge that TCP is not limited to bulk data transfers, and that it would be valuable to study other metrics that could be impacted by connection-splitting, such as startup time, packet jitter, and retransmission overheads.

Finally, our heuristic makes the simplified assumption that we can identify the bottleneck path segment based on the minimum measured throughput. In reality, the ability of a data sender to sustain that throughput depends on its ability to saturate the send buffer. This is particularly true for data senders farther along the network path, as their behavior will depend on the queue configuration and the burstiness of previous

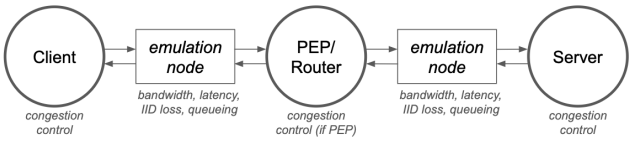


Figure 3: Two-segment network topology in `mininet`. The middle node splits the path into two segments with different properties.

CCA	Implementation
BBRv3	Linux TCP v6.4.0+ `google-bbr/v3` fork
BBRv2	Linux TCP v6.4.0+ `google-bbr/v2alpha` fork
BBRv1	Linux TCP v5.15.0-122-generic
CUBIC	Linux TCP v5.15.0-122-generic
BBRv3	Google `quiche` v131.0.6728.1
BBRv1	Google `quiche` v131.0.6728.1
CUBIC	Google `quiche` v131.0.6728.1
BBRv2	Cloudflare `quiche` v0.14
BBRv1	Cloudflare `quiche` v0.14
CUBIC	Cloudflare `quiche` v0.14
BBRv2	IETF `picoquic 29c7c53`
BBRv1	IETF `picoquic 29c7c53`
CUBIC	IETF `picoquic 29c7c53`

Table 2: The congestion control schemes and transport protocol implementations we evaluate in the measurement study.

senders. We explore this more in §6.

4 Measurement Methodology

We want to evaluate different congestion control schemes in a variety of network settings. To do this, we run emulation experiments in `mininet` with simple HTTPS clients and servers to measure the throughputs of long-lived data transfers. In this section, we describe our emulated network configurations, HTTPS endpoints and PEP, and other specifications.

Network configuration. We used two linear network topologies: a one-segment topology for caching end-to-end measurements and a two-segment topology for evaluation with a connection-splitting PEP (Fig. 3). Both have a client and server node at each end. The two-segment topology additionally has a router node in between. Each path segment has a bridging node to emulate network properties on the link.

We parameterize each path segment in three dimensions: delay, bandwidth, and a random loss rate. We configure the network properties on the bridging nodes' egress interfaces, using `tc-netem` to set delay and random loss, and `tc-htb` to set bandwidth. Additionally, we use `tc-qdisc` to configure the queues to use RED[1], which is the source of congestive loss. Each link is symmetric in the uplink and downlink directions. For some versions of BBR, we set an `fq` qdisc on the host nodes' egress interfaces for pacing.

[1]We apply RED and not droptail because it gives more continuous feedback about loss for congestion control, and is commonly used in core routers. We also do not need the multi-flow and low-delay properties of other queue disciplines. We use RED in `adaptive harddrop` mode with a maximum queuing delay of ≈ 1 BDP. Exact parameters are available in the code.

Host configuration. We create simple HTTPS wrappers around each transport protocol implementation to evaluate the congestion control schemes in Table 2. The TCP endpoints use the Python `http` module. For the QUIC implementations, we modify comparable client/server applications from each repository. With the exception of enabling TCP pacing where required by BBR, we use "default values" for tunable parameters, considering these to be part of each implementation.

In TCP, we set the CCA by using a specific Linux kernel version and loading the congestion control module. We use the default kernel's implementations of CUBIC and BBRv1, and Google's kernel fork for BBRv2 and BBRv3 (Table 2).

In QUIC, we simply select the CCA as a command-line argument to the user space implementation. We evaluate Google `quiche` [16], Cloudflare `quiche` [10], and `picoquic` [22]. Google and Cloudflare `quiche` are their production implementations of the same name, and `picoquic` is a minimalist implementation based on the IETF spec. All three include CUBIC and BBRv1 implementations, as well as some form of BBRv2 or BBRv3, which is still undergoing standardization.

For the transparent, connection-splitting TCP PEP, we use PEPsal [3]. PEPsal intercepts the SYN packet during the three-way handshake and forms separate TCP connections with each endpoint, copying data between the two sockets. Note that discussions of split QUIC are based only on the split throughput heuristic and do not use an actual splitter.

Experiment specification. In each experiment, the HTTPS client requests a specific number of bytes to be transferred from the server in the HTTPS payload. This number corresponds to the amount of data that can be transmitted through the bottleneck link over a 10-second period. We have found this to be sufficiently large to reflect sustained throughput.

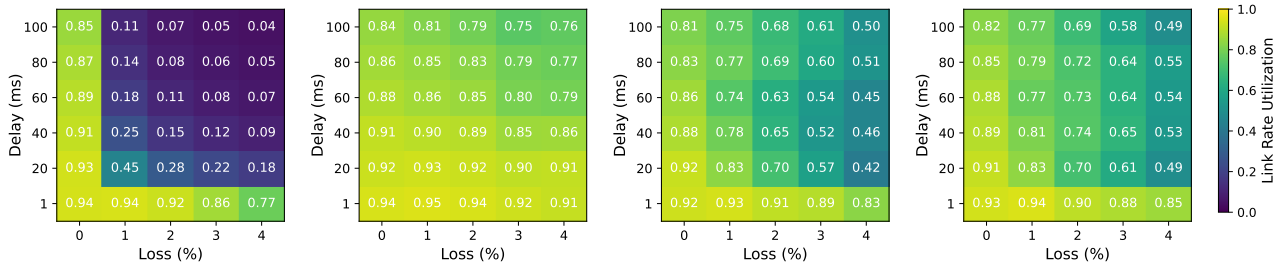
In practice, we report the "single-stream goodput" calculated as the number of application-layer bytes received (excluding HTTPS headers) divided by request completion time (from when the client sends a request to when it receives a complete response). Due to header overhead and request latency, this metric will never equal the link rate.

Machine specification. All experiments use CloudLab [12] x86_64 rs630 nodes in the Massachusetts cluster running Ubuntu 22.04.2. The nodes use the default Linux kernel v5.15.0-122-generic, except for TCP BBRv2 and BBRv3.

5 Results

In our emulation measurement study, we aim to answer three questions to gain a more comprehensive understanding of the relevance of connection-splitting with modern networks:

1. Has the model-based BBR algorithm made the throughput enhancements of connection-splitting obsolete?
2. Are there classes of network settings where BBR benefits significantly from splitting but CUBIC does not?
3. How does CCA implementation impact end-to-end behavior, and therefore split behavior?



(a) TCP CUBIC. **(b)** TCP BBRv1. **(c)** TCP BBRv2. **(d)** TCP BBRv3.

Figure 4: The link rate utilization calculated as the ratio of achieved goodput to link rate for various TCP CCAs in an emulated network path, at various loss rates and one-way delays, at 10 Mbit/s. CUBIC is the most sensitive to loss and delay, while BBRv1 is the most aggressive and achieves the highest utilizations; BBRv2 and BBRv3 are more sensitive to loss than BBRv1 and utilization starts to suffer (left to right). CCAs tend to achieve lower utilizations, though higher absolute goodputs, as the link rate increases (Appendix A). Median of $n = 20$ trials.

To recap our measurement methodology, we have a method for measuring the throughput of TCP connections in emulation both with and without a transparent PEP. We also have the ability to estimate throughput both with and without a generic connection splitter, for both TCP and QUIC, based on knowledge of the network model of each path segment, and the measured end-to-end throughput on each segment.

For our analysis, we cache measurements for the end-to-end network parameters in Table 1 and the congestion control schemes in Table 2. We also run experiments in the two-segment network topology to validate some of these predictions. Raw data for the end-to-end behavior of each CCA implementation are available in Appendix A. Our emulation benchmarks are also publicly available on GitHub: https://github.com/StanfordSNR/connection-splitting.

5.1 Finding: Splitting has become significantly more beneficial to TCP BBR since it was initially released in 2016.

Has the model-based BBR algorithm made the throughput enhancements of connection-splitting obsolete? We find this line of thought has some truth with the initial release of TCP BBRv1 in 2016. But as the BBR algorithm has evolved into BBRv2 in 2019 and now BBRv3 in 2023, it has also evolved to behave more conventionally in the sense that it benefits from being split—just like more traditional, loss-based congestion control schemes such as CUBIC.

Fig. 1 shows three different network settings in which BBRv2 and BBRv3 show significant throughput gains from connection-splitting. When BBRv1 was released, its throughput both with and without the connection-splitter were roughly the same, and nearly achieved the bottleneck link rate. With BBRv2 and BBRv3, the end-to-end throughput drastically deteriorated, behaving more like CUBIC than BBRv1. However, the split throughput remained relatively high, suggesting that BBRv3 today would still benefit from connection-splitting.

Analysis. To explore why CUBIC and BBRv3 benefit from splitting but BBRv1 does not, we analyze end-to-end throughputs of each CCA and apply the split throughput heuristic (Fig. 2). As described in §3, this methodology allows us to study split settings by measuring the much smaller parameter space of end-to-end connections. We find that connection-splitting is likely to improve throughput on lossy paths for CUBIC, BBRv2, and BBRv3 connections, but not for BBRv1.

Fig. 4 visualizes our cached end-to-end measurements as link rate utilization heatmaps of loss vs. delay. Here is an example for how to interpret these graphs using Fig. 4d. The top-left cell represents a 100 ms 0% segment with 0.82 utilization of the 10 Mbit/s link rate, and the bottom-right cell represents a 1 ms 4% segment with 0.85 utilization of the 10 Mbit/s link rate. The predicted split utilization of a network path composed of these two segments is just 0.82, the minimum. The two cells compose to the top-right cell, which represents a 100 ms 4% 10 Mbit/s segment with an end-to-end utilization of 0.49. Since $0.82 > 0.49$, we say that splitting has improved the throughput of this network path.

BBRv1 achieves high link rate utilization in all settings (Fig. 4b), showing it has little to gain from splitting. In fact, previous studies have shown that BBRv1 achieves ≈85% link utilization regardless of loss (same as our findings) before it reaches a cliff point at around 20% loss [4, 8]. This may be why it appears that splitting in lossy settings is now obsolete with BBR. However, one should note that BBRv1's high throughput has long been attributed to its aggressiveness and unfairness to legacy algorithms [4, 50], which is what led to the changes in BBRv2 and BBRv3.

While BBRv2 is a large departure from BBRv1, BBRv3 has been described as BBRv2 with bugfixes and performance tuning [6], which supports why the two are so similar. We focus on BBRv3 since Google hopes to now deprecate BBRv2 [6].

BBRv3 is more sensitive to loss than its previous versions (Fig. 4d), and more similar to CUBIC (Fig. 4a) in that there exist scenarios where end-to-end throughput suffers. These reflect existing findings of lower utilization in BBRv2 and BBRv3 [11, 47, 54]. Based on the heuristic, it is clear that in some lossy networks such as the ones empirically evaluated

in Fig. 1, connection-splitting can significantly increase the throughput of BBRv3 connections. It is possible that the BBR algorithm continues to evolve in this direction given that BBR's unfairness remains contentious today [11, 54].

Summary. While BBR may not have benefited from splitting with the release of "v1" in 2016, BBRv2 and now BBRv3 have evolved to behave more conventionally—similar to traditional, loss-based CCAs such as CUBIC—in the sense that they *do*. Even so-called "model-based" congestion control algorithms seem to now react to loss as a congestion signal, as the BBR algorithm continues to evolve.

5.2 Finding: There exist classes of network paths where TCP BBRv3 would significantly benefit from splitting but TCP CUBIC would not.

Are there new classes of network settings where BBR benefits significantly from splitting but CUBIC does not? We want to understand the network settings in which a congestion control scheme is not able to achieve practical bottleneck link rate utilizations end-to-end, but is with a connection-splitter.

We find that while splitting benefits BBRv3 in all the same scenarios as CUBIC, it also has the potential to benefit BBRv3 in many *new* scenarios. In addition to edge deployments with a lossy last-mile, BBRv3 also benefits from splitting in scenarios where there is loss on both path segments, and when the connection-splitter is located farther from the edge. We partition these scenarios into three classes of network settings:

I. Paths with asymmetric delay and a lossy last-mile,
II. Lossy paths with asymmetric delay,
III. Lossy paths with more symmetric delay.

Figs. 1a to 1c represent network settings in each class, respectively. "Asymmetric" refers to the delays on the two path segments. These emulations empirically demonstrate that CUBIC only benefits in the first class, while BBRv3 benefits in all three. BBRv1 does not need splitting in any context.

Analysis. To identify which network paths benefit from connection-splitting and where along the paths PEPs should be deployed, we apply the heuristic (Fig. 2). For each CCA, we conduct an exhaustive search of the $15 \cdot 21 \cdot 25 = 7875$ combinations of settings within our parameter space (Table 1), and efficiently predict the split and end-to-end throughputs.

We filter on the predicted throughputs for network settings where splitting improves end-to-end throughput by at least $3\times$, and where the split connection utilizes at least half the bottleneck link rate (Table 3). BBRv1 does not meet these criteria is any scenarios, and the theoretical connection-splitter is unable to improve the throughput of BBRv1 by even 50%. CUBIC and BBRv3 meet these criteria in 942 and 188 scenarios, respectively. CUBIC benefits from splitting in more scenarios because its end-to-end utilization is more frequently low, so it more frequently has a large split improvement.

Filter	BBRv1	CUBIC	BBRv3
Initial	7875	7875	7875
Split imprvmnt. $> 3\times$	0	2231	234
Split utilization > 0.5	0	942	188
Asymmetric, last-mile	0	942	38
Asymmetric, lossy	0	0	72
Symmetric, lossy	0	0	78

Table 3: An exhaustive search of network paths and their PEP locations that benefit from splitting for each CCA, and the number of filtered settings that belong to each class.

Since the distribution of network paths in our parameter space does not reflect any meaningful real world distribution, we are more interested in the *classes* of network settings that benefit from splitting. We realized that *all* of the relevant network settings for CUBIC can be clustered into Class I, as network paths where one path segment has 1 ms delay and non-zero loss, and the other has > 1 ms delay and 0% loss. However, Class I only accounts for 21% of the relevant network settings for BBRv3. We identify Class II, which is the same as I, except both path segments have non-zero loss. Class III is the same as II, except both path segments have > 1 ms delay. We used the results to select three representative network settings to empirically evaluate in Fig. 1.

Intuitively, we can understand why BBRv3 benefits more from splitting in lossy scenarios than CUBIC based on how it reacts to loss and delay (Fig. 4d). BBRv3's sensitivity to loss and delay is more gradual than abrupt, so it is more likely to benefit when splitting a lossy, high-delay network path in any way. In comparison, CUBIC's throughput falls off a cliff for many of these segmentations.

Discussion. Do these results reflect where PEP deployments have been useful in the real world? Connection-splitters have traditionally been found in satellite, cellular, and Wi-Fi networks with a wireless link or rate policer [13, 21]. This resembles Class I, where the network path consists of a lossy last-mile, and a reliable Internet path segment. It makes sense then for PEPs to be traditionally located at the edge to address the issues of loss-based schemes [14, 17, 39]. We expect these PEPs to similarly benefit BBRv3 in the same locations.

In Classes I and II, asymmetric delay can be severe in low-resource networks in addition to wireless last-mile links; Consider, for example, regions with no IXPs in which a significant proportion of Internet traffic travels internationally.

For Classes II and III, satellite (and also wireless ad-hoc) networks are known for having lossy "middle-miles" [2, 30, 39, 41]. This can be due to bad weather, fast-moving satellites, and long-distance radio waves, etc. Since CUBIC does not trivially benefit from splitting in these scenarios, the traditional solution has been to split the connection at multiple points and to use an FEC-based or other proprietary protocol in the satellite backhaul [2, 17, 39]. Our results suggest such

an invasive solution may not be necessary for BBRv3.

How could this inform the deployment of connection-splitting PEPs? With the caveat that futher exploration is required to understand how the heuristic extrapolates to the real world, one idea is to determine where to deploy PEPs along an existing network path for maximum benefit especially as congestion-control schemes evolve. Another idea is given the location of a PEP, determine which connections going through the PEP would most benefit based on knowledge of each connection's network path. We leave network operators to decide how best to model their networks and apply the heuristic to evaluate potential PEP deployments.

Summary. While TCP CUBIC only benefits from connection-splitting when the PEP is located at the lossy last-mile, TCP BBRv3 can also benefit when there is loss on both sides of the PEP and when there are longer propagation delays. This suggests that TCP connections using BBRv3 should benefit from splitters at the same locations as for CUBIC, and also that traditional methods used to address loss in the middle-mile could use simple connection-splitting instead. We believe this method of analyzing useful network paths and PEP placements for connection-splitting can be extended to model new types of networks, especially in space.

5.3 Finding: QUIC implementations of the same congestion control schemes vary significantly, and further differ from Linux's TCP implementations.

How does CCA implementation impact end-to-end behavior, and therefore split behavior? Might claims about "split throughput" depend not just on the CCA, but also the *implementation* and/or the transport protocol on top of it?

For our initial study, we compared end-to-end throughput for four open-source implementations each of CUBIC, BBRv1, and BBRv2/3; one using TCP and three using QUIC. We find that the end-to-end behavior of each CCA varies by implementation in both baseline throughput and sensitivity to loss and delay. To evaluate the split behavior of QUIC, instead of creating a custom and explicit connection-splitting PEP for each QUIC implementation, we apply the split throughput heuristic and argue that these implementations will likewise respond differently to connection-splitting PEPs.

Fig. 5 visualizes the end-to-end behavior of these schemes. We highlight that the Cloudflare and `picoquic` CUBIC implementations are less sensitive to loss than Google or TCP; the former may benefit from splitting in more classes of network settings than the latter. Additionally, their BBR implementations exhibit non-uniform behavior, suggesting a non-uniform response to connection-splitting. These variations indicate that the benefits of in-network assistance should be considered along with not just the CCA but its specific implementations.

Analysis. The Google QUIC (Figs. 5b, 5f and 5j) and TCP (Figs. 5a, 5e and 5i) implementations are most similar to each other for each CCA. This is reasonable if we take the community-based CUBIC implementation in Linux to be the standard, and considering that Google contributed to the Linux BBR implementations. In general, Google QUIC achieves slightly higher utilization than Linux TCP.

The Cloudflare QUIC BBR implementations (Figs. 5c and 5g) exhibit profoundly different behavior from Linux TCP in baseline performance. Note that Cloudflare uses BBRv1 for TCP but their use of BBR in QUIC is experimental. Anecdotally, a Cloudflare employee has expressed difficulty making their BBR implementation performant, having to reverse engineer the Linux kernel [15]. Given the wide adoption of BBRv1 for TCP at many CDNs [51], we expect it to be desirable yet challenging for these same companies to correctly incorporate BBRv3 into their QUIC stacks in the coming years.

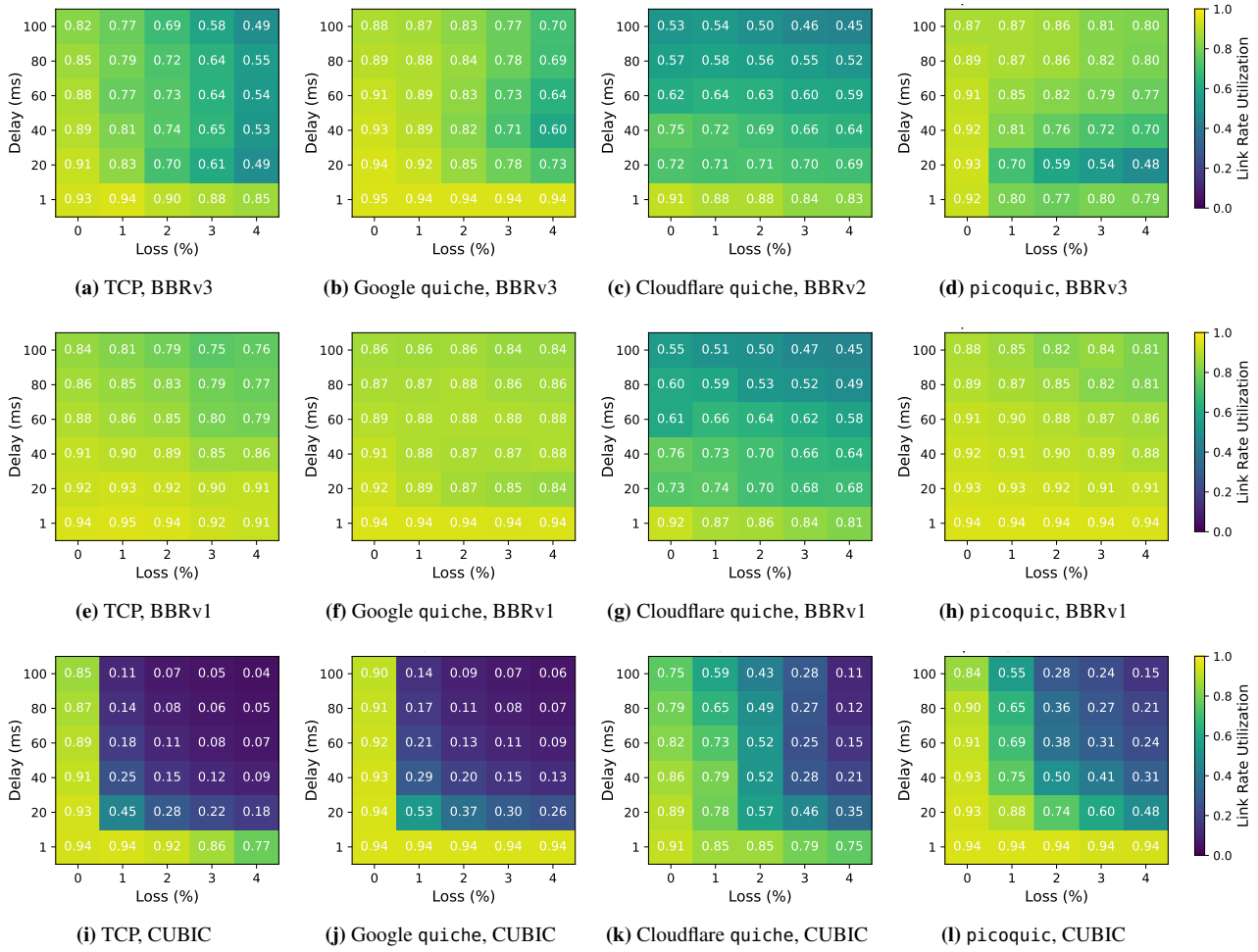
The `picoquic` BBR implementations (Figs. 5d and 5h) are more similar to Linux TCP, although its BBRv3 implementation seems to have a contradicting reaction to delay. We note that `picoquic` is intended for experimental use in the IETF [22]. It is important then to understand its congestion control behavior if it is to be used to evaluate IETF proposals. We believe its differences from Linux TCP warrant further exploration, but perhaps also that the ongoing standardization efforts of BBR in the IETF [9] indicate that there is no monolith yet of "the BBR algorithm."

The Cloudflare QUIC and `picoquic` implementations of CUBIC (Figs. 5k and 5l) interestingly both exhibit a more gradual degradation in response to loss and delay than Linux TCP (Fig. 5i). We find this harder to explain, given that TCP CUBIC has been around since 2006, and perhaps can be attributed to transport protocol mechanisms in QUIC. Nevertheless, this indicates that it is important to understand the behavior of a CCA in the context of its entire implementation.

Fig. 6 applies the heuristic to show how split behavior can vary for CCA implementations with different end-to-end behaviors. Some QUIC CUBIC implementations benefit in new network settings where TCP CUBIC does not, while the various QUIC BBRv3 implementations exhibit non-uniform end-to-end behavior and thus no clear trend in split behavior.

Discussion. Why do we believe we can extrapolate split behavior from the end-to-end behavior of QUIC? Previous studies explore the effects of QUIC's transport protocol mechanisms in such a case [26, 48]. They find that the effects of zero-RTT connection establishment with regards to long-lived throughput to be minimal, and stream multiplexing to be mutually beneficial in both scenarios. Further studies can clarify the interactions between CCAs and transport mechanisms.

How do we know that the difference in behavior is due to the congestion control implementation and not other transport protocol mechanisms? Well, we don't, and there is known to be significant variance in the features supported by different QUIC implementations [33]. However, we find it intuitive that congestion control would be a major factor in the measured sustained goodput of a bulk file transfer.



(a) TCP, BBRv3 **(b)** Google `quiche`, BBRv3 **(c)** Cloudflare `quiche`, BBRv2 **(d)** `picoquic`, BBRv3

(e) TCP, BBRv1 **(f)** Google `quiche`, BBRv1 **(g)** Cloudflare `quiche`, BBRv1 **(h)** `picoquic`, BBRv1

(i) TCP, CUBIC **(j)** Google `quiche`, CUBIC **(k)** Cloudflare `quiche`, CUBIC **(l)** `picoquic`, CUBIC

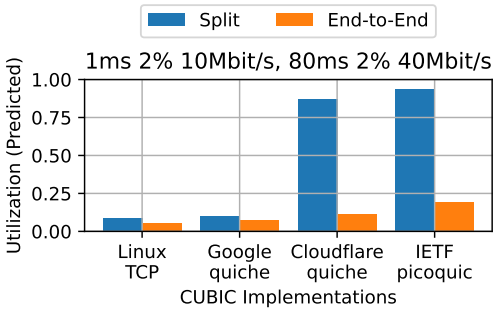
Figure 5: Heatmaps for three QUIC implementations of BBRv3 (or BBRv2), BBRv1, and CUBIC showing link rate utilization calculated as the ratio of achieved goodput to link rate, compared to Linux TCP. The heatmaps are shown at various loss rates and one-way delays with a fixed link rate of 10 Mbit/s. User-space QUIC is not CPU-limited, achieving high utilizations at 1 ms delay and 0% loss. The QUIC implementations are Google `quiche`, Cloudflare `quiche`, and a minimalist implementation based on the IETF spec called `picoquic`. Median of $n = 20$ trials.

The divergence in QUIC implementations is not so dissimilar from that of TCP at a similar state of evolution [1], but there are still some fundamental differences. With QUIC implementations in userspace, there is a larger diversity of implementations that are easier to tune for a specific application metric, as opposed to correctness or fairness from a congestion-control point of view. These algorithms are also highly parameterized, with no standard nor test suite, so it's not surprising that the implementations differ.

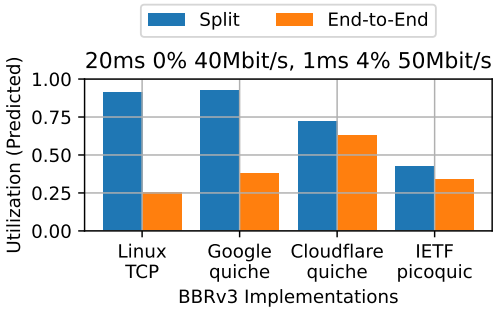
If there is reason to believe that QUIC is losing out on throughput by not connection-splitting, then there will continue to be research on how to achieve the same benefits without ossification [28, 29, 52, 53]. The heuristic helps us understand the theoretical achievable throughput with a simple connection-splitter, or even by combining multiple end-to-end congestion control schemes. In addition to research, this could motivate privacy-minded proposals in the Internet standards community to also view themselves as potential deployment opportunities for private PEPs [27, 43–45].

Another application of analyzing CCA implementations is to analyze the Linux TCP implementations of BBR *within* a major version over time. We did this analysis with BBRv1 on 13 Linux kernels between 2016 and 2024, but found no significant variance. However, given the dynamic nature of BBR and the Linux networking stack, it is possible there will still be changes to the split behavior of TCP in the future.

Summary. We believe it is important to understand the end-to-end behavior of a congestion control scheme in the context of its entire implementation. Our results suggest that BBR is challenging to implement, and that even CUBIC implementations can vary based on context. Whether the prevailing wisdom is that a specific CCA or transport protocol has made in-network assistance undesirable, these results suggest that it is valuable to consistently re-evaluate these claims.



(a) Some QUIC CUBIC implementations can benefit in new network classes where TCP CUBIC could not.



(b) The various BBRv3 implementations have non-uniform end-to-end behavior and no clear resulting split behavior.

Figure 6: Predicted bottleneck link rate utilizations calculated from the predicted end-to-end and split throughputs of the TCP and QUIC implementations, on two different network path segments. End-to-end behavior of each CCA varies significantly by implementation.

6 Accuracy Analysis

Our analysis of connection-splitting for TCP and its extensions to QUIC rely on the accuracy of the split throughput heuristic. In this section, we are primarily concerned with how accurate our predictions, which are based on measurements from a one-segment topology, are for measurements from a two-segment topology in emulation, without and with a connection-splitting TCP PEP:

- Does the `compose` function accurately represent the combined network path in the end-to-end throughput?
- Does the split throughput heuristic accurately predict split throughput?

Most importantly, we find the heuristic to be able to usefully predict *trends*. In terms of absolute predictions, we find the `pred_e2e_throughput()` and `pred_split_throughput()` functions to be correct within a reasonable tolerance, with a slight tendency to overestimate.

Orthogonally, we do not evaluate the accuracy to which emulation studies reflect the real world with multi-flow settings and more complex network properties, nor how the accuracy would extrapolate to QUIC connections with custom connection splitters. It may be interesting to explore how to incorporate such factors into the network model and heuristic.

Methodology. Recall that for a given network path composed of two path segments, we can obtain both the predicted end-to-end and split throughputs, and the ground truth throughputs in an emulated network with and without a TCP PEP. Then for a network setting, we can compute the accuracy as the percent error in predicted vs. measured throughput.

We perform an empirical accuracy analysis of BBRv3 for two end-to-end network paths with identical bandwidth and delay, both without (0%) and with (4%) loss. We test various splits for the bandwidth, delay, and loss and analyze the accuracy trends. In particular, we select delay splits such that the end-to-end delay is 80 ms, bandwidth splits such that the bottleneck bandwidth is 10 Mbit/s, and loss splits such that the total loss is either 0% or 4%. We parameterize the network path segments to use the cached measurements from Table 1.

6.1 End-to-End Throughput Accuracy

Each of our splits composes to the same end-to-end network path, so we predict the same end-to-end throughput for each. Our experimental results in Fig. 7a show that the measured end-to-end throughputs are also roughly uniform, especially without loss, indicating that our method of composing network path segments in emulation represents the same end-to-end network path.

6.2 Split Throughput Accuracy

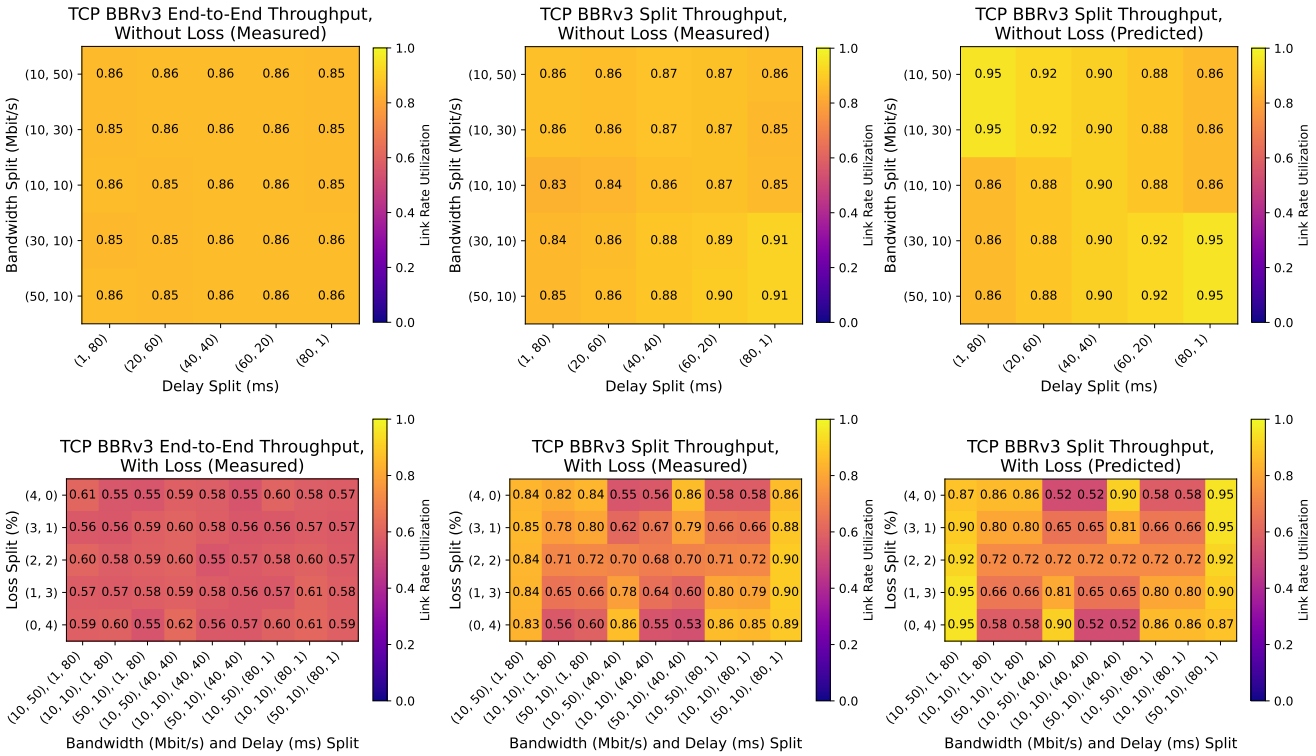
The split throughput predictions accurately reflect trends in loss, delay, and bandwidth (Fig. 7b). For example, split throughput is generally higher when the high-bandwidth link is paired with high delay (the yellow-est cells). It is also lower when the lossy link is paired with high delay (columns 1 and 2) or low bandwidth (columns 3-5).

The split throughput predictions tend to slightly overestimate, but we think the level of error is small enough to be helpful for informing real PEP deployment. The maximum error is $\pm 14\%$, and on average $\pm 4\%$. This dwarfs the measured gains in some situations, and rules out a large gain in others.

One factor that may lead to overestimation of split throughput is the queue behavior and the burstiness of the sender. With small queues and bursty sending, we would expect the far path segment from the data sender to sometimes be limited by the send buffer. This could subsequently affect how the far connection probes for and utilizes available link rate capacity.

Another factor is the proximity of the bottleneck link to the sender. While our heuristic does not account for this, the real split throughputs for symmetric pairs of network paths (e.g., the left three and right three columns in Fig. 7b), show that we slightly overestimate when the low-bandwidth bottleneck link is far from the sender. Based on our reasoning about queues, this would suggest that the far path segment, already the bottleneck, is even further under-saturated.

Overall, we find our end-to-end and split throughput predictions to usefully reflect relative trends and absolute throughput within a reasonable tolerance. They are not intended to



(a) End-to-end throughput accuracy.

(b) Split throughput accuracy.

Figure 7: Heatmaps of the measured and predicted BBRv3 throughputs for various splits of delay, bandwidth, and loss, both without (top) and with (bottom) loss. The *end-to-end throughput* predictions (not pictured) are the same for all cells at 0.86 utilization without loss and 0.58 utilization with loss, because they represent the same network path, so end-to-end prediction errors are roughly uniform. The *split throughput* predictions err slightly on the side of overestimation, but they accurately reflect trends in higher or lower throughputs for measurements on different splits of the same network path. Median of $n = 40$ trials.

make claims about exact achievable throughputs nor about the immediate utility of PEPs in the real world, but simply to reason about how connection-splitting may impact long-lived throughput in a simplified network model.

7 Conclusion

We performed an emulation measurement study on the impact of connection-splitting PEPs on sustained throughput in the context of recent developments such as BBR and QUIC. We found that TCP BBR benefits more from splitting today than when it was first released, and its current version benefits in settings where TCP CUBIC does not. QUIC congestion-control implementations exhibit substantial variability both within QUIC and with Linux TCP.

In the short term, we urge researchers to refer to congestion-control schemes by algorithm/implementation/version, not just "BBR" or even "QUIC BBRv1". We urge the community to create regression and acceptance tests—possibly including our heatmaps with "permissible zones"—for a scheme to call itself an implementation of the XYZv1 algorithm.

For connection-splitting to truly be relevant again, there must be clear scenarios where PEPs offer significant benefits but end-to-end solutions fall short. This paper is a first step, but real-world studies are needed. Next, even though the problem is old, the solution need not be to "just do the same things." We hope this paper motivates the community to pursue protocol-agnostic approaches to in-network assistance and destigmatize the controversial nature of PEPs.

Acknowledgments

We thank Michael Welzl, our shepherd Jon Crowcroft, and the anonymous USENIX ATC '25 reviewers for their invaluable feedback, Frode Kileng for a comment that partly inspired this work [25], and members of IRTF PANRG and Stanford SNR for other helpful discussions. This work was supported in part by NSF grants 2045714 and 2039070, DARPA contract HR001120C0107, a Stanford School of Engineering Fellowship, a Sloan Research Fellowship, affiliate members and other supporters of Stanford DAWN, including Meta, Google, and VMware, as well as Cisco and SAP, and Huawei, Dropbox, Amazon, and the Mozilla Foundation. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the sponsors.

References

[1] Mark Allman and Aaron Falk. On the effective evaluation of TCP. *ACM SIGCOMM Computer Communication Review*, 29(5):59–70, 1999.

[2] John Border, Bhavit Shah, Chi-Jiun Su, and Rob Torres. Evaluating QUIC's Performance Against Performance Enhancing Proxy over Satellite Link. In *2020 IFIP Networking Conference (IFIP Networking)*, pages 755–760, 2020.

[3] Carlo Caini, Rosario Firrincieli, and Daniele Lacamera. PEPsal: a Performance Enhancing Proxy designed for TCP satellite connections. In *2006 IEEE 63rd Vehicular Technology Conference*, volume 6, pages 2607–2611, 2006.

[4] Yi Cao, Arpit Jain, Kriti Sharma, Aruna Balasubramanian, and Anshul Gandhi. When to Use and When Not to Use BBR: An Empirical Analysis and Evaluation Study. In *Proceedings of the Internet Measurement Conference (IMC)*, pages 130–136, 2019.

[5] Neal Cardwell. BBR Congestion Control Work at Google, March 2018. Presentation, IETF 101 London, Internet Congestion Control Research Group (ICCRG). https://datatracker.ietf.org/meeting/101/materials/slides-101-iccrg-an-update-on-bbr-work-at-google-00.

[6] Neal Cardwell. BBRv3: Algorithm Overview and Google's Public Internet Deployment. Presentation, IETF 119 Brisbane, Congestion Control Working Group (CCWG). https://datatracker.ietf.org/meeting/119/materials/slides-119-ccwg-bbrv3-overview-and-google-deployment-00, March 2024.

[7] Neal Cardwell and Yuchung Cheng. BBR Congestion Control. Presentation, IETF 97 Seoul, Internet Congestion Control Research Group (ICCRG). https://www.ietf.org/proceedings/97/slides/slides-97-iccrg-bbr-congestion-control-01.pdf, November 2016.

[8] Neal Cardwell, Yuchung Cheng, C. Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. BBR: Congestion-Based Congestion Control. *Communications of the ACM*, 60(2):58–66, 2017.

[9] Neal Cardwell, Ian Swett, and Joseph Beshay. BBR Congestion Control. IETF Draft. https://datatracker.ietf.org/doc/draft-cardwell-ccwg-bbr/, September 2024.

[10] Cloudflare, Inc. quiche. GitHub repository. https://github.com/cloudflare/quiche, January 2025.

[11] Soumyadeep Datta and Fraida Fund. Replication: "When to Use and When Not to Use BBR". In *Proceedings of the 2023 ACM on Internet Measurement Conference (IMC)*, pages 30–35, 2023.

[12] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. The design and operation of CloudLab. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 1–14, Renton, WA, July 2019. USENIX Association.

[13] Korian Edeline and Benoit Donnet. A Bottom-Up Investigation of the Transport-Layer Ossification. In *2019 Network Traffic Measurement and Analysis Conference (TMA)*, pages 169–176, 2019.

[14] Viktor Farkas, Balázs Héder, and Szabolcs Nováczki. A Split Connection TCP Proxy in LTE Networks. In Róbert Szabó and Attila Vidács, editors, *Information and Communication Technologies*, pages 263–274, Berlin, Heidelberg, 2012. Springer.

[15] IETF Internet Engineering Task Force. IETF 119: Congestion Control Working Group 2024-03-21 03:00. https://www.youtube.com/watch?v=ZVqQiA7h-W8, March 2024. Q&A: 1:37:30.

[16] Google. QUICHE. GitHub repository. https://github.com/google/quiche/, January 2025.

[17] Jim Griner, John Border, Markku Kojo, Zach D. Shelby, and Gabriel Montenegro. Performance Enhancing Proxies Intended to Mitigate Link-Related Degradations. RFC 3135, June 2001.

[18] Sangtae Ha, Injong Rhee, and Lisong Xu. CUBIC: A New TCP-Friendly High-Speed TCP Variant. *SIGOPS Operating Systems Review*, 42(5):64–74, July 2008.

[19] Mark J. Handley, Jitendra Padhye, Sally Floyd, and Joerg Widmer. TCP Friendly Rate Control (TRFC): Protocol Specification. RFC 5348, September 2008.

[20] David A. Hayes, David Ros, and Özgü Alay. On the Importance of TCP Splitting Proxies for Future 5G mmWave Communications. In *2019 IEEE 44th LCN Symposium on Emerging Topics in Networking (LCN Symposium)*, pages 108–116, 2019.

[21] Michio Honda, Yoshifumi Nishida, Costin Raiciu, Adam Greenhalgh, Mark Handley, and Hideyuki Tokuda. Is it Still Possible to Extend TCP? In *Proceedings of the Internet Measurement Conference (IMC)*, pages 181–194, 2011.

[22] Christian Huitema. picoquic. GitHub repository. https://github.com/private-octopus/picoquic, January 2025.

[23] Jana Iyengar and Martin Thomson. QUIC: A UDP-Based Multiplexed and Secure Transport. RFC 9000, May 2021.

[24] Van Jacobson and Michael J. Karels. Congestion Avoidance and Control. In *SIGCOMM 1988*, Stanford, CA, August 1988.

[25] Frode Kileng. Comments at IETF 121 PANRG session. "there's been a sharp decline of PEPs in mobile networks in the last years. BBR is a cause for this. (When AWS switched to BBR 2-3 years ago, throughput of moblle networks with PEPs either declined or a steady trend, while those without had a significant increase in throughput. I.e. as measured bye 3rd party benchmarks, e.g. Tutela)" https://zulip.ietf.org/#narrow/stream/287-panrg/topic/ietf-121/near/144090.

[26] Mike Kosek, Hendrik Cech, Vaibhav Bajpai, and Jörg Ott. Exploring Proxying QUIC and HTTP/3 for Satellite Communication. In *2022 IFIP Networking Conference (IFIP Networking)*, pages 1–9, 2022.

[27] Mike Kosek, Tanya Shreedhar, and Vaibhav Bajpai. Beyond QUIC v1: A First Look at Recent Transport Layer IETF Standardization Efforts. *IEEE Communications Magazine*, 59(4):24–29, 2021.

[28] Mike Kosek, Benedikt Spies, and Jörg Ott. Secure Middlebox-Assisted QUIC. In *2023 IFIP Networking Conference (IFIP Networking)*, pages 1–9. IEEE, 2023.

[29] Zsolt Krämer, Mirja Kühlewind, Marcus Ihlar, and Attila Mihály. Cooperative performance enhancement using QUIC tunneling in 5G cellular networks. In *Proceedings of the Applied Networking Research Workshop*, ANRW '21, page 49–51, New York, NY, USA, 2021. Association for Computing Machinery.

[30] Nicolas Kuhn, François Michel, Ludovic Thomas, Emmanuel Dubois, and Emmanuel Lochin. QUIC: Opportunities and threats in SATCOM. In *2020 10th Advanced Satellite Multimedia Systems Conference and the 16th Signal Processing for Space Communications Workshop (ASMS/SPSC)*, pages 1–7, 2020.

[31] Adam Langley, Alistair Riddoch, Alyssa Wilk, Antonio Vicente, Charles Krasic, Dan Zhang, Fan Yang, Fedor Kouranov, Ian Swett, Janardhan Iyengar, et al. The QUIC Transport Protocol: Design and Internet-Scale Deployment. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*, pages 183–196, 2017.

[32] Aitor Martin and Naeem Khademi. On the Suitability of BBR Congestion Control for QUIC Over GEO SATCOM Networks. In *Proceedings of the Workshop on Applied Networking Research (ANRW)*, pages 1–8, 2022.

[33] Robin Marx, Joris Herbots, Wim Lamotte, and Peter Quax. Same Standards, Different Decisions: A Study of QUIC and HTTP/3 Implementation Diversity. In *Proceedings of the Workshop on the Evolution, Performance, and Interoperability of QUIC*, pages 14–20, 2020.

[34] Matt Mathis and Jamshid Mahdavi. Deprecating the TCP Macroscopic Model. *ACM SIGCOMM Computer Communication Review*, 49(5):63–68, 2019.

[35] Matthew Mathis. Reflections on the TCP Macroscopic Model. *ACM SIGCOMM Computer Communication Review*, 39(1):47–49, 2008.

[36] Matthew Mathis, Jeffrey Semke, Jamshid Mahdavi, and Teunis Ott. The Macroscopic Behavior of the TCP Congestion Avoidance Algorithm. *ACM SIGCOMM Computer Communication Review*, 27(3):67–82, 1997.

[37] Ayush Mishra, Xiangpeng Sun, Atishya Jain, Sameer Pande, Raj Joshi, and Ben Leong. The Great Internet TCP Congestion Control Census. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 3(3):1–24, 2019.

[38] Giorgos Papastergiou, Gorry Fairhurst, David Ros, Anna Brunstrom, Karl-Johan Grinnemo, Per Hurtig, Naeem Khademi, Michael Tüxen, Michael Welzl, Dragana Damjanovic, and Simone Mangiante. De-Ossifying the Internet Transport Layer: A Survey and Future Perspectives. *IEEE Communications Surveys & Tutorials*, 19(1):619–639, 2017.

[39] Abhinav Pathak, Angela Wang, Cheng Huang, Albert Greenberg, Y. Charlie Hu, Randy Kern, Jin Li, and Keith Ross. Measuring and Evaluating TCP Splitting for Cloud Services. In *Proceedings of the 11th International Conference on Passive and Active Measurement (PAM)*, pages 41–50, 2010.

[40] Adithya Abraham Philip, Ranysha Ware, Rukshani Athapathu, Justine Sherry, and Vyas Sekar. Revisiting TCP Congestion Control Throughput Models & Fairness Properties at Scale. In *Proceedings of the 21st ACM Internet Measurement Conference (IMC)*, pages 96–103, 2021.

[41] Alain Pirovano and Fabien Garcia. A New Survey on Improving TCP Performances Over Geostationary Satellite Link. *Network and Communication Technologies*, 2(1):pp–xxx, 2013.

[42] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, 2(4):277–288, November 1984.

[43] Patrick Sattler, Juliane Aulbach, Johannes Zirngibl, and Georg Carle. Towards a tectonic traffic shift? investigating apple's new relay network. In *Proceedings of the 22nd ACM Internet Measurement Conference (IMC)*, pages 449–457, 2022.

[44] David Schinazi. Proxying UDP in HTTP. RFC 9298, February 2021.

[45] David Schinazi and Lucas Pardue. HTTP Datagrams and the Capsule Protocol. RFC 9297, August 2022.

[46] Marten Seemann. QUIC Interop Runner. https://interop.seemann.io/, January 2025.

[47] Yeong-Jun Song, Geon-Hwan Kim, Imtiaz Mahmud, Won-Kyeong Seo, and You-Ze Cho. Understanding of BBRv2: Evaluation and Comparison with BBRv1 Congestion Control Algorithm. *IEEE Access*, 9:37131–37145, 2021.

[48] Ludovic Thomas, Emmanuel Dubois, Nicolas Kuhn, and Emmanuel Lochin. Google QUIC Performance Over a Public SATCOM Access. *International Journal of Satellite Communications and Networking*, 37(6):601–611, 2019.

[49] W3Techs. Usage statistics of QUIC for websites. Web Technology Surveys. https://w3techs.com/technologies/details/ce-quic, April 2025.

[50] Ranysha Ware, Matthew K. Mukerjee, Srinivasan Seshan, and Justine Sherry. Modeling BBR's Interactions with Loss-Based Congestion Control. In *Proceedings of the Internet Measurement Conference (IMC)*, pages 137–143, 2019.

[51] Ranysha Ware, Adithya Abraham Philip, Nicholas Hungria, Yash Kothari, Justine Sherry, and Srinivasan Seshan. CCAnalyzer: An Efficient and Nearly-Passive Congestion Control Classifier. In *Proceedings of the ACM SIGCOMM 2024 Conference*, pages 181–196, 2024.

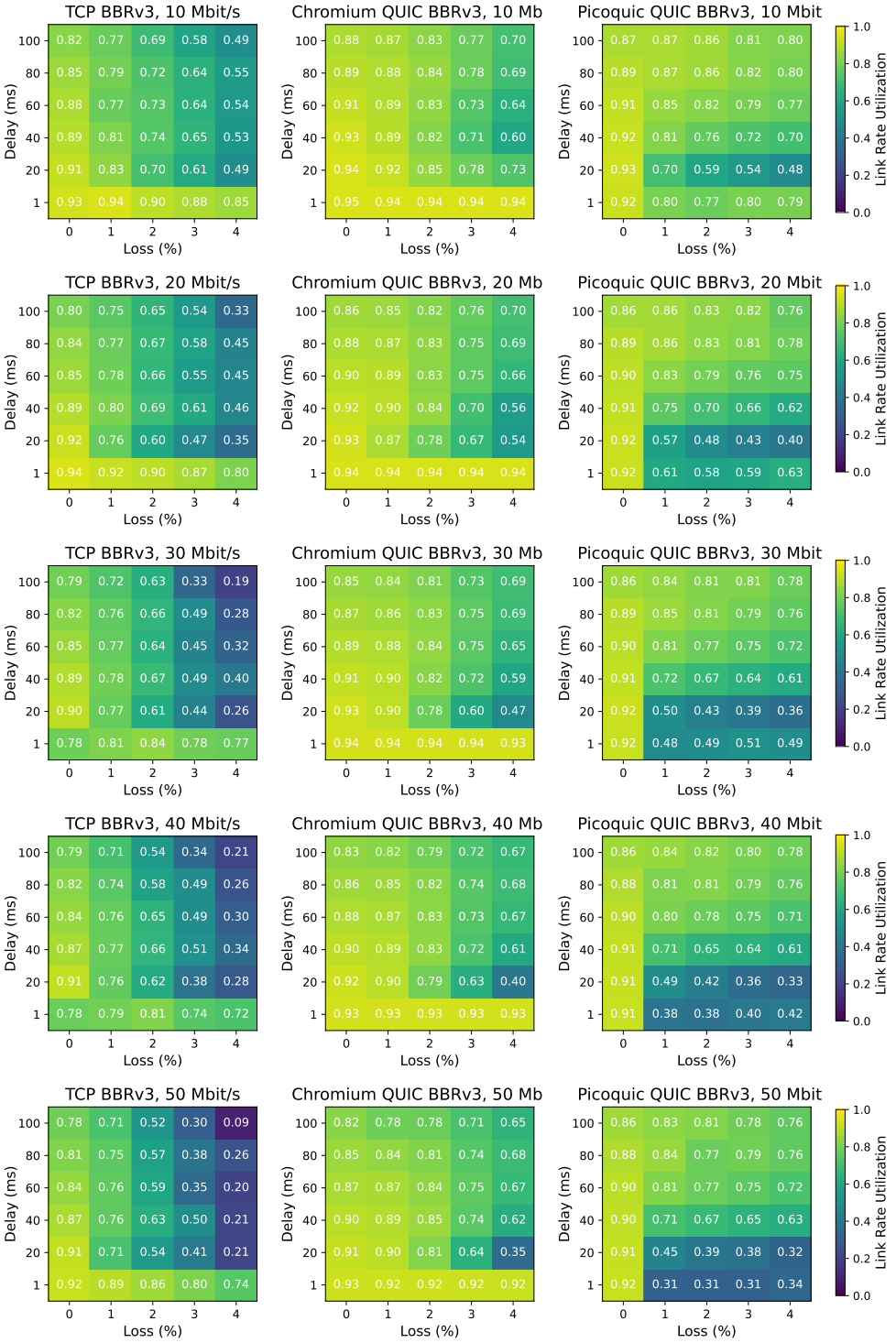
[52] Gina Yuan, Matthew Sotoudeh, David K. Zhang, Michael Welzl, David Mazières, and Keith Winstein. Sidekick:In-Network Assistance for Secure End-to-End Transport Protocols. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 1813–1830, 2024.

[53] Gina Yuan, David K. Zhang, Matthew Sotoudeh, Michael Welzl, and Keith Winstein. Sidecar: In-Network Performance Enhancements in the Age of Paranoid Transport Protocols. In *Proceedings of the 21st ACM Workshop on Hot Topics in Networks*, pages 221–227, 2022.

[54] Danesh Zeynali, Emilia N. Weyulu, Seifeddine Fathalli, Balakrishnan Chandrasekaran, and Anja Feldmann. Promises and Potential of BBRv3. In *International Conference on Passive and Active Network Measurement (PAM)*, pages 249–272. Springer, 2024.

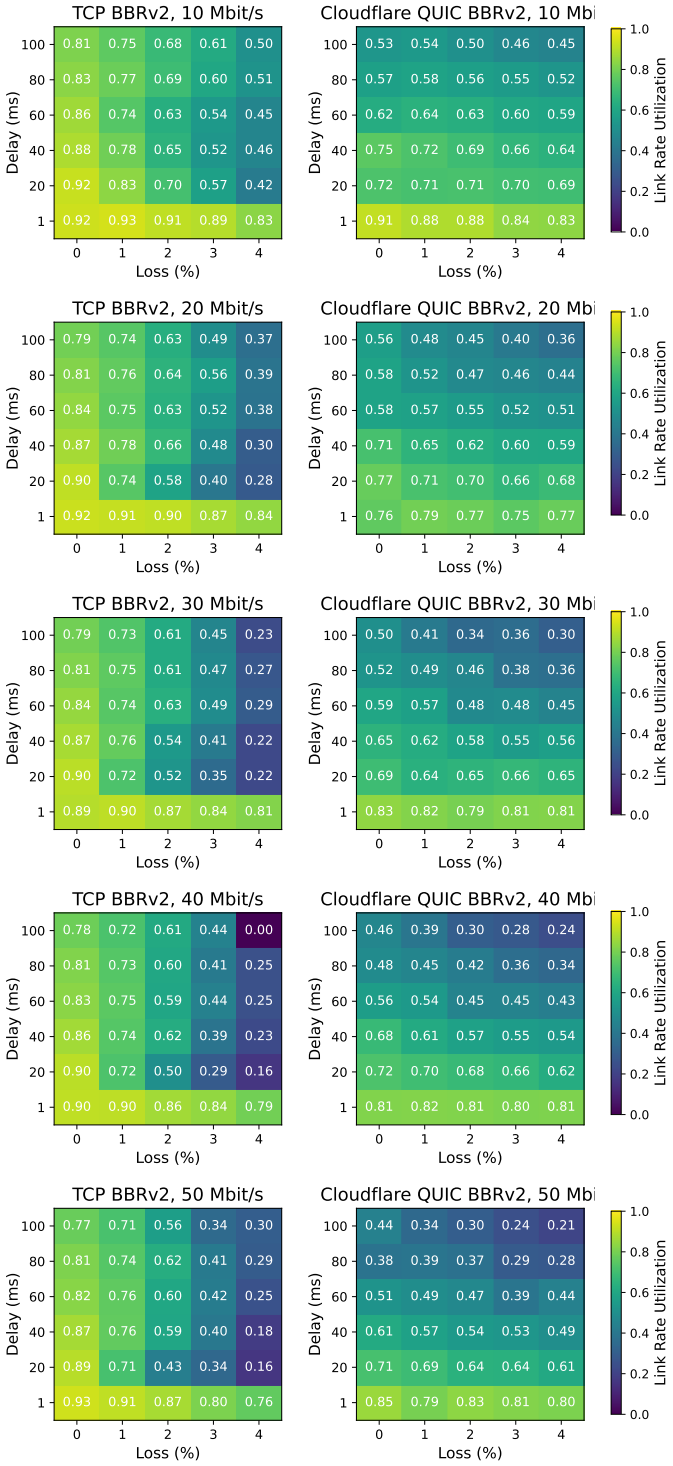
A Appendix

In the appendix, we present the raw data for our characterizations of each congestion control scheme.



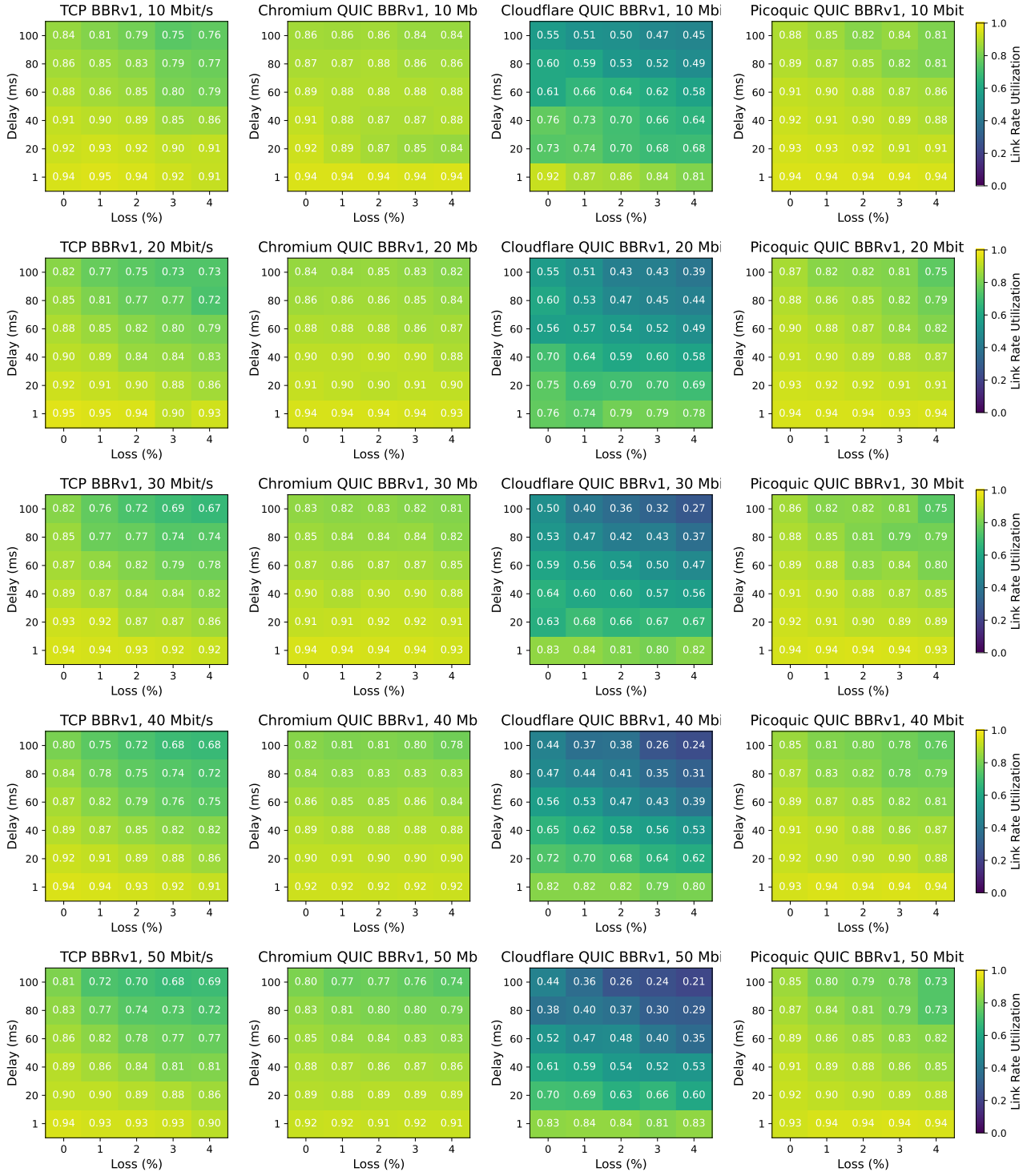
(a) Linux TCP. **(b)** Google `quiche`. **(c)** `picoquic`.

Figure 8: BBRv3.



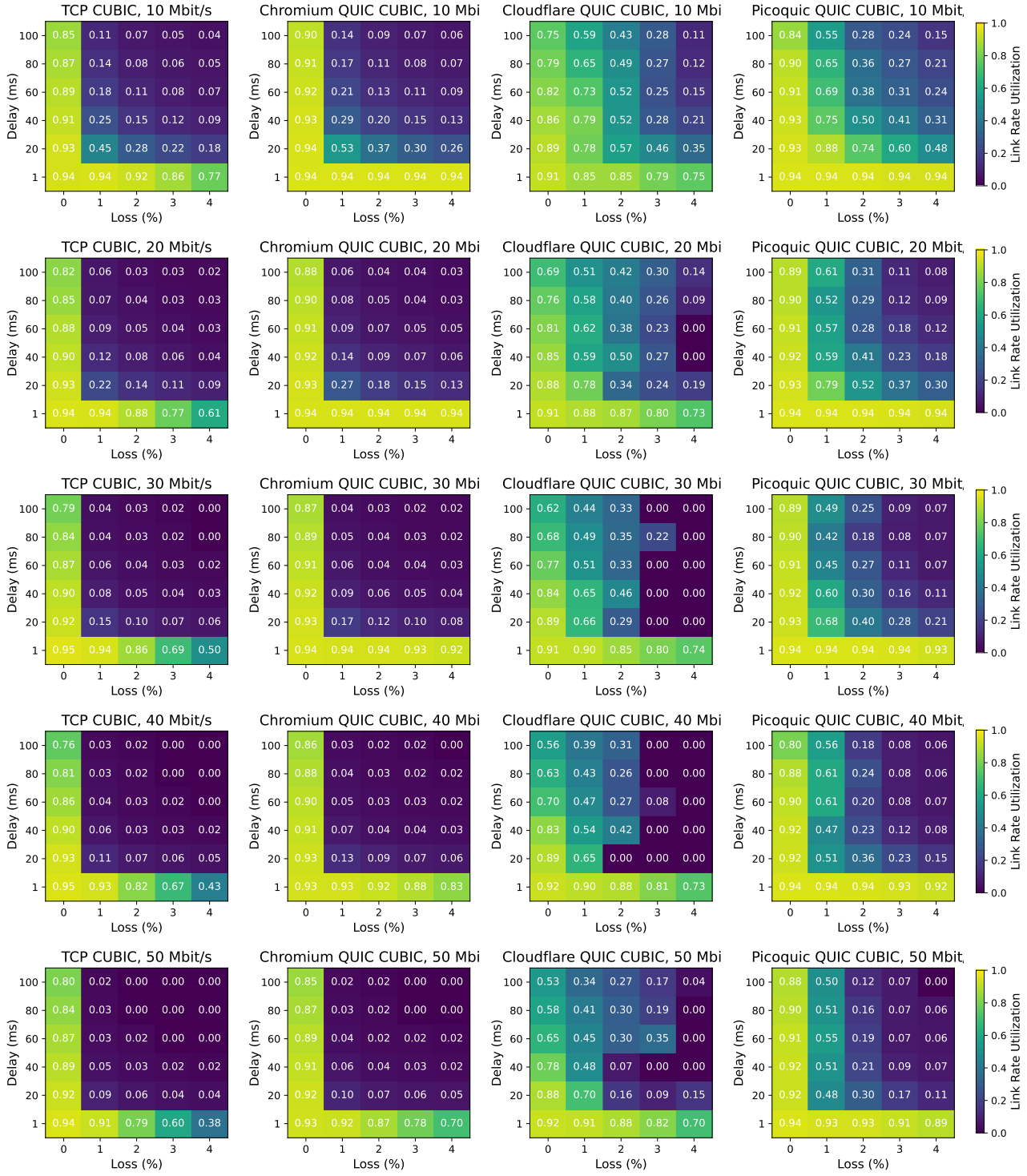
(a) Linux TCP. **(b)** Cloudflare `quiche`.

Figure 9: BBRv2.



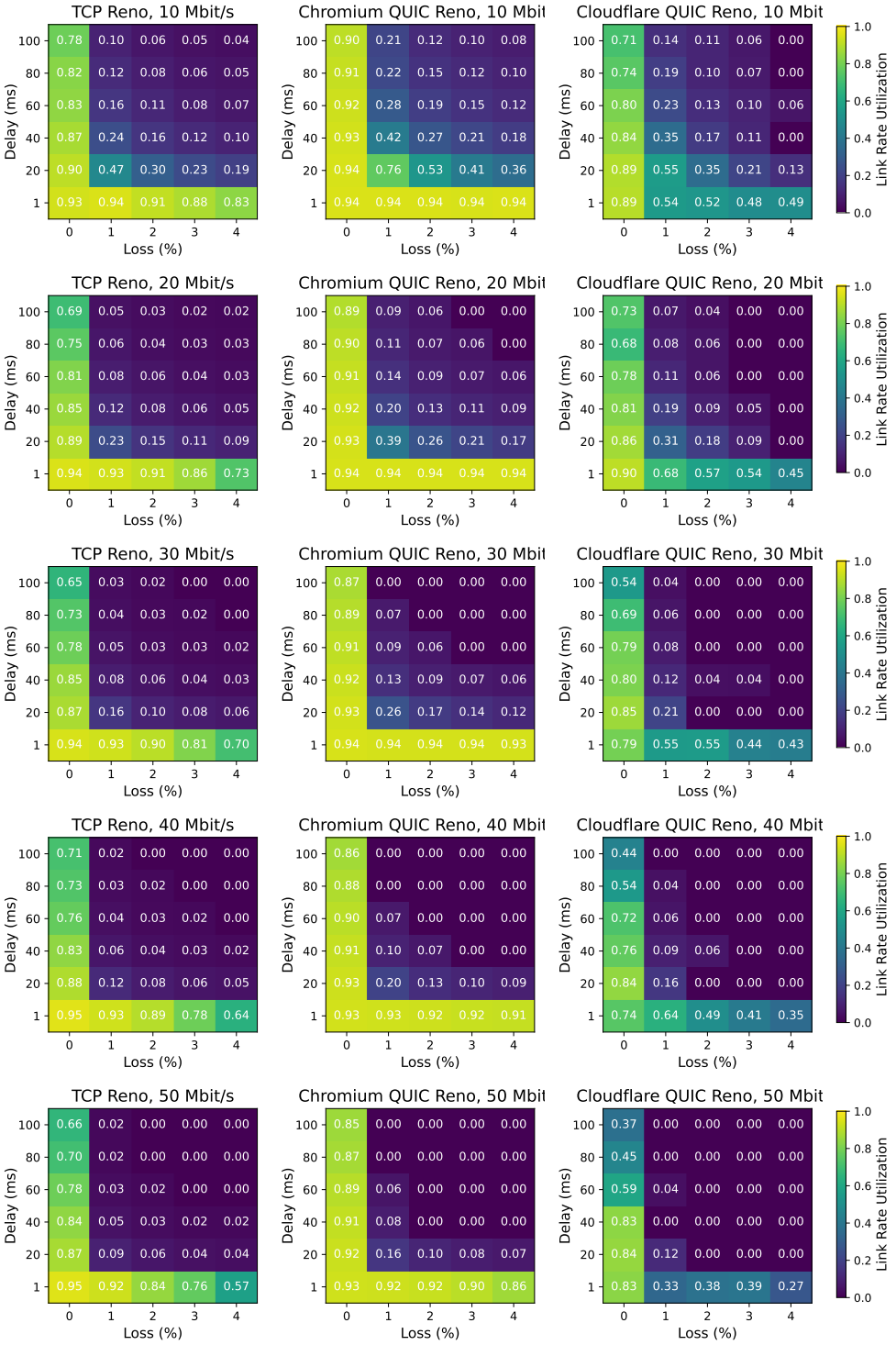
(a) Linux TCP. **(b)** Google `quiche`. **(c)** Cloudflare `quiche`. **(d)** `picoquic`.

Figure 10: BBRv2.



(a) Linux TCP. **(b)** Google `quiche`. **(c)** Cloudflare `quiche`. **(d)** `picoquic`.

Figure 11: CUBIC.



(a) Linux TCP. **(b)** Google `quiche`. **(c)** Cloudflare `quiche`.

Figure 12: Reno.

WIC: Hiding Producer-Consumer Synchronization Delays with Warp-Level Interrupt-based GPU Communications

Jiajian Zhang[1,2], Fangyu Wu[1]*, Hai Jiang[3], Qiufeng Wang[1], Genlang Chen[4], Chaoyi Pang[4]

[1]Xi'an Jiaotong-Liverpool University
[2]University of Liverpool
[3]Beijing University of Posts and Telecommunications
[4]NingboTech University

Abstract

GPU communication plays a pivotal role in collaborative computation across multiple devices. Despite advancements in inter-device communication fabrics and architectures, synchronization still remains a significant challenge due to the manual coordination required between producers and consumers at the application level. In this work, we first reveal that traditional synchronization is a primary bottleneck in GPU communication, where consumers frequently poll for producer data availability. Specifically, early-started polling leads to the unnecessary occupation of computational resources. To address this issue, we propose Warp-level Interrupt-based Communication (WIC), a novel synchronization framework for GPU communication that introduces a fine-grained interruption mechanism at the warp level to replace repetitive polling. WIC preemptively stalls warps engaged in frequent polling and releases computational resources for other warps, thereby effectively overlapping producer-consumer synchronization with ongoing computations. Comprehensive experiments demonstrate that WIC significantly outperforms conventional polling methods by 1.13× on average across various applications with diverse communication patterns.

1 Introduction

GPU communication plays a pivotal role in collaborative computation between CPUs and GPUs [23], especially as the complexity [22, 25, 47] and the scale of parallel computing tasks [13, 44] outpace the capabilities of single-GPU setups [48]. Despite advancements in communication architectures such as *GPUDirect P2P* [32] and *RDMA* [33], and architectures like *NVLinks* [34], synchronization challenges persist prominently within GPU communication [49]. Specifically, programmers must manually synchronize the timing of producer production and consumer consumption [17], typically involving repetitive polling for the availability of producer data before transfer [20, 35, 41, 43], which introduces considerable overhead and complexity into system operations [1, 36].

Traditionally, GPU cross-device communication research focuses on inter-kernel coordination, strategically scheduling computing kernels (k_0-k_n) to overlap synchronization overheads and boost throughput [12, 21], as shown in Figure 1(a). In-kernel optimizations enable asynchronous data transfers without kernel terminations to fine-tune producer-consumer models. Predominantly, these studies emphasize optimizations on the producer side, including scheduling additional transfer threads [31] or employing page placements [9, 30] to dispatch data asynchronously. For the consumer side, a typical approach is GPS [29], which proactively transfers data to the consumer assuming data availability before consumer demand, as shown in Figure 1(b).

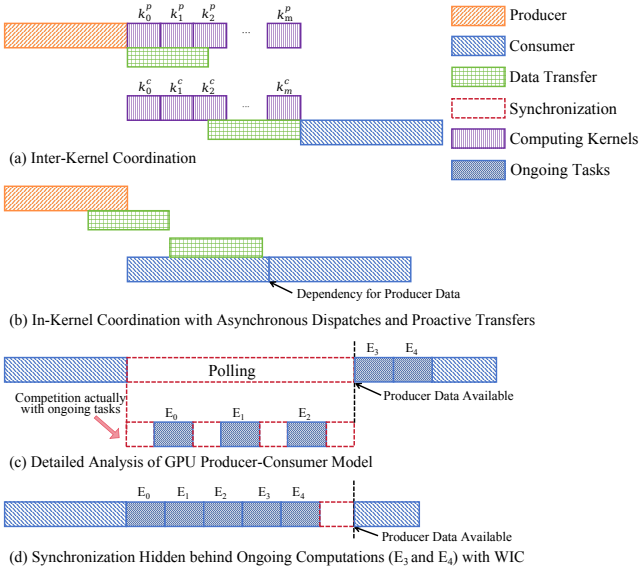


Figure 1: Different Communication Paradigms

However, our experimental results from practical GPU com-

*Corresponding author: Fangyu Wu (fangyu.wu02@xjtlu.edu.cn)

munication applications reveal significant synchronization overhead on the consumer side, primarily due to consumers waiting for the producer's data and repeatedly polling for its availability across devices, as detailed in Section 4. While existing research has contributed valuable insights, it has not fully captured this issue, leading to limited improvements in real-world application performance. To effectively hide the synchronization overhead, this paper proposes a novel GPU communication framework, **W**arp-level **I**nterrupt-based **C**ommunication (WIC), that introduces a fine-grained interruption mechanism at the warp level to replace repetitive polling for producer data availability.

In this work, we first pinpoint the nuances of the GPU producer-consumer model. From a traditional producer-consumer perspective, executable tasks before polling have usually completed, leaving no computational tasks to overlap with synchronization, which makes the effects of polling and interruption seem similar. However, in GPU environments, the situation is markedly different. Consumer kernels are executed across multiple threads with varying completion times. Further exploration at the thread level reveals that early-started polling is a primary cause of synchronization overhead. Specifically, substantial threads remain incomplete when polling begins, causing unnecessary polling to squander computational resources and compete with ongoing tasks (E_0-E_2), as illustrated in Figure 1(c). Furthermore, by the end of polling, many threads remain unfinished, indicating that their unfinished execution (E_3 and E_4) could have overlapped with the synchronization latency.

To mitigate these inefficiencies, WIC preempts warps engaged in frequent polling, thereby releasing their computational resources for other warps. This approach allows unfinished tasks delayed by polling to take precedence, effectively hiding synchronization behind ongoing computations and improving overall GPU communication efficiency, as shown in Figure 1(d). WIC incorporates three key components: the **Interrupter** module, detecting warp requests for producer data and triggers interruptions to swap other executable warps; the **Monitor** module, tracking the availability of the producer data; and the **Activator** module, resuming the stalled warps once the required data becomes available. To manage warp interruptions efficiently, WIC is integrated within the Unified Virtual Memory (UVM) context, employing UVM page placements to transfer communication messages and track data availability. All of these changes are implemented through modifications to the UVM host kernels, allowing programmers to request producer data functioning in a manner akin to a system call, thereby bypassing the need for manual orchestration of synchronization with polling, ensuring full transparency to programmers.

Our contributions can be summarized as follows:

- We present an in-depth analysis of the synchronization overhead within GPU communications, highlighting that early-started polling monopolizes computational resources, the crucial bottleneck in GPU communications.
- We introduce WIC, a novel GPU communication framework that incorporates a finer-grained interruption mechanism at the warp level to replace repetitive polling, aiming to hide the synchronization latency behind the computation.
- We propose a novel classification perspective that shifts from traditional memory access patterns to a focus on communication patterns. Comprehensive experimental results across 10 applications with diverse communication patterns demonstrate significant performance enhancements with WIC, achieving an average speedup of 1.13× compared to conventional methods.

2 Background

2.1 Producer-Consumer Model

The producer-consumer model is a concurrency design pattern that coordinates communication between processes sharing common resources [7]. In this model, the producer generates data and populates a buffer, while the consumer retrieves that data from the buffer, allowing both to operate simultaneously and thus boosting operational efficiency through continuous data flow [18]. However, the model necessitates robust synchronization mechanisms to ensure data integrity, preventing the producer from overwriting unread data and the consumer from retrieving uninitialized data [28]. Effective synchronization is typically achieved using semaphores or monitors [14]. Semaphores include two primary types: counting semaphores, which manage how many slots are occupied or available, and mutexes, which provide mutual exclusion when multiple processes access the buffer. Monitors offer an alternative approach by encapsulating the buffer operations within a single construct that coordinates producer and consumer interactions.

2.2 GPU Cross-Device Communications

Cross-device communications in GPU environments involve multiple GPUs cooperating to share data, a critical process in large-scale applications such as deep learning training, scientific simulations, and graphics rendering [38, 46]. Different from traditional host-side producer-consumer models [5], the simultaneous execution of massive threads in GPU environments necessitates advanced synchronization strategies to manage fine-grained interactions [37]. However, due to the absence of native cross-device semaphores in GPU APIs, programmers must manually coordinate synchronization through CUDA streams, events, or synchronization flags. Typically, producers indicate data readiness via flags, while consumers must actively poll or wait on these signals to maintain data

consistency and avoid race conditions [20, 35, 43]. Our work specifically addresses the polling phase, which incurs substantial overhead on the consumer side.

3 Analysis Methodology

In this section, we introduce a novel classification for communication applications. Based on the classification, we detail the specific applications with various communication patterns selected for analysis and describe the system specifications.

3.1 Communication Patterns

The reliance on traditional memory access patterns for GPU communication analysis [4, 9, 29–31, 42] can be misleading, as these patterns generally provide a macroscopic view of program behavior rather than reflecting the specific dynamics of device interactions. For example, the *Bitonic Sort* algorithm utilizes a random memory access pattern. However, its inter-device communication, essential for the global merging phase, primarily exhibits a gather-scatter pattern, where GPUs fetch pre-sorted data (gather) and then write back the results following comparison and swapping (scatter). Moreover, the communication model is also a critical factor in classifying applications, through its impact on synchronization methods and data flow across communication iterations.

To conduct a more comprehensive analysis of GPU communications, we propose a dual-dimensional classification, *communication patterns*, including communication access patterns and communication models. Communication access patterns are categorized into four types—streaming, adjacent, scatter-gather, and random—each representing different access behaviors during device interactions. Additionally, communication models are defined by the nature of data transfer during communication iterations, as detailed in Table 1.

Table 1: Communication Models

Unidirectional Communication $(D_0 \rightarrow D_1)^N$ This model represents one-way communication between two devices ($D_{0,1}$). D_0 consistently acts as the producer, while D_1 always functions as the consumer. N represents the number of communication iterations.	**Alternating Communication** $(D_0 \rightarrow D_1, \quad D_1 \rightarrow D_0)^N$ Within a single communication iteration, D_0 starts as the producer, sending messages to D_1. After processing these messages, D_1 then acts as the producer, sending the resultant data back to D_0. Both devices alternate roles between producer and consumer throughout a communication iteration.
Multiple Communication $((D_0 \rightarrow D_1)^{n_0}, \quad (D_1 \rightarrow D_0)^{n_1})^N$ This model involves multiple transmissions between devices within a single communication iteration. D_0, acting as the producer, transfers messages to D_1 multiple times (n_0), and after processing, D_1 transfers messages back to D_0 n_1 times.	**Probabilistic Communication** $((D_0 \rightarrow D_1)^{\epsilon_i^0 \cdot n_0}, \quad D_1 \rightarrow D_0)^{\epsilon_i^1 \cdot n_1})^N$ This model characterizes the communication between devices where the mode and frequency of transfers within each communication cycle are probabilistic.

3.2 Applications

To encompass a comprehensive spectrum of UVM scenarios, we select ten applications from FAIR1M [40], Mantevo [26], Comb [6], Polybench [39], Rodinia [8], and Savina [16] benchmark suites as listed in Table 2. This benchmark selection, based on a classification of different patterns, has been demonstrated to avoid *benchmarking crimes* [45] and is widely adopted in GPU communication evaluations [4, 9, 11]. In our study, we further refine this classification to focus more sharply on communication access patterns and incorporate synchronization dynamics into our selection process. The chosen applications fully cover the complete dual-dimensional classification defined in Section 3.1, encompassing four communication access patterns: streaming, adjacent, scatter-gather, and random patterns, as well as four communication models: unidirectional, alternating, multiple, and probabilistic models. Specifically, C_SRS and G_KMN require only unidirectional, streaming patterns, where data is transferred back to a single device. C_CG, C_FE, and G_C2D engage in an Alternating communication model, where two devices exchange information with Random and Streaming access patterns. Multiple transfers during each iteration are necessary for C_H3D, G_BS, G_STN, and G_GEM. For instance, G_STN repeatedly sends boundary data of its subdomain to an adjacent GPU. These applications exhibit Adjacent and Scatter-Gather access patterns. G_BFS, with its unpredictable access manner, fits into a Random and Probabilistic model. Additionally, the inclusion of the communication-intensive application, C_H3D, enriches our evaluation with diverse GPU communication behaviors.

The applications include both CPU-GPU communications (the first four applications listed in Table 2) and inter-GPU communications (the last six applications listed in Table 2).

Table 2: List of Applications

Abbr.	Applications	Benchmark Suite	Comm. Model	Access Pattern
C_SRS	Satellite Remote Sensing Image Processing and Distribution	FAIR1M	streaming	Unidirectional
C_CG	High Performance Computing Conjugate Gradients	Mantevo	Random	Alternating
C_FE	Finite Element Mini-Application	Mantevo	Random	Alternating
C_H3D	Halo 3D Cube	Comb	Adjacent	Multiple
G_BFS	Breadth-first Search	Rodinia	Random	Probabilistic
G_BS	Bitonic Sort	Savina	Scatter-Gather	Multiple
G_C2D	Convolution 2D	Polybench	Adjacent	Alternating
G_STN	Stencil	Polybench	Adjacent	Multiple
G_GEM	General Matrix Multiplication	Polybench	streaming	Multiple
G_KMN	Kmeans	Rodinia	streaming	Unidirectional

3.3 Naive Communication Applications

In GPU cross-device tasks, programmers must manually coordinate fine-grained synchronization to ensure data integrity during producer-consumer communications. To analyze the communication inefficiency, our work adopts a representative GPU producer-consumer paradigm, as detailed in Algorithm 1. Communication resources are divided into subdomains associated with consumer thread block requests, each marked by a *syncFlag*, indicating producer data availability. Both communication data and flags are maintained within UVM, allowing uniform access by producers and consumers.

Algorithm 1 Naive Communications

```
1:  procedure CONSUMERSYNC
2:      Independent Processing (C1);
3:      __syncthreads();
4:      if tid == 0 then
5:          while atomicAdd(&(producer[bid]) -> syncFlag), 0) != 1 do
6:          end while
7:          Access the producer's data through UVM;
8:      end if
9:      __syncthreads();
10: end procedure
11: procedure PRODUCER
12:     Compute producer's data (P1);
13:     __syncthreads();
14:     if tid == 0 then atomicAdd(&(producer[bid]) -> syncFlag), 1);
15:     end if
16:     __syncthreads();
17: end procedure
```

On the consumer side, threads complete preliminary processing ($C1$), synchronizing via *__syncthreads*. Subsequently, the lead thread ($tid = 0$) continuously polls syncFlag in a loop (Line 5). Once the flag indicates data availability, an atomic operation secures the syncFlag from producer-side updates, enabling the lead thread to safely access the corresponding data (Line 7). Remaining threads wait via *__syncthreads* until the lead thread completes this communication step. On the producer side, once data production completes, the corresponding syncFlag is set to 1. This communication model involves three distinct UVM page faults: the consumer polling an unavailable syncFlag, updating syncFlag upon data readiness, and transferring producer data.

3.4 System Specifications

CPU-GPU Communication Setup consists of an NVIDIA RTX 4090 GPU with 24GB of DRAM, coupled with a high-performance CPU, the 14900KF, featuring 24 cores.
Inter-GPU Communication Setup consists of four NVIDIA $A800$ GPUs, each with 80GB of DRAM, coupled with two high-performance CPUs, Xeon 6138, each featuring 20 cores.

4 Motivation

In the motivation section, we first establish that consumer-side overhead significantly impedes efficiency in producer-consumer communications. We then identify that repetitive polling is the predominant source of this overhead. Moreover, a detailed examination of the polling phase at the thread level reveals that polling initiated too early causes considerable resource contention, which further indicates that these resources could have been better utilized by overlapping with synchronization latency, allowing other executable tasks to proceed without delay. Our experimental setup for collecting data is discussed in Section 3.

4.1 Critical Focus on the Consumer Side

Observation 1: The consumer side exhibits significantly higher latencies compared to the producer side. Figure 2 shows the proportion of communication latency for each application on both the consumer and producer sides. For the producer side, latencies are consistently low, generally staying within 10% across various applications, and the differences in latency proportions among these applications are relatively minor. In contrast, the consumer side displays markedly higher latency proportions, typically ranging from 20∼ 50%. This discrepancy is most pronounced in the C_H3D application, where its communication-intensive nature results in a consumer-side latency peaking at 81.97%.

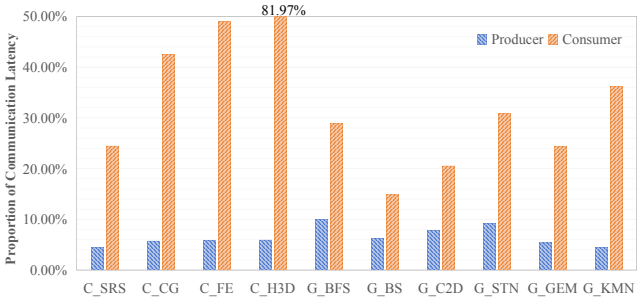


Figure 2: Proportion of Communication Latency on both Consumer and Producer Sides across Different Applications

The higher latency on the consumer side primarily stems from additional synchronization requirements. On the producer side, Direct Memory Access (DMA) enables asynchronous operations that permit data transfers without halting ongoing computational tasks, effectively minimizing communication latency. However, the situation on the consumer side presents more complexities. The consumer kernel verifies the

physical presence of producer data on the local device prior to processing, inherently introducing additional latency.

This asymmetry highlights a pivotal direction for communication optimizations: the consumer side, as the primary contributor to communication latency, demands targeted efforts. While existing research largely focuses on optimizing the producer side [9, 30, 31], addressing the latency issues more prevalent on the consumer side holds significant potential to boost the overall efficiency of GPU communications.

4.2 A Deeper Investigation into the Consumer-Side Latency

The consumer-side communication can be analyzed by categorizing it into two distinct scenarios based on the availability of producer's data. Firstly, we divide the consumer kernel into two parts: C_1 and C_2. The C_1 portion operates independently of the producer's data and is typically used for auxiliary tasks or computations from previous communication iterations. C_2, however, depends on data generated by the producer in the current iteration. The completion time of C_1 is denoted as T_c, and the producer's data generation is denoted as T_p. Two scenarios are distinguished based on the relationship between T_c and T_p, specifically assessing whether the consumer must wait for the producer's data to become available.

Scenario 1: T_p occurs before T_c, as shown in Figure 3(a). In this scenario, the producer's data is pre-available, allowing the consumer to directly copy the data without availability checks. Transfer time (W_t) can be overlapped with the C_1, by predicting the data dependency and proactively transferring the data to the consumer in advance.

Scenario 2: T_c occurs before T_p, as shown in Figure 3(b). In this scenario, the producer's data is not available when the consumer needs it, requiring the consumer to repeatedly check its status (*polling*). The total waiting time includes both the polling (W_p) and data transferring time (W_t).

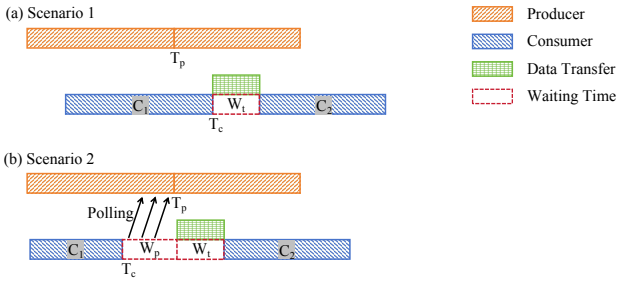


Figure 3: Two Scenarios based on the Availability of Producer's Data

To investigate consumer communication scenarios across real-world applications, we analyze the average time differences between T_p and T_c along with their standard deviations, where positive values indicate T_c occurring before T_p, while negative values indicate the reverse. Given substantial variability in these time differences, we normalize them to the communication task duration to provide a clear impact of these delays, as shown in Figure 4.

Observation 2: In most applications, consumers face substantial waiting times for producer data, with significant variability across and within applications.

Specifically, all applications except G_BS exhibit positive values, indicating that in most cases, the consumer's T_c occurs before the producer's T_p. Moreover, except for the G_BFS application, the time difference for the other applications exceeds 20%, suggesting that consumers typically face considerable waiting time until producers finish generating data. The large standard deviations for C_CG, C_FE, and G_BFS highlight significant variability in waiting times in the same applications. For C_CG and C_FE, the variation is primarily due to the alternating computation tasks between the CPU and GPU. Meanwhile, G_BFS's high variability is due to its unpredictable access manner, leading to more random data availability timings.

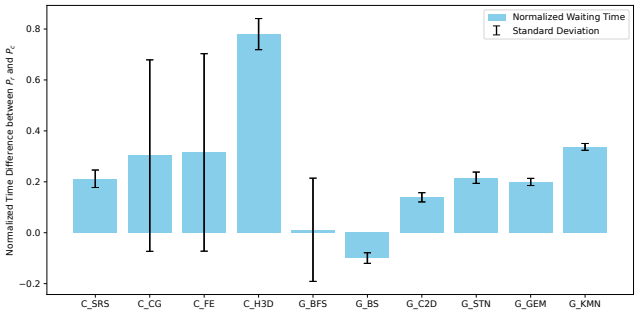


Figure 4: Normalized Time Difference between Production (T_p) and Consumption (T_c)

Based on the observation, we conclude that *Scenario* 2, where the consumer is required to wait for the producer's data, is more common. Furthermore, due to the uncertainty and potential variability of data generation on the producer side, the polling mechanism employed in *Scenario* 2 is crucial for ensuring stable and correct program execution.

Observation 3: Consumer-side polling, where consumers repeatedly check the availability of the producer's data, constitutes a significant portion of the consumer communication overhead in *Scenario* 2. Figure 5 illustrates both the proportion of polling within the overall consumer communication overhead and the average number of checks required per communication. Except for G_BFS, polling accounts for over 60% of the consumer communication overhead, with applications, C_H3D and G_KMN, exceeding 90%. Additionally, the number of checks per communication is extremely high, typically in the range of millions to tens of millions. This

extensive polling overhead arises from necessary inter-device communication.

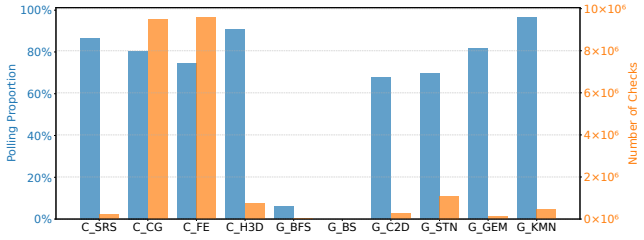


Figure 5: The Polling Proportion within Consumer Communication Overhead and the Average Number of Checks Required per Communication

Summary of Consumer-Side Latency: The high latency on the consumer side is primarily due to consumers waiting for the producer's data and repeatedly checking its availability with cross-device communication.

4.3 A Finer-Grained Analysis on Polling

The producer-consumer model in GPU architectures significantly diverges from traditional CPU-based setups. In CPU systems, a producer or consumer task tends to be assigned to a single process. Therefore, when reaching the T_c in *Scenario* 2, there are no additional tasks within the consumer that can be pipelined to fill the waiting time gap. Consequently, the overhead associated with polling cannot be optimized further. However, in GPU architectures, the consumer kernel is executed by a multitude of parallel threads. The non-uniform completion progress of these threads, orchestrated by the GPU's *warp scheduler*, often means that not all threads reach the T_c simultaneously during the polling phase. Moreover, some *independent threads* may not depend on the producer's data and thus do not need to repeatedly check for its availability.

To deepen our understanding at the thread level, we analyze the proportion of threads that have not reached the T_c and the proportion of independent threads remaining incomplete as polling progresses for four applications, C_SRS, C_CG, G_BFS, and G_STN, with different communication types.

Observation 3.1: Consumer's polling significantly impacts the completion of computational tasks before reaching the T_c. This observation can be inferred from the results shown in Figure 6, supported by three key insights as follows:

- At the onset of polling, substantial threads still completing their C_1 tasks have not reached T_c, causing them to compete for computational resources with polling.
- The curve features several nearly horizontal segments, signifying substantial preemption of computational resources for these threads. This preemption results in stagnation, as polling dominates the limited resource within the device.
- Even after the completion of polling, a large number of threads still have not reached T_c.

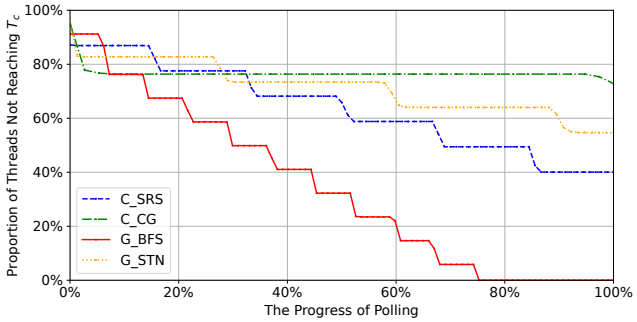


Figure 6: The Proportion of Threads not Reaching T_c as the Polling Progress

Observation 3.2: Consumer's polling significantly disrupts the computational tasks of independent threads. Despite these threads comprising a smaller proportion of the overall program, they suffer prolonged stagnation due to resource preemption by polling activities, particularly in applications like G_BFS and G_STN, as shown in Figure 7. Furthermore, these independent threads still fail to complete their tasks promptly even after polling ends.

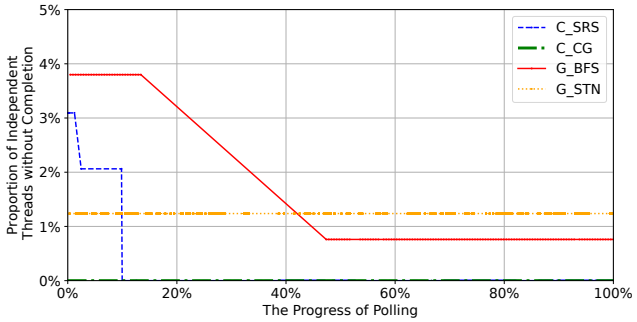


Figure 7: The Proportion of Independent Threads without Completion as the Polling Progress

The analysis of consumer-side polling reveals significant inefficiencies in resource utilization, with early-started polling significantly tying up GPU computational resources and delaying tasks. These include tasks from threads completing C_1 operations and those from threads that operate independently of polling demands, suggesting that synchronization overhead could be effectively overlapped with these ongoing tasks.

5 Design of WIC

5.1 Interrupt-based Mechanism at Warp Level

Based on the analysis in Motivation, it is advised to delay polling to overlap synchronization delays with other executable tasks. However, directly intervening in GPU runtime kernels to reschedule computational resources is impractical for three main reasons:

- Determining an optimal execution sequence is challenging due to the difficulty in accurately predicting thread dependencies at runtime.
- Rescheduling the execution order of parallel threads can disrupt the inherent parallelism of the program.
- GPU APIs typically do not provide direct access for rescheduling resources.

We propose an interrupt-based synchronization framework, WIC, at the warp level to optimize computational resources by prioritizing executable tasks. Specifically, WIC preemptively stalls warps engaged in a blockage due to unavailability of data, switching to another warp that is ready to execute. This mechanism hides the synchronization latency and enhances GPU resource utilization by ensuring that all executable tasks are processed before waiting, minimizing idle times.

WIC incorporates three key components: Interrupter, Monitor, and Activator, as illustrated in Figure 8. The **Interrupter** prevents warps from being blocked by triggering interruptions when they request producer data and switching to other warps. The **Monitor** tracks data availability and notifies the Activator when the data is ready. The **Activator** then facilitates data transfer to the consumer side, and resumes the stalled warps.

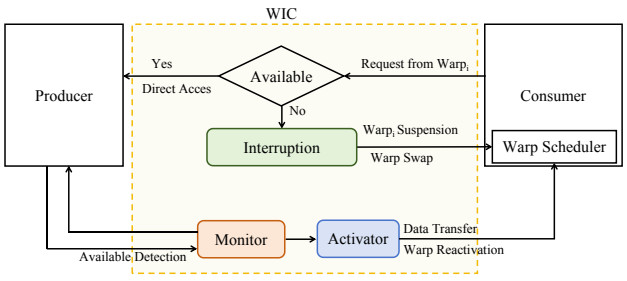


Figure 8: The Overall Workflow of Warp-Level Interrupt-Based Communication (WIC)

5.2 Integration with UVM

To efficiently manage interruptions, we propose integrating WIC within the UVM context, employing UVM page placements to preempt blocked warps by modifying open-source UVM kernel drivers, without the need for additional GPU hardware intervention.

The UVM system uniquely offers explicit control over the *stalling* and *replaying* of warps in UVM kernel drivers: **On the GPU side**, when warps attempt to access UVM pages that are not present on the local device, the UVM system stalls these warps and schedules other warps for processing. **On the host side**, UVM drivers can send replay signals to the GPU to reactivate the stalled warps.

By artificially inducing UVM page faults, WIC can interrupt warps waiting for producer data. By pushing a replay signal to the consumer kernel, WIC can dynamically reactivate these stalled warps upon data available. Additionally, by transferring producer data directly onto the faulting pages, WIC enables efficient producer-consumer communication without requiring extra transfers.

Figure 9 illustrates the integration of the WIC within the UVM system in detail. WIC defines a specific memory space within UVM, which comprises two key components: the Producer-Consumer Communication Medium (PCM) and Producer Availability Tags (PAT).

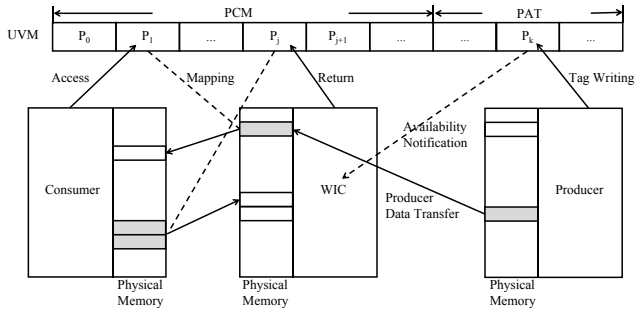


Figure 9: Integration with UVM

PCM is essential for transfer management between producers and consumers and interruption supports for blocked warps. Memory in PCM is allocated on a per-page basis, with free PCM pages residing on the host side to ensure they are available for triggering page faults in consumer-side warps. When warps request producer data, WIC directs them to access PCM pages, triggering a page fault that stalls the warps. These faults are processed on the host side further, where the WIC writes the producer data into the corresponding pages. Once the transfer is complete, WIC signals the consumer kernel to resume stalled warps. After consumption, the pages are returned to the host for future use in WIC communications.

PAT is employed to notify WIC when the producer's data is ready for transfer in inter-GPU communications. Rather than repeatedly polling the PAT, WIC relies on page placement to determine data availability. Specifically, data tags are stored as individual pages on the host side. When the producer completes data preparation, it modifies the tag, trig-

gering a page fault. The host-side UVM driver then captures the tag's address, allowing the identification of the corresponding producer-available data. After the data is consumed, the tags are recycled by the host for future use.

Prefetching options for PCM and PAT within UVM memory are disabled to prevent other memories from proactively migrating to devices, which would otherwise hinder the triggering of necessary faulting.

5.3 Interrupter

The Interrupter module manages PCM page allocations for communication and provides interruption support for warps. To optimize transfer utilization, the Interrupter batches 16 warps at a time, allocating the contiguous PCM pages based on $\lceil totalSize/pageSize \rceil$. After allocation, it registers the page information, including the starting address and length, along with the requested data details, with the WIC. The Interrupter then directs the warps to access the starting address of these pages, triggering a page fault and stalling the warps.

To efficiently locate continuous free pages within PCM, the Interrupter incorporates a *segment tree*. Typically, PCM encompasses $16K$ pages, capable of handling up to $256M$ of communication data. The availability of these pages is tracked using a boolean array, where a '0' indicates an allocated page and a '1' denotes a free page. A $64K$ tree array is used to build the segment tree, where each node represents the state of a segment of pages, including the total number of free blocks (*sum*), the length of the longest contiguous block of free pages (*maxLen*), the length of contiguous free pages from the left boundary (*leftLen*), and from the right boundary (*rightLen*). When querying for k consecutive free pages, if the root node's *maxLen* is less than the required length, it indicates a lack of available space. Otherwise, the search proceeds recursively from the top down, checking *maxLen* of left and right child nodes, as well as the potential contiguous length across the boundary of the two segments (*leftNode.rightLen* + *rightNode.leftLen*). The traversal for each search descends one level down the tree hierarchy, making the query time complexity $O(\log n)$. Similarly, updating the status of a page also starts from the root and proceeds recursively down to the leaf node, maintaining the same time complexity of $O(\log n)$.

5.4 Monitor

The Monitor module, deployed within the host driver, oversees the fault entries retrieved from the GPU fault buffer, specifically capturing PCM and PAT faults. It utilizes PCM faults to manage producer-consumer communication processing and leverages PAT faults to notify the availability of data from the producer.

The Monitor module maintains two key data structures: the Data Availability Bitmap (DAB) and the Pending Faults Queue (PFQ), as shown in Figure 10. Each bit in the DAB corresponds to an availability tag for the producer data. The PFQ holds pending faults due to unavailable data, with each entry containing the fault address and the associated data tags indicating unavailability.

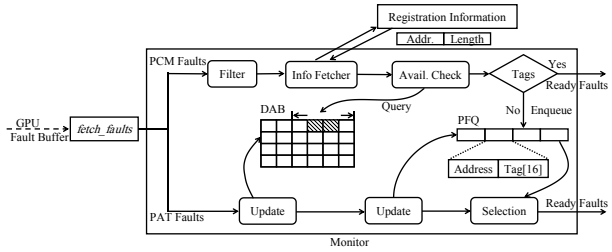


Figure 10: The Monitor Module in WIC

The Monitor detects PCM and PAT faults by checking if the fault address falls within the registered memory block ranges, which are stored in the WIC when these blocks are defined. Upon capturing a PCM fault, the Monitor assigns an independent thread to manage the communication task, ensuring that overall process flow is not disrupted. It first filters out duplicate faults, then reads the required producer data addresses from the Registration Information and checks the corresponding availability tags in DAB. If any tags are false, the fault and corresponding tag IDs are enqueued in the PFQ, suspending the faulting process. If all tags are true, the Monitor notifies the Activator module to proceed.

Monitoring the availability of producer data occurs in two scenarios. For CPU-GPU communications, where the producer is a user process on the host, it updates the DAB directly after data production, prompting the Monitor to update the unavailable tags in the PFQ and verify if fault processing can proceed (i.e., ensuring no unavailable tags remain). Once ready, the Monitor resumes fault processing. Alternatively, for inter-GPU communications, where the producer kernel resides on another GPU, the Monitor detects producer data availability through PAT faults. After the producer finalizes data, it modifies tags in the PAT on the host side. Upon capturing a PAT fault, the Monitor updates the DAB and PFQ with the new available data indicated by the fault address. If a fault process in PFQ is ready, the Monitor activates it and notifies the Activator module to proceed with the next steps.

5.5 Activator

The Activator module fulfills two functions: activation and recycling. Its activation role entails reactivating pending fault processing and reactivating stalled consumer-side warps following the fault processing. The recycling function reclaims PCM pages from the consumer and returns them to the host, ensuring their availability for future communications.

Activation: Upon receiving a signal from the Monitor that

a pending fault is ready, the Activator resumes fault processing and transfers the producer data onto designated PCM pages. If the producer kernel resides on another GPU, the data is transferred via DMA once the physical GPU address is verified. After all data has been written, the Activator services the fault and sends a replay signal to the consumer device, which reactivates the stalled warps. These warps then access the locally available PCM pages to retrieve the producer data, thus completing a cycle of producer-consumer communication.

Recycling: At the end of a communication cycle, PCM pages holding producer data often remain on the consumer side. To ensure proper triggering of page faults by the Interrupter module in subsequent requests, it is essential to recycle these pages back to the host. The Activator addresses this by sending an invalidation signal from the host to notify the consumer device that these pages have been updated. However, the timing of this signal is not precisely controlled to fit between the completion of data retrieval by consumer warps and the subsequent request from the Interrupter module. To address this, each PCM page P is paired with a companion page P', serving alternately in communication cycles to ensure effective and accurate invalidation. Specifically, when the Activator issues a replay signal for P, it concurrently sends an invalidation signal for P'. Once the consumer-side warps have successfully retrieved data from P, P' becomes invalidated. In the subsequent cycle, the Interrupter allocates P' to trigger interruptions, and the Activator prepares P for the next cycle by pushing its invalidation signal, maintaining continuous communication flow.

5.6 Putting All Together

Figure 11 provides an overview of the entire process of WIC. It begins when consumer warps request producer data via the Interrupter module, which then allocates free PCM pages based on the data size (i) and guides warps to access these pages, thereby triggering a page fault and stalling the warps (ii). On the host side, the Monitor module captures these PCM faults and checks for the availability of the required producer data (iii). If the data is unavailable, fault processing is suspended, and the faults are queued in the PFQ to await data availability. Conversely, if the data is available, the faults progress to the Activator for processing. Simultaneously, the Monitor tracks producer data availability by monitoring PAT faults, updating the local DAB, and scanning the PFQ for any ready faults (iv). These ready faults are then reactivated and forwarded to the Activator module for processing. The Activator writes the producer data into the designated PCM pages (v) and subsequently issues replay signals to reactivate the stalled consumer warps, while also recycling these pages for future communications (vi).

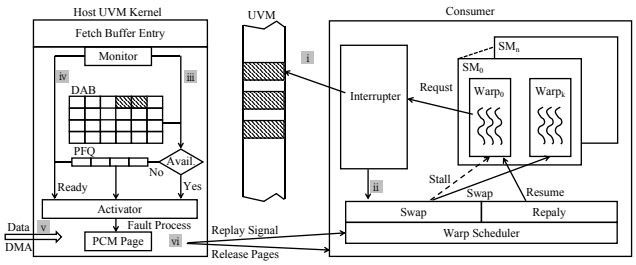


Figure 11: Overview of WIC

5.7 Adaptability of WIC

The WIC's core principle of preemptive scheduling, although primarily implemented within the host-side UVM context, is adaptable to other GPU communication architectures, such as NVLink and Direct P2P. By orchestrating warp interruptions through the communication control path, WIC introduces a fine-grained, non-blocking synchronization mechanism that significantly alleviates the synchronization overhead inherent to traditional polling methods. This design is independent of the specific control path used; thus, if the communication path is optimized beyond the host-side UVM kernel, WIC's interruption management can similarly be integrated into alternative mechanisms, such as capturing the interrupted faults in the PCI-e bus protocol for Direct P2P communication.

6 Evaluation

In this section, we first evaluate the overall performance of WIC against traditional cross-device communications. To delve deeper into its detailed effectiveness, we focus on whether WIC allows more executable tasks to overlap with synchronization time, a concern highlighted in Section 4. Furthermore, we examine the overhead associated with WIC and evaluate its scalability across different numbers of parallel threads. Finally, we compare WIC with state-of-the-art (SOTA) systems for cross-device communication. Our evaluations employ ten applications with diverse communication patterns, conducted on both CPU-GPU and inter-GPU platforms, as detailed in Section 3.

6.1 Overall Performance

We evaluate the effectiveness of our WIC method against traditional cross-device communications. Figure 12 presents the performance enhancements achieved with WIC, as well as shifts in communication overhead proportions. From these results, we derive three key observations.

Firstly, WIC significantly enhances the overall communication performance. As depicted in Figure 12(a), except for G_BFS and G_BS, most applications exhibit about 10% im-

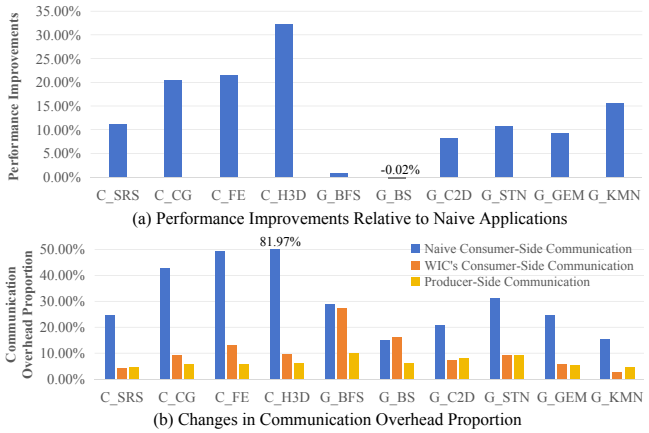


Figure 12: Performance Improvements and Communication Overhead Proportion

provements. The effect is especially significant in CPU-GPU communications, where applications such as C_CG, C_FE, and C_H3D experience enhancements exceeding 20%. For the communication-intensive application, C_H3D, optimization of communication through WIC can lead to substantial performance gains, with increases surpassing 30%.

Secondly, WIC is compatible with both communication scenarios described in Section 4.2. It is ideally suited for *Scenario* 2, where consumers await producer results, but it also supports G_BFS and G_BS applications correctly. G_BS operates entirely under *Scenario* 1 with pre-generated data, and G_BFS involves both scenarios during runtime. This versatility benefits from the Monitor module of WIC, which promptly checks for data availability and activates the Activator for immediate fault processing. However, for G_BS, WIC does not enhance performance, showing a slight decline, primarily because it only maintains a copy of producer availability on the host side without pre-fetching data to the device. Additionally, the overhead with WIC slightly detracts from performance compared to naive communication applications.

Thirdly, WIC significantly decreases consumer-side communication overhead, thereby enhancing program performance. Specifically, compared to naive implementations in Figure 12(b), WIC reduces the consumer-side overhead by about 20%, nearly equalizing the proportion of overhead with the producer-side. Notably, applications with communication access patterns characterized as streaming, such as C_SRS and G_KMN, exhibit minimal communication overhead with WIC. This efficiency is attributed to WIC's Interrupter module capability to batch PCM pages based on their regular access patterns. Conversely, applications with probabilistic communication models and random access patterns generally show a larger proportion of overhead.

6.2 Detailed Analysis

We validate the detailed effectiveness of our method by evaluating its ability to complete more executable tasks on the consumer side before initiating unnecessary waiting, thus overlapping synchronization time. Our analysis concentrates on two aspects: the completion status of threads before waiting and the synchronization time overlapped with computations.

In the WIC framework, we redefine unnecessary waiting times as periods when no active tasks are executed on the consumer end. We track the completion of executable threads from the moment they first request data from the producer until they commence waiting. As GPUs do not provide direct statistics on completion times for all threads, we use *Nsight System* to approximate executable task completion times by tracking when the threads trigger WIC's PCM faults. Figure 13 illustrates the completion status of PCM faulting threads as they wait for producer data availability, covering four applications with diverse communication patterns as detailed in Section 4.3. The results demonstrate that WIC allows the majority of tasks to be completed before the waiting time, with the curves showing that the tasks are completed in advance of waiting periods. Furthermore, the absence of prolonged flat segments in the completion curves under WIC indicates that there are no significant stalling states, thus confirming minimal unnecessary waiting times.

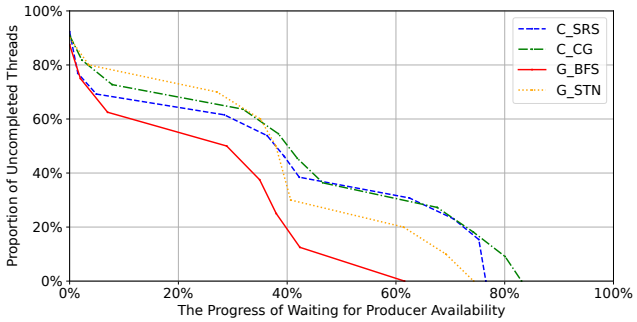


Figure 13: Uncompleted PCM Faulting Threads as Waiting for Producer Data Availability

The results in Figure 14 demonstrate that WIC significantly enhances the communication efficiency by the significant proportion of synchronization hidden behind computation. Different from naive applications, which typically achieve only about 10% overlapping efficiency, most applications equipped with WIC achieve an average overlapping efficiency exceeding 80%. Notably, C_H3D and C_FE exhibit the largest gains at approximately 85%, while even the less significant improvements in applications like G_STN and G_KMN are around 70%. This boost in overlapping efficiency demonstrates WIC's capability to pipeline more computational tasks effectively. Specifically, this improvement is achieved by re-

placing traditional polling with interruptions and its granular scheduling at the warp level. Furthermore, WIC is particularly adept at handling CPU-GPU communication tasks, where the proportion of hidden synchronization reaches up to 90%, compared to 80% in inter-GPU communications. This advantage is largely attributed to WIC's integration with the host-side UVM kernel, enabling more flexible task scheduling and overlapping, and its Monitor module's efficient checking of producer data availability, allowing producer data to be more readily accessible. These benefits depend on the current WIC implementation, and performance might vary if WIC's communication control path were based on other architectures like NVLink or PCI-e bus protocols.

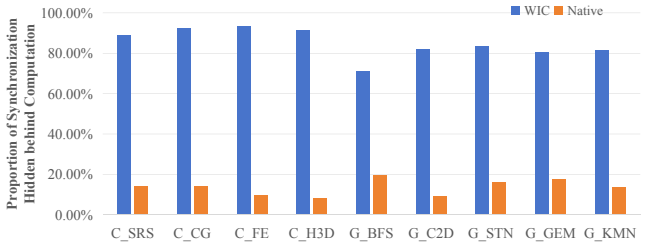


Figure 14: Synchronization Overlapped with Computations

6.3 Overheads

WIC's overhead is primarily divided between the consumer and host sides. On the consumer side, the main overheads originate from the Interrupter module's allocation of PCM pages. On the host side, the overheads are mainly attributed to the Monitor module's role in suspending and awakening page fault processing, as well as checking and updating the DAB for producer availability. We evaluate these specific overheads, along with the total overhead associated with WIC, as illustrated in Figure 15.

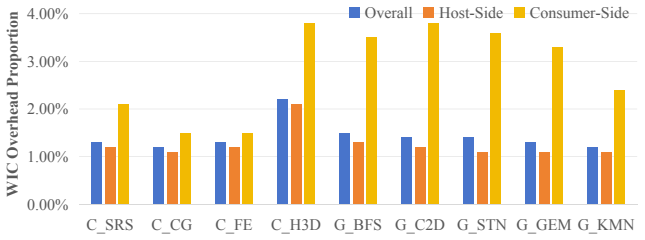


Figure 15: Overhead Associated with WIC

The overall overhead associated with WIC is relatively minor and maintains remarkable stability across different applications. Apart from C_H3D, where overhead is slightly higher, the overheads for other applications consistently hover around 1.4%, with deviations not exceeding 0.2%. Despite relatively higher overheads on the host side, the total overhead is minimally impacted thanks to WIC's interruption mechanism, which schedules other executable warps to overlap with host processing time.

Consumer-side overhead closely mirrors the total overhead and consistently remains stable across various applications, regardless of their differing communication patterns. This stability is supported by the Interrupter module's utilization of a *segment tree* for PCM page management, ensuring efficient and consistent identification of free pages with a time complexity of $O(\log n)$.

Host-side overhead is relatively higher, averaging around 3%, and demonstrates variability according to different communication patterns of applications. Inter-GPU communications often incur about 1-2% higher overhead than CPU-GPU communications due to the complexities involved in transferring producer data and monitoring availability across devices. Applications characterized by multiple communication models, such as G_STN and G_GEM, show the highest overheads, reaching up to 3.6% and 3.4% respectively, due to frequent demands for data availability checks required by multiple transfers. Conversely, applications like G_KMN, exhibit lower overheads at 2.4%, benefiting from relatively regular access behaviors.

WIC introduces no additional overheads for page faulting compared to naive applications. In naive setups, page faulting occurs three times per communication cycle: reading the availability tag for producer data, copying the tags after updates, and transferring the producer data. In contrast, WIC simplifies this process to just one PCM fault on the consumer side. If the producer kernel is on another GPU, an additional PAT fault should be required. Beyond this, WIC leverages UVM's direct copy for all other cross-device data transfers, treating the data in a read-only fashion at the receiving end. WIC ensures these transfers maintain transactional integrity, thus eliminating the need for faulting to safeguard transmission.

6.4 Scaling Performance

In the scaling experiment, we evaluate the performance of WIC as the number of threads is escalated from 64 to 1 million across four applications with distinct communication patterns. The results, as illustrated in Figure 16, demonstrate that WIC consistently boosts performance even under extensive thread scaling. Specifically, performance speedups in applications with thread counts from $1K$ to 1 million threads remain near peak values. However, despite this overall stability, there is a slight performance decline as thread counts increase. For instance, C_CG exhibits approximately a 5% decrease from its peak performance. This performance limitation is attributed to the UVM's capacity to handle only 256 faults at a time. When WIC allocates more than 256 PCM faults for consumer warps, any surplus faults must queue,

creating a bottleneck. Although the Interrupter module attempts to mitigate this issue by batching warps' requests to minimize the number of faults, blockages may still occur as producer-consumer communication escalates. However, applications like G_STN, benefit from quick producer responses to PCM faults, mitigating some scaling issues. In smaller-scale scenarios, particularly from 64 to 512 threads, WIC shows little benefit due to insufficient executable warps to effectively overlap synchronization overhead.

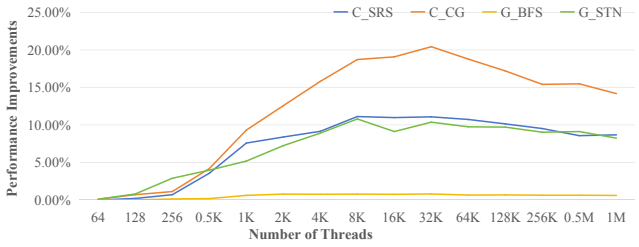


Figure 16: Performance for Scaling Experiment

6.5 Compared to State-of-the-art

In GPU communication research, SOTA systems are primarily categorized into CPU-GPU and inter-GPU communications. For comparison, several well-regarded systems that support in-kernel communications serve as baselines. For CPU-GPU communications, systems like NBlocking [10] and DemandCpy [19] are considered, whereas EffShare [31], GPS [29], and FinePack [30] are evaluated for inter-GPU communications. Notably, systems like DemandCpy, GPS, and FinePack, which involve hardware modifications, are not feasible in actual GPU environments and are therefore reimplemented at the application level for this analysis.

Figure 17 presents a performance comparison between WIC and various SOTA systems, normalized to naive setups. The results indicate that WIC outperforms most SOTA systems in various applications, with the exceptions of G_BS and G_BFS, where it slightly lags behind EffShare and GPS. Specifically, in CPU-GPU communication applications, WIC outperforms NBlocking by an average of 20% and DemandCpy by 15%. This superior performance is attributed to WIC's use of an interruption-based mechanism to overlap synchronization times between the host and device, contrasting with the simple busy-wait synchronization used by other systems. Furthermore, WIC focuses on enhancing producer-communication across various communication patterns, which contributes to its strong performance across different applications. Conversely, NBlocking's strategy of utilizing kernel fusion shows limited success in applications with multiple communication models, such as C_SRS and C_CG. DemandCpy, tailored for unidirectional transfers by replacing in-kernel demand page requests with direct *memcpy*, struggles with bi-directional communication models.

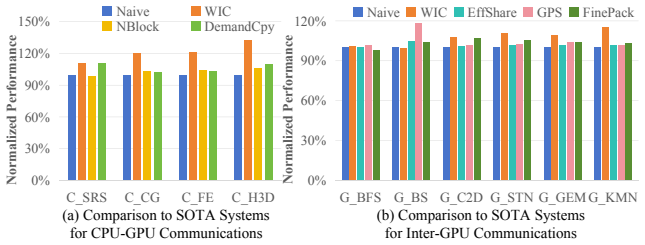


Figure 17: Comparison to State-of-the-art Systems

For inter-GPU communications, WIC demonstrates a 15% performance lead on average, with the exceptions of G_BS and G_BFS. In G_BS, EffShare and GPS perform better due to their strategies of pre-copying producer-generated data to the consumer side, effectively reducing consumer waiting times. FinePack, which proposes packing small and frequent transfers, shows adequate performance in scenarios with regular communication access patterns like streaming or adjacent. However, these systems still depend on cross-device polling when waiting for producer data, which incurs significant overheads, making their overall performance inferior to WIC.

7 Related Work

Optimizations in cross-device communication are predominantly classified into two categories: inter-kernel and in-kernel coordination.

7.1 Inter-Kernel Coordination

StarPU [2] first highlighted the significant impact of synchronization overheads, like data transfers, on CPU-GPU computational performance and introduced strategies such as Greedy to manage parallel tasks effectively. Building on this, Lustig et al. [27] further developed schemes by striking the balance between data transfer and computation, effectively minimizing idle times. Shifting away from static predictions, GPUSync [11] adopted a priority scheduling system that dynamically adjusted based on real-time monitoring of execution budgets. To analyze the intricate dependencies of tasks more precisely, Taskflow [15] employed an expressive task graph programming model incorporating in-graph control flow to facilitate task management. Furthermore, Bak et al. [3] expanded on this model by integrating gang-scheduling with work-stealing techniques, aiming for a more effective balance of workloads across tasks.

Despite the throughput improvements by overlapping more discrete kernels, inter-kernel strategies do not inherently enhance the performance of individual producer or consumer

kernels. Furthermore, the process of launching and terminating kernels imposes additional overheads, further complicating efficient execution within inter-kernel strategies.

7.2 In-Kernel Coordination

Koukos et al. [24] pioneered a fine-grained CPU-GPU collaborative architecture leveraging the UVM framework to lower cross-device communication costs. However, DemandCpy [19] criticized the substantial overhead associated with UVM's page placements, and proposed a novel CPU-GPU transfer method that combined traditional blocking *memcpy* with asynchronous page transfers, although it struggles with bidirectional read-write operations. Alternatively, NBlocking [10] explored an in-kernel communication strategy by merging kernels and employing UVM's zero-copy feature for asynchronous data transmission to GPUs. FinePack [1] tackled the underutilization of UVM's transfers by suggesting that transfer messages be accumulated in a buffer and batched. However, these approaches generally overlooked synchronization issues between producers and consumers.

EffShare [31] assigned dedicated *transfer threads* to manage producer-consumer data transfers, enabling producers to write results directly into global memory and continue with other computational tasks, promoting asynchronous data dispatch. However, the synchronization required to activate these transfer threads still introduces significant overhead. On the consumer side, GPS [29] pioneered a proactive data transfer approach assuming pre-generated producer data, designed to overlap the loading time with communication.

Despite these innovations, the prevalent and costly issue of consumers waiting for producer data in practical applications has not been fully explored. Our method, WIC, addresses this by introducing a fine-grained interruption mechanism that effectively overlaps the synchronization latency with ongoing computations when consumers wait for producer data, significantly enhancing the efficiency of GPU communications.

8 Conclusion

This paper proposes WIC, a warp-level, interrupt-based synchronization framework designed to optimize GPU communication by overlapping synchronization delays with computations. We first pinpoint traditional synchronization, characterized by consumers repetitively polling for producer data, as a primary bottleneck in GPU communication. Detailed thread-level analysis within the GPU producer-consumer model reveals that early-started polling squanders significant computational resources and competes with ongoing tasks. To tackle this inefficiency, WIC employs a fine-grained interruption mechanism that preempts warps waiting for data, freeing up resources for active warps. Specifically, WIC integrates three core components: the Interrupter to trigger interruptions to swap other executable warps, the Monitor to track the availability of the producer data, and the Activator to resume the stalled warps when data is ready. Built on unified memory copying, WIC supports over 20 types of cross-platform data communications. Experimental results show that our proposed WIC achieves an average of 13% performance improvements compared to traditional cross-device communications.

Acknowledgments

We sincerely thank our shepherd, Aurojit Panda, and anonymous reviewers for their insightful and constructive comments. This research is supported by XJTLU Research Development Fund (Grant No. RDF-22-02-108), Suzhou Science and Technology Development Planning Programme (Grant No. ZXL2023176), Municipal Key R&D Program of Ningbo (Grant No. 2024Z259), and the Key Technology RD Program of Ningbo (Grant No. 2022Z149).

References

[1] Tyler Allen, Bennett Cooper, and Rong Ge. Fine-grain quantitative analysis of demand paging in unified virtual memory. *ACM Transactions on Architecture and Code Optimization*, 21(1):1–24, 2024.

[2] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. Starpu: a unified platform for task scheduling on heterogeneous multicore architectures. In *Euro-Par Parallel Processing: International Euro-Par Conference, Delft, The Netherlands*, pages 863–874. Springer, 2009.

[3] Seonmyeong Bak, Oscar Hernandez, Mark Gates, Piotr Luszczek, and Vivek Sarkar. Task-graph scheduling extensions for efficient synchronization and communication. In *Proceedings of the Acm International Conference on Supercomputing*, pages 88–101, 2021.

[4] Trinayan Baruah, Yifan Sun, Ali Tolga Dinçer, Saiful A Mojumder, José L Abellán, Yash Ukidave, Ajay Joshi, Norman Rubin, John Kim, and David Kaeli. Griffin: Hardware-software support for efficient page migration in multi-gpu systems. In *IEEE International Symposium on High Performance Computer Architecture*, pages 596–609. IEEE, 2020.

[5] Roberto Belli and Torsten Hoefler. Notified access: Extending remote memory access programming models for producer-consumer synchronization. In *2015 IEEE International Parallel and Distributed Processing Symposium*, pages 871–881. IEEE, 2015.

[6] Jason Burmark. Comb. Technical report, Lawrence Livermore National Laboratory, Livermore, CA, 2018.

[7] Gregory T Byrd and Michael J Flynn. Producer-consumer communication in distributed shared memory multiprocessors. *Proceedings of the IEEE*, 87(3):456–466, 1999.

[8] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *IEEE international symposium on workload characterization*, pages 44–54. Ieee, 2009.

[9] Yuxin Chen, Benjamin Brock, Serban Porumbescu, Aydin Buluç, Katherine Yelick, and John D Owens. Scalable irregular parallelism with gpus: Getting cpus out of the way. In *SC: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–16. IEEE, 2022.

[10] Bengisu Elis, Olga Pearce, David Boehme, Jason Burmark, and Martin Schulz. Non-blocking gpu-cpu notifications to enable more gpu-cpu parallelism. In *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region*, pages 1–11, 2024.

[11] Glenn A Elliott, Bryan C Ward, and James H Anderson. Gpusync: A framework for real-time gpu management. In *IEEE Real-Time Systems Symposium*, pages 33–44. IEEE, 2013.

[12] Daniel Gerzhoy and Donald Yeung. Pipelined cpu-gpu scheduling to reduce main memory accesses. In *Proceedings of the International Symposium on Memory Systems*, pages 1–10, 2021.

[13] Yicheng Gu, Yun Wang, Yunfan Sun, Yuxin Xiang, Yufan Jiang, Xuyan Hu, Zhengwei Qi, and Haibing Guan. gvulkan: Scalable gpu pooling for pixel-grained rendering in ray tracing. In *USENIX Annual Technical Conference*, pages 1151–1165, 2024.

[14] Ralph C Hilzer Jr. Synchronization of the producer/consumer problem using semaphores, monitors, and the ada rendezvous. *ACM SIGOPS Operating Systems Review*, 26(3):31–39, 1992.

[15] Tsung-Wei Huang, Dian-Lun Lin, Chun-Xun Lin, and Yibo Lin. Taskflow: A lightweight parallel and heterogeneous task graph computing system. *IEEE Transactions on Parallel and Distributed Systems*, 33(6):1303–1320, 2021.

[16] Shams M Imam and Vivek Sarkar. Savina-an actor benchmark suite: Enabling empirical evaluation of actor libraries. In *Proceedings of the International Workshop on Programming Based on Actors Agents & Decentralized Control*, pages 67–80, 2014.

[17] Abhinav Jangda, Saeed Maleki, Maryam Mehri Dehnavi, Madan Musuvathi, and Olli Saarikivi. A framework for fine-grained synchronization of dependent gpu kernels. In *2024 IEEE/ACM International Symposium on Code Generation and Optimization*, pages 93–105. IEEE, 2024.

[18] Kevin Jeffay. The real-time producer/consumer paradigm: A paradigm for the construction of efficient, predictable real-time systems. In *Proceedings of the 1993 ACM/SIGAPP symposium on Applied computing: states of the art and practice*, pages 796–804, 1993.

[19] Donghun Jeong, Jihun Park, and Jungrae Kim. Demand memcpy: Overlapping of computation and data transfer for heterogeneous computing. *IEEE Access*, 10:79925–79938, 2022.

[20] Jinwoo Jeong, Seungsu Baek, and Jeongseob Ahn. Fast and efficient model serving using multi-gpus with direct-host-access. In *Proceedings of the Eighteenth European Conference on Computer Systems*, pages 249–265, 2023.

[21] Xu Jiang, Nan Guan, He Du, Weichen Liu, and Wang Yi. On the analysis of parallel real-time tasks with spin locks. *IEEE Transactions on Computers*, 70(2):199–211, 2020.

[22] Changue Jung, Suhwan Kim, Ikjun Yeom, Honguk Woo, and Younghoon Kim. Gpu-ether: Gpu-native packet i/o for gpu applications on commodity ethernet. In *IEEE INFOCOM Conference on Computer Communications*, pages 1–10. IEEE, 2021.

[23] Mahmoud Khairy, Amr G Wassal, and Mohamed Zahran. A survey of architectural approaches for improving gpgpu performance, programmability and heterogeneity. *Journal of Parallel and Distributed Computing*, 127:65–88, 2019.

[24] Konstantinos Koukos, Alberto Ros, Erik Hagersten, and Stefanos Kaxiras. Building heterogeneous unified virtual memories (uvms) without the overhead. *ACM Transactions on Architecture and Code Optimization*, 13(1):1–22, 2016.

[25] Yiwei Li and Mingyu Gao. Hydrogen: Contention-aware hybrid memory for heterogeneous cpu-gpu architectures. In *SC: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15. IEEE, 2024.

[26] Paul T Lin, Michael A Heroux, Richard F Barrett, and Alan B Williams. Assessing a mini-application as a performance proxy for a finite element method engineering application. *Concurrency and Computation: Practice and Experience*, 27(17):5374–5389, 2015.

[27] Daniel Lustig and Margaret Martonosi. Reducing gpu offload latency via fine-grained cpu-gpu synchronization. In *IEEE International Symposium on High Performance Computer Architecture*, pages 354–365. IEEE, 2013.

[28] Syed Nasir Mehmood, Nazleeni Haron, Vaqar Akhtar, and Younus Javed. Implementation and experimentation of producer-consumer synchronization problem. *International Journal of Computer Applications*, 975(8887):32–37, 2011.

[29] Harini Muthukrishnan, Daniel Lustig, David Nellans, and Thomas Wenisch. Gps: A global publish-subscribe model for multi-gpu memory management. In *MICRO: IEEE/ACM International Symposium on Microarchitecture*, pages 46–58, 2021.

[30] Harini Muthukrishnan, Daniel Lustig, Oreste Villa, Thomas Wenisch, and David Nellans. Finepack: Transparently improving the efficiency of fine-grained transfers in multi-gpu systems. In *IEEE International Symposium on High-Performance Computer Architecture*, pages 516–529. IEEE, 2023.

[31] Harini Muthukrishnan, David Nellans, Daniel Lustig, Jeffrey A Fessler, and Thomas F Wenisch. Efficient multi-gpu shared memory via automatic optimization of fine-grained transfers. In *ACM/IEEE Annual International Symposium on Computer Architecture*, pages 139–152. IEEE, 2021.

[32] NVIDIA. Nvidia gpudirect p2p. https://developer.nvidia.com/gpudirect, 2024.

[33] NVIDIA. Nvidia gpudirect rdma. https://docs.nvidia.com/cuda/gpudirect-rdma, 2024.

[34] NVIDIA. Nvidia nvlink. https://docs.nvidia.com/deploy/nvml-api, 2024.

[35] Suchita Pati, Shaizeen Aga, Mahzabeen Islam, Nuwan Jayasena, and Matthew D Sinclair. T3: Transparent tracking & triggering for fine-grained overlap of compute & collectives. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 1146–1164, 2024.

[36] Xiaowei Ren, Daniel Lustig, Evgeny Bolotin, Aamer Jaleel, Oreste Villa, and David Nellans. Hmg: Extending cache coherence protocols across modern hierarchical multi-gpu systems. In *IEEE International Symposium on High Performance Computer Architecture*, pages 582–595. IEEE, 2020.

[37] Rahul Sharma, Michael Bauer, and Alex Aiken. Verification of producer-consumer synchronization in gpu programs. *ACM SIGPLAN Notices*, 50(6):88–98, 2015.

[38] Changmin Shin, Jaeyong Song, Hongsun Jang, Dogeun Kim, Jun Sung, Taehee Kwon, Jae Hyung Ju, Frank Liu, Yeonkyu Choi, and Jinho Lee. Piccolo: Large-scale graph processing with fine-grained in-memory scatter-gather. In *2025 IEEE International Symposium on High Performance Computer Architecture*, pages 641–656. IEEE, 2025.

[39] John A Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, and Wen-mei W Hwu. Parboil: A revised benchmark suite for scientific and commercial throughput computing. *Center for Reliable and High-Performance Computing*, 127(7.2), 2012.

[40] Xian Sun, Peijin Wang, Zhiyuan Yan, Feng Xu, Ruiping Wang, Wenhui Diao, Jin Chen, Jihao Li, Yingchao Feng, Tao Xu, et al. Fair1m: A benchmark dataset for fine-grained object recognition in high-resolution remote sensing imagery. *ISPRS Journal of Photogrammetry and Remote Sensing*, 184:116–130, 2022.

[41] Qihan Wang, Zhen Peng, Bin Ren, Jie Chen, and Robert G Edwards. Memhc: an optimized gpu memory management framework for accelerating many-body correlation. *ACM Transactions on Architecture and Code Optimization*, 19(2):1–26, 2022.

[42] Yueqi Wang, Bingyao Li, Aamer Jaleel, Jun Yang, and Xulong Tang. Grit: Enhancing multi-gpu performance with fine-grained dynamic page placement. In *IEEE International Symposium on High-Performance Computer Architecture*, pages 1080–1094. IEEE, 2024.

[43] Yuke Wang, Boyuan Feng, Zheng Wang, Tong Geng, Kevin Barker, Ang Li, and Yufei Ding. Mgg: Accelerating graph neural networks with fine-grained intra-kernel communication-computation pipelining on multi-gpu platforms. In *USENIX Symposium on Operating Systems Design and Implementation*, pages 779–795, 2023.

[44] Yuke Wang, Boyuan Feng, Zheng Wang, Guyue Huang, and Yufei Ding. TC-GNN: Bridging sparse GNN computation and dense tensor cores on GPUs. In *2023 USENIX Annual Technical Conference*, pages 149–164, Boston, MA, July 2023. USENIX Association.

[45] Qi Yu, Bruce Childers, Libo Huang, Cheng Qian, and Zhiying Wang. Hpe: Hierarchical page eviction policy for unified memory in gpus. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(10):2461–2474, 2019.

[46] Tailing Yuan, Yuliang Liu, Xucheng Ye, Shenglong Zhang, Jianchao Tan, Bin Chen, Chengru Song, and Di Zhang. Accelerating the training of large language models using efficient activation rematerialization and

optimal hybrid parallelism. In *2024 USENIX Annual Technical Conference*, pages 545–561, 2024.

[47] Jiajian Zhang, Fangyu Wu, Hai Jiang, Guangliang Cheng, Genlang Chen, and Qiufeng Wang. Syncmalloc: A synchronized host-device co-management system for gpu dynamic memory allocation across all scales. In *Proceedings of the International Conference on Parallel Processing*, pages 179–188, 2024.

[48] Jiajian Zhang, Fangyu Wu, Hai Jiang, Qiufeng Wang, Genlang Chen, Guangliang Cheng, Eng Gee Lim, and Keqin Li. Alignmalloc: Warp-aware memory rearrangement aligned with uvm prefetching for large-scale gpu dynamic allocations. *IEEE Transactions on Parallel and Distributed Systems*, 36(7):1444–1459, 2025.

[49] Lingqi Zhang, Mohamed Wahib, Haoyu Zhang, and Satoshi Matsuoka. A study of single and multi-device synchronization methods in nvidia gpus. In *IEEE International Parallel and Distributed Processing Symposium*, pages 483–493. IEEE, 2020.

Primus: Unified Training System for Large-Scale Deep Learning Recommendation Models

Jixi Shan[†|], Xiuqi Huang[‡|*], Yang Guo[†], Hongyue Mao[†], Ho-Pang Hsu[†], Hang Cheng[†], Can Wang[†], Jun Song[†], Rui Shi[†*], Xiaofeng Gao[§], Jingwei Xu[†], Shiru Ren[†] Jiaxiao Zheng[†], Hua Huang[†], Lele Yu[†], Peng Xu[†], Guihai Chen[§]

[†]*ByteDance Inc.* [‡]*State Key Lab of CAD&CG, Zhejiang University* [§]*Shanghai Jiao Tong University*

Abstract

The scale of deep learning recommendation models (DLRM) continues to grow, demanding increasingly vast computing and storage resources. In production environments, improving training efficiency and effectiveness has become the primary goal to meet the needs of numerous model training jobs under resource limitations. We introduce Primus, a unified training system that unifies the training resources, data, and paradigms to support high-performance DLRM training at ByteDance. Specifically, ① Primus provides a unified abstraction of resources and interoperates with multiple scheduling systems, achieving a consistent training experience with horizontal and vertical dynamic scaling strategies across resource pools. ② Primus offers a unified three-tier data definition and employs a data task graph generation approach to support data orchestration of multi-source training samples composed of batch and stream data. ③ Primus devises a new hybrid training paradigm for DLRMs that ensures high model timeliness by controlling parameter updates and applying fine-grained prioritization of mixed batch and stream data.

Primus has demonstrated its efficiency and effectiveness in handling large-scale, enterprise-grade DLRM training over five years of deployment at ByteDance. Evaluations show Primus's optimizations of resources, data, and paradigms. Firstly, dynamic scaling reduces training cost by 17.1% at the cluster level and increases CPU utilization from 50% to 80% per job. Secondly, data orchestration accelerates task generation by 23× and achieves higher training throughput. Lastly, after applying the hybrid training paradigm with 4 different DLRMs, advertising revenue increases by 0.4%-2.4%.

1 Introduction

Deep learning has significantly impacted the internet industry, particularly in applications such as search [63], advertising [14], and recommendations [16, 54]. As the user base and data volume continue to grow, the complexity and computational demands of deep learning recommendation models (DLRM) [60] have increased substantially [11, 22].

As shown in Fig. 1, ByteDance's internal DLRM daily training jobs have grown tenfold in five years, with daily training data per model increasing from 20 TB to 160 TB, and daily CPU virtual core usage rising from 1.5 million to 9 million. As of 2025, ByteDance has employed more than 10 million CPU virtual cores, tens of thousands of GPUs, and 7 EB of total training data for DLRM training. The total training volume for a single model can exceed 20 PB of data and involve more than 1 billion neural network parameters.

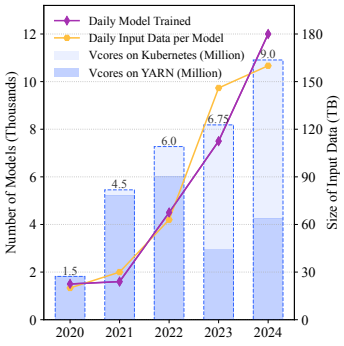


Figure 1: Our DLRM Trends

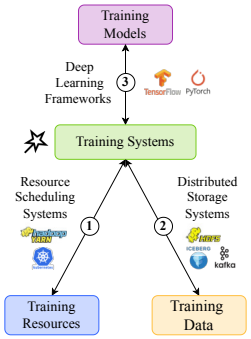


Figure 2: Training Systems

To support such large-scale training, efficient coordination of training resources, data, and models is essential. As illustrated in Fig. 2, developers submit models via deep learning frameworks, after which the training system requests necessary training resources from resource scheduling systems and concurrently loads relevant training data from distributed storage systems. Training large-scale DLRMs requires substantial GPU resources for model parameter updates and CPU resources for data preprocessing [26, 39]. The underlying training system directly affects resource utilization, convergence speed, and model quality [2, 27, 29, 38].

At ByteDance, DLRMs power core services, including Douyin, Xigua, and Toutiao. Meeting the demands of high-

[|]Co-first authors. [*]Co-corresponding authors. [|*]Emails: shanjixi.inf@bytedance.com, huangxiuqi@zju.edu.cn, shirui@bytedance.com

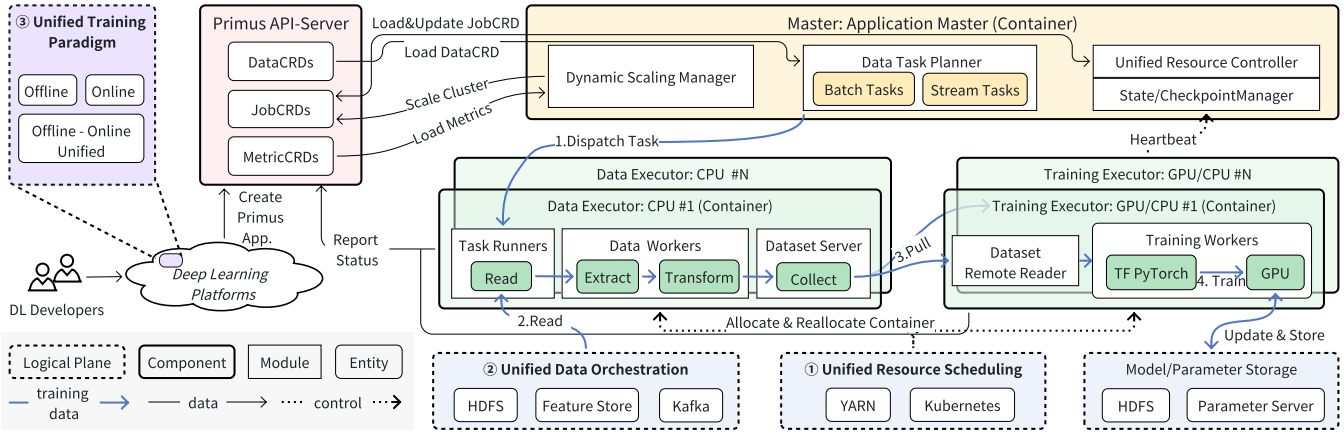


Figure 3: Primus Architecture. ① Unified resource scheduling includes a training master that monitors new JobCRDs and utilizes a unified resource controller to allocate or reallocate executors. ② Unified data orchestration provides a data task planner to generate batch and stream tasks, which are then dispatched to data executors for data loading. Training executors load remote data samples via RPC and construct minibatches for training. ③ Unified training paradigm allows developers to submit DLRM training jobs to the API-Server, whether offline, online, or offline-online, by declarative JobCRDs and DataCRDs definitions.

frequency, large-scale training requires an efficient solution capable of coordinating training resources, orchestrating data processing, and supporting training paradigms. Therefore, establishing a unified training system is imperative [18, 21, 56] to meet ByteDance's needs for enhanced training efficiency and effectiveness.

Challenges & Limitations. Although some existing studies [3,25,32,45,64] have addressed one or more aspects of unified training systems, the limitations of current solutions still fail to fully resolve the following production-scale challenges.

① *Complex and diverse environments, with resources hosted in multiple scheduling systems.* Modern large-scale deployments typically span heterogeneous resource pools managed by different scheduling systems. However, current machine learning scheduling frameworks typically only support a single scheduling system, such as YARN [12, 19, 28] or Kubernetes [27, 31]. Elastic training strategies lack the capability to operate across different scheduling systems, like [13, 26, 57, 65, 68] are restricted to GPU elastic training within Kubernetes. Others lack support for heterogeneous resources, for example, only adapting to GPU adjustment [18, 26, 32, 55, 57] or being limited to CPU scheduling [62]. Even systems [50, 64] consider multiple sources but overlook the horizontal and vertical scaling mechanisms.

② *Large-scale multi-source data, including stream and batch processing data.* DLRM training increasingly involves reading samples from multiple data sources, stored across a variety of systems and formats. The lack of a unified data description complicates orchestration, transformation, and integration. Moreover, the growing complexity of feature engineering pipelines [26, 60] imposes substantial preprocessing demands on CPUs, resulting in a bottleneck in data reading and processing during training [50, 66]. Existing training systems [47, 52, 66] rarely offer native support for orchestrating data from both stream and batch data sources. In particular, integrating emerging storage systems and data formats, like Feature Store [43], require intrusive modifications to training frameworks like TensorFlow [1] and PyTorch [44].

③ *Further online model effectiveness improvement, constrained by catastrophic forgetting.* Maintaining the timeliness of DLRMs is critical for ensuring inference effectiveness [36], given the continual evolution of user behaviors and preferences over time. Timely training updates empower DLRMs to reflect current trends more accurately. While online training approaches [28, 33, 35] directly train with stream data to enhance model timeliness, relying solely on stream data poses the forgetting problem. Specifically, data distribution shifts [59], leading to inadequate utilization of historical sample space. Besides, online learning also does not solve the problem of delayed feedback [30]. Even with the use of both offline and online training modes [10], the lack of unified training support in systems hinders high-frequency updates due to the need for model dumping and loading.

Our Solution. To address the above challenges, we have designed and developed Primus, a unified and production-ready training system tailored for large-scale DLRM workloads. As shown in Fig. 3, Primus is a layered yet centralized architecture that provides unified resource scheduling, unified data orchestration, and a unified training paradigm to integrate with multiple resource scheduling systems and distributed storage systems. Primus coordinates resources, data, and models in a cohesive solution, achieving DLRM training efficiency and effectiveness at the industrial scale.

Contributions. Using Primus, thousands of DLRM training jobs across millions of CPU cores, and tens of thousands of GPUs have been trained efficiently and effectively. We

summarize the following four key contributions:

- We introduce a unified resource abstraction technique, which leverages multi-cluster resources and achieves multi-strategy dynamic scaling. We standardize DLRM training jobs and heterogeneous resources through custom resource definitions (CRDs), providing a unified API across scheduling systems. It enables seamless resource scaling, reduces the overhead of restarting training jobs, and improves cluster resource utilization. (Sec. 3)
- We propose a unified three-tier data definition, comprising dataset, data stream, and data source layers, to manage DLRM's training dataset effectively. With this definition, we further optimize the data task graph generation for data orchestration, enhancing the efficiency of processing batch and stream data from multiple sources, and adapting to the diverse data needs of training jobs. (Sec. 4)
- We present a unified offline-online mixture training paradigm that meets the demands for both robustness and timeliness in DLRMs. It leverages both batch and stream data to continuously improve model performance. Primus optimizes the logic for model parameter updates and enhances the integration of batch and stream data for DLRMs at the system level. (Sec. 5)
- Primus has been actively supporting DLRM training within ByteDance for five years. Through empirical evaluations on production models and datasets, we showcase Primus' capabilities in large-scale resource scheduling and data orchestration. Notably, mixture training with Primus yields up to a 2.43% increase in advertising revenue. (Sec. 6)

Open-Source. The open-source version of Primus is available at https://github.com/bytedance/primus.

2 Primus Overview

The architecture of Primus is presented in Fig. 3. Primus is designed as a centralized, layered architecture that addresses the complexity and inefficiencies in large-scale DLRM training. Primus consists of three logical planes: unified resource scheduling, unified data orchestration, and unified training paradigm. We will briefly introduce the end-to-end training workflow and core system components.

2.1 Workflow for DLRM Training

DLRM training typically consists of two phases: offline training and online training [41]. In the offline phase, training jobs' data executors load batch data from offline storage systems like HDFS and Feature Store. After decoding and transformations, the data is used in deep learning frameworks such as TensorFlow or PyTorch to calculate model weights. Subsequently, online training is employed to improve model performance [35] continuously. In the online phase, training jobs use online data samples from stream sources instead of batch sources to update model weights. These model weights are synchronized to inference servers at regular intervals for online service [48]. Due to the presence of high-dimensional sparse features and large embedding tables, DLRM training systems must sustain high data throughput while maintaining training stability across distributed executors.

Developing, training, and deploying DLRMs involves coordinating multiple complex systems [66]. Figure 3 illustrates the DLRM training process within Primus. In the unified training paradigm plane, deep learning developers submit a DLRM training job, defining the training mode as offline, online, or offline-online. The configuration of the training job is stored in the API-Server as JobCRD and DataCRD. In the unified training resource plane, the Primus master initiates first, then it monitors the status of JobCRDs, DataCRDs, and MetricCRDs in the API-Server in real-time, and requests resources from the scheduling system based on JobCRD to build the training topology. Typically, a training topology comprises several data executors for data loading and training executors for training. In the unified training data plane, the task planner in the master constructs DataCRD into data tasks, which are then dispatched to the executors via remote procedure calls (RPC). The data executors read samples from HDFS, Kafka, and Feature Store through the corresponding task runners. These samples are then converted into minibatches and saved to the data server. The training executors pull the minibatches remotely and process them in the neural network to complete model training. The intermediate states and results of the model training are stored in HDFS or parameter servers (PS).

2.2 Component Design and Functionality

Following the concept of tri-unified training resources, data, and paradigms, Primus implements three core components: Primus APIs, Primus Master, and Primus Executors, which collaborate to optimize training efficiency and effectiveness.

- Primus APIs: DLRM information is submitted to the API-Server by deep learning developers via an API-Client. The API-Server stores and records job and data details, that describe each model's requirements and configuration.
- Primus Master: Acting as the central unit for training, the master is responsible for scheduling computational tasks and managing training data. It requests resources from various scheduling systems through the unified resource controller. During execution, the dynamic scaling manager adjusts resources. The task planner translates the DataCRD into actionable tasks while the state/checkpoint manager records the status of tasks. As a centralized design, Primus adopts frequent checkpoints, resource reservation and stable constraints for fault tolerance and high availability.
- Primus Executors: Executors are responsible for executing training tasks. Data executors handle data management,

Table 1: Primus Component Design and Module Functionalities

Component	Module	Functionality
Primus APIs	API-Server	- Stores JobCRDs, DataCRDs, and MetricCRDs, provides API-Client to read/update/watch CRDs.
Primus Master: Training Master	Data Task Planner	- Compiles DataCRDs into dataset logical plans and optimizes them. - Compiles dataset logical plans into dataset physical plans for batch/stream data task generations.
	Unified Resource Controller	- Unifies different resources belonging to various resource scheduling systems. - Monitors JobCRDs, following the change requirements by request, release, and update resources.
	Dynamic Scaling Manager	- Monitors and loads training metrics through the API-Client, generating scaling strategies. - Selects the optimal horizontal and vertical scaling strategies, changing JobCRDs accordingly.
	State/Checkpoint Manager	- Receives heartbeats from executors and records the status of batch tasks (running, finished) into persistent storage.
Primus Data Executor: CPU	Task Runner	- Fetches tasks from the task planner output queue of the master. - Parses and loads tasks as training samples according to their configurations.
	Data Worker	- Extracts, filters, and converts data into standard training samples, saving them locally.
	Dataset Server	- Collects local outputs from different upstream task runners. - Provides an RPC service endpoint for remote data reading by downstream training executors.
Primus Training Executor: GPU/CPU	Training Worker	- Loads models into deep learning frameworks, reads sample tensors, performs forward and backward propagation, and then applies optimization to get final model parameters.
	Dataset Remote Reader	- Remotely reads data from data executors, and aggregates results from different upstream sources, and constructs training minibatches.

while training executors perform complex model computations. Each data executor includes task runners reading different format data from various storage systems, data workers performing sample conversion, and the dataset server aggregating the converted samples. The training executors retrieve data from the data executors via the dataset remote reader and perform further training computations.

Table. 1 outlines the modules within each component, detailing their associated entities and respective functionalities.

3 Unified Resource Scheduling

To fully utilize existing resources to meet the increasing demands of large-scale DLRM training at ByteDance, Primus integrates resources through the unified resource controller and dynamic scaling manager, as shown in Fig. 4. Specifically, the unified resource controller standardizes the semantics of different resource scheduling systems with JobCRD. Additionally, the dynamic scaling manager enhances the capability for elastic training through collaborative vertical and horizontal scaling. Primus significantly improves training efficiency through multi-cluster resource scheduling and multi-strategy dynamic scaling capabilities.

3.1 Multi-Cluster Resource Scheduling

We design the unified resource controller to adapt to the two most widely used resource scheduling systems, YARN [6], and Kubernetes [9]. This allows developers to define a training job topology independent of the underlying resource scheduling systems using the standard JobCRD, ensuring a unified user experience and reducing development costs.

The unified resource controller converts the JobCRD into resource requests specific to each scheduling system, providing a container manager for each job on YARN or an operator as a resource controller for each cluster on Kubernetes. When running on YARN, an API-Server is started for each job to provide read/write services for CRDs, whereas, on Kubernetes, the cluster's API-Server is used directly. Primus can automatically select resources from multiple YARN and Kubernetes clusters for scheduling based on their idle status. It leverages resources from underutilized clusters while releasing resources when cluster resource utilization is high, thus increasing overall resource efficiency.

Besides, the unified resource controller monitors JobCRD changes and schedules resources accordingly without requiring active user intervention. The dynamic scaling manager modifies JobCRDs determined on optimal scaling strategies using MetricCRDs. The unified resource controller facilitates seamless implementation of horizontal and vertical scaling across different scheduling systems. It addresses the issue of insufficient resources on low-performance executors while enhancing the cost-effectiveness of clusters.

3.2 Multi-Strategy Dynamic Scaling

The dynamic scaling manager combines multiple scaling strategies to support distributed training within large-scale, heterogeneous clusters. Motivated by production experience, key metrics like CPU, memory, network, GPU, and dataset pool size are monitored and aggregated into MetricCRDs. Based on these metrics, the following scaling strategies are automatically selected to increase training throughput and improve resource utilization. For stability, only one scaling

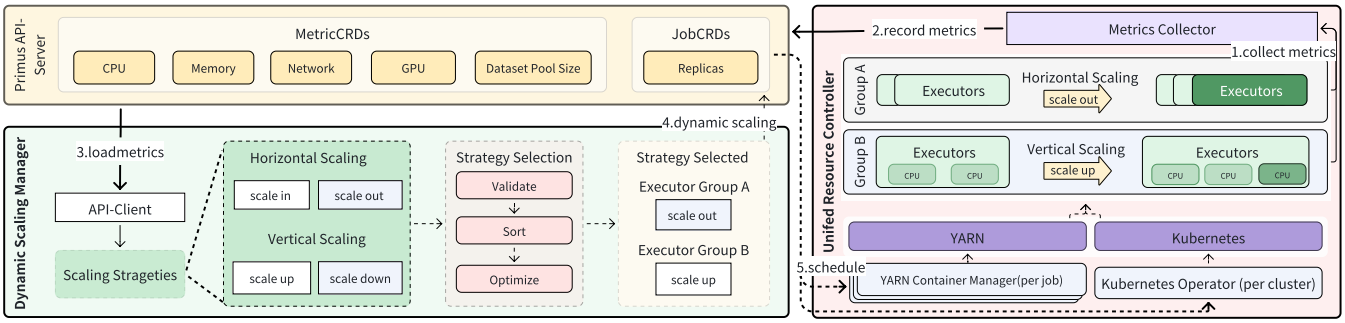


Figure 4: Unified Resource Scheduling

strategy is executed at a time, while it is still a minute-level dynamic scheduling decision. The selection involves validating, sorting, and optimizing steps using previous scaling strategies, resource type, adjustment range, etc.

Dynamic Horizontal Scaling: Primus' horizontal scaling capabilities are well-suited for large-scale cluster scenarios, achieving dynamic parallelism. In CPU and GPU collaborative training scenarios, tuning the ratio of data executors and training executors is allowed.

- Efficient scaling of executors within large clusters: Gradually increasing the number of training executors has been shown to improve AUC (area under the receiver operating characteristic curve) performance [17]. However, scheduling latency is challenging in large-scale scheduling systems with over 1 million cores. Primus reduces scheduling latency by deploying sharded operators for larger clusters and assigning different training jobs to different operators using hashed job names.
- Auto-tuning in collaborative CPU and GPU training: Predetermining the optimal ratio of data to training executors for various jobs is challenging. Primus employs horizontal scaling to dynamically adjust the ratio of data executors to training executors based on a load value L_u calculated using metrics from data executors. Let $\mathbf{u} = [u_b, u_c, u_m]$ represent the total buffer pool, CPU, and memory utilization of data executors, respectively. Users can assign customized weight vector $\mathbf{w} = [w_1, w_2, w_3]$ for each utilization metric based on their training job's requirements. Primus offers an intuitive interface and recommended weights.

$$L_u = \mathbf{w} \cdot \mathbf{u} \tag{1}$$

A load threshold L_θ is predefined based on historical job data and performance metrics. When $L_u < L_\theta$, indicating a higher training executor capacity than data executor capacity, Primus increases the number of data executors. Conversely, when $L_u > L_\theta$, Primus reduces the number of data executors. This automatic adjustment improves training throughput and efficiency without resource waste.

Dynamic Vertical Scaling: Primus uses real-time metrics to dynamically adjust CPU and memory resources on individual executors. Each executor has a metrics collector that regularly gathers data on CPU and memory utilization, I/O throughput, and network bandwidth. These metrics are collected from each executor and made accessible through the API-Server. The dynamic scaling manager calculates future resource requirements based on these metrics and updates the JobCRD accordingly. The unified resource controller then adjusts the resource allocation as needed.

We gather N_R historical resource metrics denoted by R, and utilize function $D(N_R)$ to forecast resource changes in the subsequent period, as expressed in Eq. (2). Recognizing the temporal relevance of these metrics, we assign weights $w_i = \frac{1}{N_R - i + 1}$, emphasizing a stronger influence from more recent metrics. To determine the adjustment range R_x, we analyze resource changes over the past x minutes, maintaining a small interval—typically 10 minutes—to enhance utilization while ensuring stability. That is, $R_x = D(x)$.

$$D(N_R) = \sum_{i \in [1, N_R]} \frac{w(i) \cdot (R_i - R_{i-1})}{N_R}, w(i) = \frac{1}{N_R - i + 1} \tag{2}$$

We use R_{alloc} to represent the currently allocated resources for each executor, while R_{use} denotes the maximum resource used over the past x minutes. To ensure stability, a predefined resource safety margin R_s guarantees resource redundancy, preventing individual executors from encountering out-of-memory (OOM) errors due to overly sensitive adjustments. The sign of R_x determines the direction of resource adjustment. We employ Eq. (3) to compute the updated resource allocation R'_{alloc}. Additionally, a minimum adjustment threshold R_θ is set. Adjustments where $|R'_{alloc} - R_{alloc}| \leq R_\theta$ are not implemented to avoid frequent minor adjustments.

$$R'_{alloc} = R_{used} + R_x + R_s \tag{3}$$

4 Unified Data Orchestration

At ByteDance, data sources are typically organized by business units and stored across various storage systems, such as

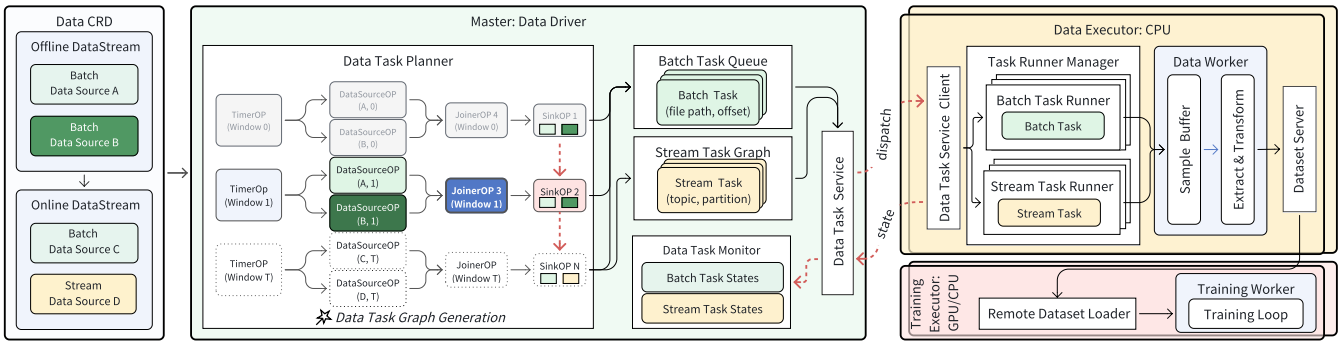


Figure 5: Unified Data Orchestration

HDFS, Kafka, and Feature Store. Additionally, utilizing data samples from multiple data sources can help DLRMs capture more complete information. While existing data orchestration solutions [7, 64] and modern ML frameworks [1, 44] support diverse data types, challenges remain in efficiently managing chronological data across multiple sources, particularly for stream data in online training environments at production scale. In pursuit of efficient data management for DLRMs, we propose an integrated data component that abstracts both data definition and acquisition logic, serving data to training jobs in the desired order through streamlined APIs. This solution optimizes data orchestration, improving both performance and stability in DLRM training.

4.1 Three-Tier Data Definition

To facilitate training jobs for DLRM in our production environments, there are three unique data orchestration requirements for optimizing model effectiveness: 1) scheduling training data among data workers with a sliding window over data generation time, 2) supporting stream data sources to minimize the time required for online training data to enter a training topology, and 3) providing mixed training data from stream and batch data sources for more sophisticated models. Therefore, to fulfill these requirements, Primus stratifies the definition of training data into three tiers: Primus dataset, Primus data stream, and Primus data source:

- Primus Dataset: A Primus dataset represents an unbounded stateful data input for a training job. It manages the lifecycle of one or multiple Primus data streams, which can be served sequentially or simultaneously according to the requirements of training models.
- Primus Data Stream: A Primus data stream is composed of one or multiple Primus data sources. It serves data to its designated data workers by mixing and scheduling the Primus data sources with a sliding window in either hourly or daily granularity.
- Primus Data Source: A Primus data source represents the data of a given period persisted in a particular storage system. This period could be either bounded or unbounded for continuous online training. Additionally, Primus data sources are responsible for converting data to the format specified by training models to eliminate discrepancies in data formats across different storage systems.

4.2 Data Task Graph Generation (DTGG)

Primus embraces data parallelism to harvest the performance improvement from distributed training. Although existing solutions are able to efficiently orchestrate data at scale [52], they lack the support for mixing multiple data sources and serving them chronologically. To address these needs, Primus analyses the assigned Primus data streams and generates data tasks, which are subsequently dispatched to the data executor according to their generation order, where a data task specifies the data from a Primus data source in a time window.

However, data task generation has become a bottleneck due to the increasing data volume since naive implementations may leave training workers pending for data and ultimately lead to underutilized computational resources. Therefore, we introduce data task graph generation (DTGG), a technique designed to improve the performance of data task generation. A DTGG comprises four types of operations (OPs) to serve a Primus data stream, as shown in Fig. 5.

- Timer OPs: The timer OPs are populated for every time window that can be derived from a Primus data stream, and thus, timer OPs are populated indefinitely for unbounded Primus data streams.
- Data Source OPs: The data source OPs are derived from their corresponding Primus data sources and are responsible for generating data tasks based on the time window propagated from their depending timer OPs. For instance, a Feature Store data source OP queries the designated Feature Store table with the propagated time window to locate the relevant data files on storage, which are later built into individual data tasks.
- Joiner OPs: The joiner OPs collect all the data tasks generated from their depending data source OPs. The completion

of a joiner OP marks the successful generation of data tasks for its designated time window.

- Sink OPs: The sink OPs are responsible for pushing the generated data tasks into a buffer for future assignments to data executors. Importantly, sink OPs always depend on their predecessor to ensure the correct chronological order in data task executions.

In addition to parallelization, DTGG can further accelerate data task generation at runtime by caching data tasks for repeated usages and fusing data source OPs derived from the same Primus data sources to amortize the cost of interacting with storage systems.

4.3 Data Component Architecture

To ensure stability, especially for online models, each training job is assigned to a dedicated data component. As shown in Fig. 5, the data driver generates data tasks for the configured Primus data streams and orchestrates their execution among data executors at the center. Meanwhile, the surrounding data executors process the dispatched data tasks and serve the acquired data to their corresponding data workers. Benefiting from the centralized architecture [7], the data component can load balance across data workers via dynamic sharding and provides fault tolerance mechanisms.

4.3.1 Data Driver

In order to well orchestrate data tasks among data executors, the data driver has to generate data tasks efficiently, distribute them to data executors, monitor their status, and provide fault tolerance mechanisms. Therefore, to fulfill these responsibilities, the data driver is composed of five modules.

- Data Task Planner: The data task planner obtains the configured Primus data stream for the training job from API-Server and generates corresponding data tasks using DTGG. The generated tasks from batch data sources and stream data sources are then forwarded to the batch task queue and stream task graph, respectively, for subsequent assignments to data executors.
- Batch Data Task Queue: The batch task queue manages the lifecycles of generated batch tasks, which are tracked through four states: pending, running, succeeded, and failed. Meanwhile, the batch task queue persists batch tasks on remote storage to prevent batch task reconstructions in cases of errors.
- Stream Task Graph: Similarly to the batch task queue, the stream task graph manages the lifecycles of stream tasks. However, since stream tasks run infinitely by binding a data executor to a message queue topic and partition, there are only two states for stream tasks: assigned and unassigned, where the stream data loaded offsets are directly committed to the message queue.
- Data Task Service: The data task service is RPC-based that orchestrates data tasks among data executors by assigning data tasks to them and tracking their status. Importantly, to ensure alignment between batch and stream data serving time windows, the data task service may throttle task assignments when necessary.
- Data Task Monitor: The data task monitor tracks the status of assigned data tasks as reported by data executors. It also periodically or on-demand creates checkpoints for fault tolerance to accurately align with model checkpoints.

4.3.2 Data Executor

Data executors are initiated by the data driver to serve orchestrated data. Through the data task service, data executors obtain data tasks and report both their own status and that of the assigned tasks. Upon data task assignments, data executors launch a new data task runner to execute the assigned data task and push the pre-processed data into an in-memory sample buffer, which is finally served to its hosting training worker in a standardized format such as Apache Arrow [4]. As shown in Fig. 5, multiple data task runners are employed in parallel on each data executor to fully utilize the allocated computational resources.

- Batch Task Runner: For every assigned batch task, the data executor launches a new batch task runner, which executes the batch task and terminates upon completion. Meanwhile, the data executor periodically reports the status of the assigned batch tasks to the data driver.
- Stream Task Runner: Streaming task runners are launched for assigned stream tasks. Nevertheless, pursuing better stability in loading stream data sources, stream task runners instead keep running without termination.

However, the static relationships between data executors and the underlying topic and partition of message queues make data component vulnerable to skewed data, which could significantly impact the effectiveness of online models due to the delays in learning stream data. Therefore, data component supports a shuffle mechanism. By allowing data executors to obtain data from each other, data component can well handle skewed data with overheads on memory footprint and network bandwidth in exchange for better overall stability in preparing samples from stream data sources. To support various training scenarios, Primus provides two different stream data loading modes of data executors as listed in Tab. 2:

5 Unified Training Paradigm

To effectively combine online stream data with offline batch data, leveraging the strengths of both online and offline training, we propose a unified training paradigm. Utilizing unified

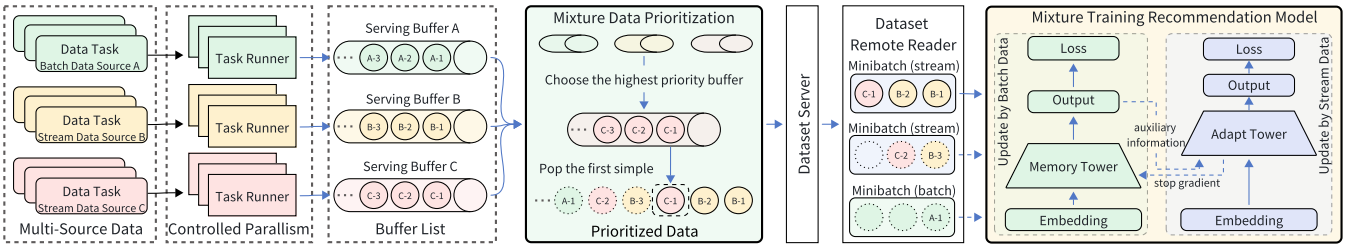


Figure 6: Unified Training Paradigm

Table 2: Stream Data Loading Modes of Data Executors

Mode	Description
Forward	Fetch and pre-process data only from assigned topics and partitions.
Rebalance	Re-deliver fetched data to others for pre-processing.

resources and data, we delegate the complex resource scheduling and data orchestration to Primus. This allows deep learning developers to focus primarily on model structure updating, thereby facilitating model design. As shown in Fig. 6, we design a mixture training recommendation model (MTRM) that controls the learning sequence of batch and stream data through different computational graphs for training. Additionally, we introduce a mixture data prioritization mechanism, which optimizes the logic for multi-source data retrieval and provides row-level priority control. MTRM is one of the training paradigms supported by Primus, which also provides pure batch, pure stream, and sequential batch-then-stream training.

5.1 Mixture Training Recommendation Model

Through statistical analysis and hypothesis testing, we identify two critical observations: 1) The non-MTRM suffers from catastrophic forgetting, where the model prioritizes memorizing the "current distribution". This leads to two primary issues: first, it fails to retain samples over time, underutilizing historical knowledge; and second, it becomes more susceptible to bias and delays in feedback, which results in instability. 2) Different batch data sources exhibit varying output delays due to output strategies, such as label splicing goals (e.g., next-day retention, 7-day retention, etc.). Therefore, MTRM requires controlling the sequence of multi-source data and aligning batch data with stream data.

We propose the universal recommendation model framework MTRM for mixture training, as shown in Fig. 6 (right). The model comprises two key components: the memory tower and the adapt tower. The memory tower focuses on long-term parameter storage, processing historical data from batch data sources to prevent catastrophic forgetting. The adapt tower adapts to recent data from stream data sources, ensuring the model remains updated with the latest trends. Typically, the memory tower has a smaller network structure, minimizing computational overhead. MTRM offers flexibility, allowing the memory and adapt towers to be extended to various neural network structures, supporting model innovation.

Algorithm 1: MTRM Training Process

Input: Continuous minibatches $\mathbf{X}$ from D_{pri}
Output: Updated MTRM with θ_M^{new} and θ_A^{new}

Initialize or load model parameters: memory tower θ_M, adapt tower θ_A;
Set learning rates α and loss functions $\mathcal{L}_M$, $\mathcal{L}_A$;
while *Not all minibatches* $\mathbf{X}$ *are processed* **do**
  if $\mathbf{X}$ *is* $\mathbf{X}_{batch}$ **then**
    $\mathbf{e}_{\text{batch}} \leftarrow \text{Embedding}(\mathbf{X}_{\text{batch}})$;
    $\mathbf{h}_M \leftarrow \text{MemoryTowerForward}(\mathbf{e}_{\text{batch}}, \theta_M)$;
    $\mathcal{L}_M \leftarrow \text{ComputeLoss}(\mathbf{X}_{\text{batch}}, \mathbf{h}_M, \theta_M)$;
    $\theta_M^{\text{new}} \leftarrow \theta_M - \alpha \nabla_\theta \mathcal{L}_M$;
  else if $\mathbf{X}$ *is* $\mathbf{X}_{stream}$ **then**
    $\mathbf{e}_{\text{stream}} \leftarrow \text{Embedding}(\mathbf{X}_{\text{stream}})$;
    $\mathbf{h}_{\text{aux}} \leftarrow \text{MemoryTowerForward}(\mathbf{e}_{\text{stream}}, \theta_M)$;
    $\mathbf{h}_A \leftarrow \text{AdaptTowerForward}(\mathbf{e}_{\text{stream}} \oplus \mathbf{h}_{\text{aux}}, \theta_A)$;
    $\mathcal{L}_A \leftarrow \text{ComputeLoss}(\mathbf{X}_{\text{stream}}, \mathbf{h}_A, \theta_A)$;
    $\theta_A^{\text{new}} \leftarrow \theta_A - \alpha \nabla_\theta \mathcal{L}_A$;
return θ_M^{new} or θ_A^{new}

During training, the dataset remote reader maintains two separate queues for batch and stream samples pushed by the dataset service. The accumulation speed is dynamically determined by the mixture data prioritization mechanism in Sec. 5.2. Whenever samples in either queue reach the size of a minibatch, the training worker executes a training loop as outlined in Algorithm 1. The updated model parameters are synchronized periodically between training workers. For convenience, we denote a minibatch $\mathbf{X}$ of batch or stream data samples as $\mathbf{X}_{\text{batch}}$ or $\mathbf{X}_{\text{stream}}$, respectively.

Each data sample from batch or stream sources is utilized only once during training to avoid overfitting to the latest data and to leverage long-term trends. We design the neural network parameter update control logic as follows:

- Embedding Calculation: The input minibatch $\mathbf{X}$ passes

through the embedding layers, generating the corresponding feature embeddings $\mathbf{e}_{\text{batch}}$ or $\mathbf{e}_{\text{stream}}$.

- Forward Propagation with Memory or Adapt Towers: 1) The memory tower processes $\mathbf{e}_{\text{batch}}$ or $\mathbf{e}_{\text{stream}}$, generating $\mathbf{h}_M = f_{\theta_M}(\mathbf{e}_{\text{batch}})$ or $\mathbf{h}_{\text{aux}} = f_{\theta_M}(\mathbf{e}_{\text{stream}})$, with θ_M as its parameters. 2) The adapt tower combines $\mathbf{e}_{\text{stream}}$ with $\mathbf{h}_{\text{aux}}$ to generate $\mathbf{h}_A = f_{\theta_A}(\mathbf{e}_{\text{batch}} \oplus \mathbf{h}_{\text{aux}})$, using θ_A as its parameters.
- Loss Calculation and Parameter Updates: During backpropagation, the memory (or adapt) tower's output $\mathbf{h}_M$ (or $\mathbf{h}_A$) are used to compute the loss $\mathcal{L}_M$ (or $\mathcal{L}_A$). Its gradients iteratively update the corresponding tower's parameters.

Since batch data samples are processed several hours or days after the corresponding stream data, the memory tower has not yet encountered samples from the most recent time period. Besides, $\mathbf{h}_{\text{aux}}$ is not used in the backpropagation of the memory tower. This temporal and tower parameter separation minimizes the risk of information leakage when $\mathbf{h}_{\text{aux}}$ is used as auxiliary information for the adapt tower.

By leveraging this parameter update control logic, we aim to balance overfitting concerns with the efficient handling of delayed feedback, thus enhancing the robustness and effectiveness of recommendation models.

5.2 Mixture Data Prioritization

Essentially, MTRMs are trained on data generated at different times with constant offsets; therefore, the ability to effectively orchestrate and integrate data from multiple sources is essential. By creating the prioritized data constructed from stream and batch data sources configured with different offsets, the required mixture data is served to model training. This design is inspired by empirical observations of performance degradation caused by stream data lag in production environments.

Due to the nature of production services, MTRMs are occasionally struck by sudden surges in stream data, particularly during peak hours or hot events. This can cause delays in loading stream data and subsequently affect model effectiveness. In such cases, merely scaling out data workers is not ideal, as computing resources are limited and surge durations are unpredictable. Therefore, an adaptive and resource-aware prioritization strategy is essential.

To retain MTRMs' effectiveness under high load conditions, Primus enables data workers to prioritize data from different data sources based on their importance and current load. Algorithm 2 shows the detailed process of how data are prioritized and served to the training workers. Data tasks from multi-sources are executed by data workers with controlled parallelism and pushed into their respective serving buffers. When the dataset remote reader requests data, the dataset server fetches data from the serving buffer with the highest priority calculated by Eq. (4):

$$\arg \max_{i \in 1,2,\ldots,n} p_i = w_i \cdot \frac{1}{1+e^{-q_i}} \tag{4}$$

Algorithm 2: Prioritized Data from Serving Buffers

Input: Serving buffers $D = \{D_1, D_2, \ldots, D_n\}$, buffer weight $W = \{w_1, w_2, \ldots, w_n\}$, and buffer queued data length $Q = \{q_1, q_2, \ldots, q_n\}$
Output: Continuous prioritized data queue D_{pri}

```
while Serving buffers D ≠ ∅ do
    maxIndex ← −1, maxPri ← −∞;
    for i ← 0 to bufferList.size − 1 do
        p_i ← w_i · sigmoid(q_i) ;
        if p_i > maxPri then
            maxIndex ← i, maxPri ← p_i ;
    sample ← D_maxIndex.pop();
    D_pri.append(sample);
return D_pri;
```

where n is the number of severing buffer, p_i is the priority for the i-th buffer, w_i is the weight assigned to the i-th buffer indicating its importance, and q_i is the length of the queued data in the i-th buffer. The sigmoid function ensures that buffer length contributes non-linearly to the priority score, allowing Primus to flexibly balance data timeliness, buffer backpressure, and source importance.

In production, the weights of stream data sources are much higher than their batch counterparts due to their impacts on model effectiveness. This flexible mixed data prioritization can dynamically adjust the processing order of different data sources to reduce data latency during high load periods.

6 Evalutions

We validate Primus' capabilities within a large-scale distributed environment. The key highlights are as follows:

- Horizontal scaling saves over 17.1% of resources while vertical scaling improves utilization from 50% to 80%.
- Generating around 4 million data tasks speeds up 23× in 1 thread and can be completed in 42s with 4 threads.
- Model effectiveness in advertising is significantly improved by 0.03%-0.07% in AUC and 0.4%-2.4% in revenue.

As a large-scale, enterprise-grade DLRM training system within ByteDance, we provide detailed information about workloads currently undertaken by Primus in Tab. 3.

6.1 Unified Resource Evaluation

We assess Primus' horizontal and vertical dynamic scaling capabilities in different training scenarios. The evaluation measures training throughput (minibatches/s), CPU/memory utilization, and model effectiveness (AUC).

Dynamic Horizontal Scaling: To evaluate the performance of large-scale dynamic parallelism, a baseline setup running

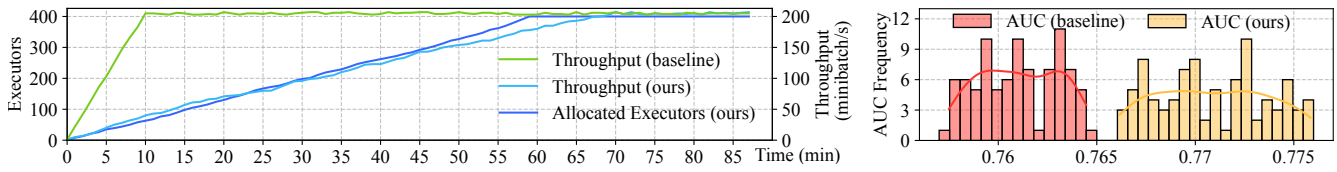


Figure 7: Dynamic Horizontal Scaling

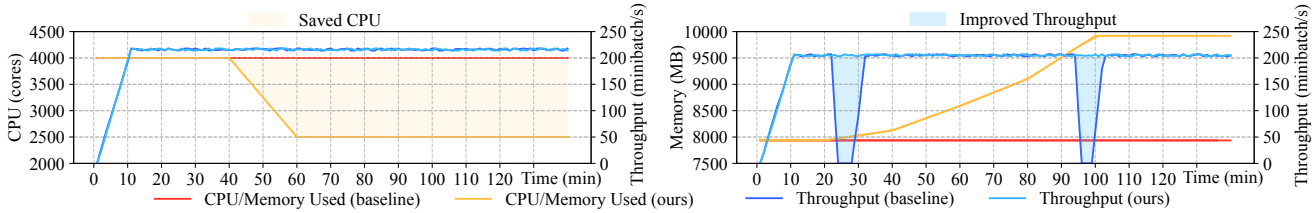


Figure 8: Dynamic Vertical Scaling Improves Resource Efficiency

a fixed number of 400 data executors is compared with our setup, where data executors are gradually scaled from 0 to 400 every 10 seconds. As shown in Fig. 7, Primus's training throughput is gradually increased to 210 minibatch/s, similar to the baseline setup's throughput of 205 minibatch/s, while AUC is approximately 0.4% higher than the baseline, improving training effectiveness with the scaling strategy.

To evaluate the impact of tuning the ratio of training executors and data executors, a baseline setup using 900 8-core data executors and 8 training executors throughout the training process is compared with our setup initially using 450 8-core data executors and also 8 training executors. As shown in Tab. 4, the total training time and the training AUC of our setup are comparable to the baseline. However, our setup eventually uses 690 data executors, resulting in a 23.33% saving in CPU cores compared to the baseline. The tuning improves resource utilization without negatively affecting training performance or speed. When the feature is enabled for a production cluster, training throughput per core (ROI) gradually increases from 30.26 to 35.44, representing a 17.1% improvement.

Dynamic Vertical Scaling: We assess Primus' vertical scaling with two typical training scenarios. The first scenario simulates an excessive CPU resource allocation to workers, causing resource wastage. Both jobs are started with 400 10-core data executors, with Primus' vertical scaling expected to adjust the CPU core allocation of the executors. As shown in Fig. 8 (left), the baseline utilizes a fixed number of CPU cores, while ours scales down by 1600 CPU cores, representing an adjustment per executor from 10 to 6 cores. The effect is an increase of CPU utilization from 50% to 80% and no impact on both training throughput and AUC.

The other scenario simulates insufficient memory allocation for PS executors, leading to OOM errors, which cause failovers and disrupt training progress. Both the baseline and ours are initiated with 496 PS executors with 16 GB of memory each. Primus' vertical scaling is expected to dynamically adjust the memory of PS executors for our task to avoid OOM errors. As shown in Fig. 8 (right), the baseline frequently encounters PS executor OOM errors, resulting in periods with low training throughput. Ours adjusts the allocated memory to a total of 9920 GB from 7936 GB compared to the baseline, which prevents the OOM issues. The effect is an increase in throughput from 275 minibatch/s to 496 minibatch/s, and an AUC increase from 0.62 to 0.78.

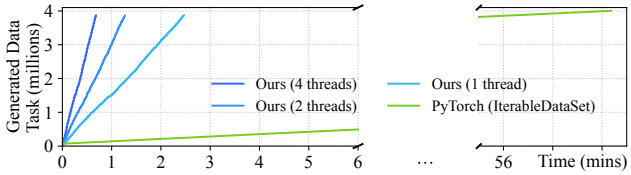


Figure 9: Data Task Generation Speed

6.2 Unified Data Evaluation

We evaluate the performance of Primus data component, which analyses the designated data sources and subsequently serves the data to training workers. Therefore, the overall performance is a collective result of the two major responsibilities: data task generation and data loading.

Data Task Generation Evaluation: We evaluate the efficiency of Primus's data component in generating data tasks from large-scale production data. The experiment is performed on a single data driver with 8 CPU cores and 25 GB memory to eliminate the impact of memory pressure. The input workload consists of 20-day production data from two Feature Store tables, comprising over 56 million underlying files and yielding approximately 4 million generated data tasks. As shown in Fig. 9, DTGG achieves significantly faster task generation compared to a naive single-threaded baseline implemented with PyTorch. By fusing multiple scan requests

Table 3: Characteristics of DLRM Training Workloads on Primus

Categories	Types	Details
Resource	General Data Workers	800 CPU data workers, each with 5 cores and 50 GB memory
	General Training Workers	4 GPU training workers, each with 8 GPUs
Data	Data Tasks	Ranging from 0 to 80 million tasks ($\sim$ 180–540 days' data)
	Data Sources	Spanning 2 to 18 sources, each with 10–20 PB data
	Training Throughput	Ranging from 200,000 to 600,000 samples per second
Model	Model Size	Varying from 500 GB to 1 TB
	Sample Features	The 75th percentile of the number of features is $\sim$3200

Table 4: Dynamic Horizontal Scaling Comparision

Method	CPU Used (cores)	AUC	Training Time (days)	Cluster ROI
Baseline	7200	0.9385	1.86	30.26
Ours	5520	0.9382	1.90	35.44+17.12%

Table 5: Model and Resource Settings

No.	Model Basis*	Model Size (GB)	Executors Number	Executors CPU (cores)	Executors Memory (GB)
1	DSSM [24], LHUC [49]	600	60	13	45
2	FM [46], Transformer [51]	1200	40	6	30
3	FM, DIN [67], Transformer	800	50	12	35
4	EMSNet [20], LHUC	3700	50	8	30

*Production DLRMs absorb and integrate the various model structures listed.

to the underlying HDFS storage, the 1-thread DTGG reduces the generation time from 58 minutes to 149 seconds (23×) and further improves to 42 seconds with the 4-thread DTGG. These results demonstrate DTGG's efficiency in handling multiple data sources and time windows.

Data Loading Evaluation: Compared to batch data sources, stream data sources are widely adapted by online training jobs which have higher performance requirements, especially in production environments. Figure 10 shows the performance comparison of serving stream data sources between Primus and Flink [5] (a widely used online model training framework). This experiment is performed on 40 executors on machines equipped with 2 CPU cores and 4 GB memory each, where 10 of them are configured extremely slower than others to simulate anomalies such as hardware degradation in production environments. Both Primus and Frink enable shuffle mechanisms to balance load across executors dynamically. Primus data component achieves 3.97 GB/s, which is 1.25× to 3.17 GB/s with Flink. Since shuffling data with Flink incorporates two layers of task managers for generalized usages, Flink is more fragile to defective machines, as the back pressure induced by the latter layer could lead to resource underutilization in the former layer. On the contrary, Primus data executors are equal and thus will not impede the performance of others. Therefore, though the resource utilization of slow executors is similar, Primus's shuffle mechanism design allows for better overall utilization across all executors. This leads to higher effective throughput under unstable environments or skewed data.

6.3 Unified Training Evaluation

To evaluate the performance of the unified training paradigm, we use production data to conduct the effectiveness of MTRM and mixture data prioritization.

MTRM Evaluation: We select four representative production models for Conversion Rate (CVR) and Click-Through Rate (CTR) estimation models to thoroughly assess the effectiveness of MTRM in various scenarios. Each model's structural information and resource specifications are detailed in Tab. 5. Without MTRM (w/o), these baseline models can be viewed as single-tower-like models that process both batch and stream data without separation. When utilizing MTRM (w/), these four models all employ a memory tower to learn from 24-hour delayed batch data, while an adapt tower remains the same structure as baselines but only updated by stream data. Both the models with and without MTRM are pre-trained on one year of historical data. Both the models with and without MTRM are pre-trained on one year of historical data. Since both models utilize the same stream data, they share relatively comparable AUC values. AUC quantifies the model's ability to rank positive instances (e.g., clicks or conversions) ahead of negative ones, and serves as a reliable proxy for model quality prior to online deployment. As shown in Tab. 6, the implementation of MTRM results in increased AUC uniformity across the four production models, with an average improvement of 0.03% to 0.07%. Additionally, we conduct A/B testing, with the number of samples diverted by each model presented in Tab. 6. After applying MTRM, the advertising revenue increases by 1.045%, 0.806%, 2.438%, and 0.397%, respectively. These results demonstrate that using MTRM can effectively enhance the performance and revenue of recommendation models.

Mixture Data Prioritization: For evaluating serving mixture data under traffic pressure, we conduct an experiment with 60

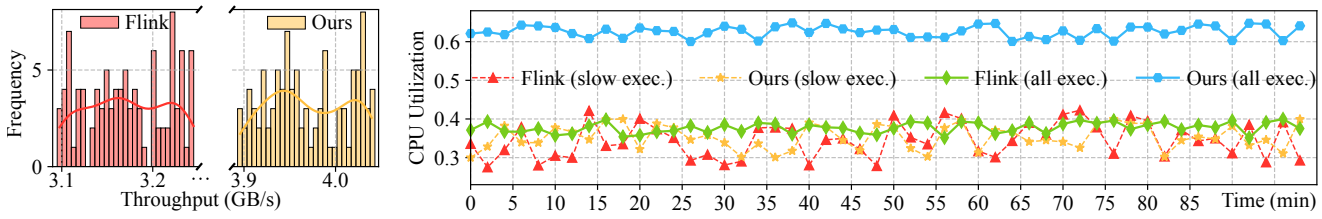


Figure 10: Primus Online Training VS. Flink Online Training with 25% Slow Executors

Table 6: Comparison of AUC and Revenue Performance for Various Models without (w/o) and with (w/) MTRM

Model		Evaluation (AUC) Day 1	Day 2	Day 3	Day 4	Day 5	Day 6	Day 7	Avg.	A/B Testing Samples	Revenue
1	w/o	0.92411	0.91992	0.92538	0.92737	0.92187	0.92554	0.92642	0.92437	2,252,037	-
	w/	0.92439+0.03%	0.92018+0.03%	0.92559+0.02%	0.92758+0.02%	0.92212+0.03%	0.92582+0.03%	0.92660+0.01%	0.92461+0.03%	2,252,459	+1.045%
2	w/o	0.87125	0.86748	0.87103	0.86939	0.86885	0.86855	0.86887	0.86935	21,643,587	-
	w/	0.87173+0.06%	0.86804+0.06%	0.87156+0.06%	0.86987+0.06%	0.86930+0.05%	0.86907+0.06%	0.86945+0.07%	0.86986+0.06%	21,643,223	+0.806%
3	w/o	0.95333	0.95083	0.95175	0.95098	0.95070	0.95218	0.95141	0.95160	21,141,647	-
	w/	0.95380+0.05%	0.95131+0.05%	0.95220+0.05%	0.95143+0.05%	0.95122+0.05%	0.95269+0.05%	0.95188+0.05%	0.95208+0.05%	21,148,298	+2.438%
4	w/o	0.83299	0.83337	0.83471	0.83443	0.83380	0.83460	0.83608	0.83428	254,869,721	-
	w/	0.83352+0.06%	0.83395+0.07%	0.83527+0.07%	0.83500+0.07%	0.83438+0.07%	0.83513+0.06%	0.83666+0.07%	0.83484+0.07%	254,864,629	+0.397%

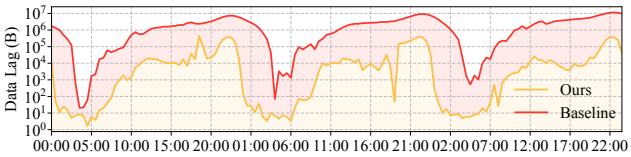


Figure 11: Data Prioritization Reduces Loading Lag

data executors equipped with 11 CPU cores and 40 GB memory loading 1-day data of a production Kafka topic of 1,024 partitions with average traffic of 600 MB/s and its corresponding historical copy persisted on Feature Store. As shown in Fig. 11, the baseline result without mixture data prioritization suffers badly from data loading lags during peak hours, which eventually renders negative impacts on the model effectiveness. On the contrary, with mixture data prioritization, the loading lags are hugely reduced, and thus, the model effectiveness is better protected, which verifies that Primus is able to control the proportion of loaded data from different data sources in a fine-grained manner, especially during peak hours with limited computation resources.

7 Lessons

In this section, we share our experiences and lessons learned during the design and implementation of Primus.

Road Towards Unified Resource Scheduling: In the early stages of Primus, Kubernetes lacked sufficient pod scheduling throughput and did not support gang scheduling semantics. Consequently, training resources were primarily managed by YARN, through a hybrid deployment of YARN NodeManager and Kubelet. As Kubernetes' scheduling capabilities improved, along with its support for gang scheduling semantics, a greater portion of training jobs was migrated to Kubernetes. During this migration, ensuring consistency in user experience became critical, as system changes could not impact users. To address this, Primus services as an intermediary layer, insulating users from the differences between scheduling systems by the unified resource scheduling design. Over time, Primus evolves to support more comprehensive dynamic horizontal and vertical scaling capabilities.

Optimizing Data Access for Various Formats: The vast quantities of sample data are stored in diverse formats (e.g., TFRecord, Protobuff, and 4mz) and compressed with multiple algorithms (e.g., Snappy, ZSTD, and GZIP). Initially, a separate Java process (using InputFormat) was used to read files and pipe them to training executors which faced overhead in cross-process communication and data format conversions. To address this, Primus re-engineers the data access layer using Arrow for cross-language efficiency and NativeClient for optimized stream data handling, reducing CPU usage by 40%. Additionally, Primus master employs ClassLoader and other isolation techniques to allow seamless integration of diverse data lakes like Iceberg and Hudi, balancing data management complexity with high retrieval efficiency.

Necessity of Mixture Data Prioritization: Initially, Primus employed two distinct groups of executors to process batch and stream data separately. However, we observed that batch data processing occurred more rapidly, resulting in resource wastage. Then, Primus was updated to handle both batch and stream data within the same executors. While this approach

increased resource efficiency, we found that the large volume and fast arrival rate of batch data often consumed excessive resources, delaying the processing of stream data, particularly during peak hours. To address this, we design the continuous prioritized data queue that ensures stream data is processed first. It helps prevent MTRM performance degradation by ensuring timely learning of stream data.

8 Related Work

Deep Learning Scheduler. Deep learning training optimization with various resource schedulers has received attention in recent years in recent years [34, 42]. Studies indicate that YARN and Kubernetes are the mainstream training frameworks at present [23]. Current optimization strategies like dynamic resource allocation [15] , face portability issues when applied across heterogeneous environments. Systems like Oobleck [26] and DynamoML [13] concentrate on elastic GPU training but remain tightly coupled to Kubernetes, limiting their applicability to more diverse production infrastructures.. GoldMiner [64] proposes a novel resource allocation paradigm on Kubernetes, separating CPU and GPU and allocating both resources automatically to improve the efficiency of the cluster. However, GoldMiner is only adapted to Kubernetes, which cannot be extended to other schedulers. While ElasticFlow [18] elegantly supports dynamic GPU reallocation, it is not designed to accommodate heterogeneous executor configurations. Moreover, while horizontal pod autoscaling leverages workload predictions to reduce unnecessary resource allocation, discussions on machine learning-based vertical pod autoscaling remain relatively scarce [61]. Primus enables dynamic scaling and precise resource control for both CPU and GPU workers, supporting collaboration with multiple scheduling systems.

Deep Learning Data Loading. Along with the increase of data amount, data reading and preprocessing become complicated and critical [60]. tf.data [40] proposes a machine learning preprocessing framework, which mainly concentrates on throughput optimization during sample reading. However, massive metadata training is not considered, and data fragmentation is not supported in elastic training scenarios. tf.data service [7] maximizes GPU utilization to save training time and implements sample concurrent preprocessing with CPU data preprocessing. However, there is an additional loss in RPC data transmission. Meanwhile, current training systems such as Horovod [47], DLRover [52], and others [58] usually only have a single data source, which fails to have built-in support for multiple data sources with various storage systems and formats. Primus seamlessly integrates various data sources, encompassing both stream and batch data stored in HDFS, Kafka, and Feature Store.

DLRM Online Training. DLRMs are widely employed by internet companies for various tasks such as ad CTR prediction [41]. Due to the rapid increase in data scale, the training of DLRM models in distributed memory and computing systems faces challenges [53]. cDLRM [8] implements training large DLRM model with single GPU and pure data parallel training by unique caching strategy. Monolith [35] and Persia [33] focus primarily on innovations in model structures, such as sparse-dense parameter separation and large model efficient distributed updates. They are tightly coupled with specific model structures and directly train online DLRM with only stream data in homogeneous GPU-based training environments. General stream processing systems, such as Flink [5] and Ray [37], require a lot of user code modifications to use batch data. Primus provides strong model-agnostic support for DLRM training, providing features such as error handling, dynamic scaling, and data orchestration.

9 Conclusion

We present Primus, a tri-unified training framework for DLRM that encompasses three primary innovations. First, Primus provides a unified resource scheduling abstraction across multiple resource scheduling systems. To the best of our knowledge, Primus is the first to offer a multi-strategy dynamic scaling mechanism utilizing a standardized API server that interoperates seamlessly with both YARN and Kubernetes clusters. Second, Primus introduces a three-tier data definition that facilitates efficient offline and online training within a single framework. It implements DTGG to enable parallel task generation for orchestrating training data at scale. Third, Primus features an offline-online mixture training paradigm, allowing for continuous ingestion of historical offline batch data during online model training, resulting in a significant increase in advertising revenue. Since 2019, Primus has been utilized for all ByteDance production DLRM training, leveraging a resource scale exceeding 10 million CPU cores and tens of thousands of GPUs. We continue to enhance Primus to achieve better efficiency and effectiveness in DLRM training.

Acknowledgments

We would like to thank all those who contributed to the design and development of Primus, including Han Qian, Dongyan Zhou, Hanqing Zhao, Irshad Kandy, Kai Xie, Chi Zhang, Di Wu, and Hoang Le. We are also grateful to our business partners across the search, advertising, and recommendation teams for their valuable support and collaboration. This work was supported by the National Key R&D Program of China (2024YFF0617700), the National Natural Science Foundation of China (U23A20309, 62272302, 62372296), "Pioneer" and "Leading Goose" R&D Program of Zhejiang (2024C01167), and the ByteDance Research Project (CT20250218122839). Xiuqi Huang completed the majority of her contributions during her Ph.D. studies at Shanghai Jiao Tong University and her research internship at ByteDance.

References

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Gregory S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian J. Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Józefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Gordon Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul A. Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda B. Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *CoRR*, abs/1603.04467, 2016.

[2] Bilge Acun, Matthew Murphy, Xiaodong Wang, Jade Nie, Carole-Jean Wu, and Kim M. Hazelwood. Understanding training efficiency of deep learning recommendation models at scale. In *IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 802–814. IEEE, 2021.

[3] Saurabh Agarwal, Amar Phanishayee, and Shivaram Venkataraman. Blox: A modular toolkit for deep learning schedulers. In *European Conference on Computer Systems (EuroSys)*, pages 1093–1109. ACM, 2024.

[4] Apache Software Foundation. Apache Arrow, 2024. A universal columnar format and multi-language toolbox for fast data interchange and in-memory analytics. https://arrow.apache.org/.

[5] Apache Software Foundation. Apache Flink, 2024. A framework and distributed processing engine for stateful computations over unbounded and bounded data streams. https://flink.apache.org/.

[6] Apache Software Foundation. Apache Hadoop YARN, 2024. The fundamental idea of YARN is to split up the functionalities of resource management and job scheduling/monitoring into separate daemons. https://hadoop.apache.org/docs/stable/hadoop-yarn/hadoop-yarn-site/YARN.html.

[7] Andrew Audibert, Yang Chen, Dan Graur, Ana Klimovic, Jirí Simsa, and Chandramohan A. Thekkath. tf.data service: A case for disaggregating ML input data processing. In *ACM Symposium on Cloud Computing (SoCC)*, pages 358–375. ACM, 2023.

[8] Keshav Balasubramanian, Abdulla Alshabanah, Joshua D Choe, and Murali Annavaram. cDLRM: Look ahead caching for scalable training of recommendation models. In *ACM Conference on Recommender System (RecSys)*, pages 263–272. ACM, 2021.

[9] Brendan Burns, Brian Grant, David Oppenheimer, Eric A. Brewer, and John Wilkes. Borg, Omega, and Kubernetes. *Communications of the ACM (CACM)*, 59(5):50–57, 2016.

[10] Zhangming Chan, Yu Zhang, Shuguang Han, Yong Bai, Xiang-Rong Sheng, Siyuan Lou, Jiacen Hu, Baolin Liu, Yuning Jiang, Jian Xu, and Bo Zheng. Capturing conversion rate fluctuation during sales promotions: A novel historical data reuse approach. In *ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD)*, pages 3774–3784. ACM, 2023.

[11] Shubham Chaudhary, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, and Srinidhi Viswanatha. Balancing efficiency and fairness in heterogeneous GPU clusters for deep learning. In *European Conference on Computer Systems (EuroSys)*, pages 1:1–1:16. ACM, 2020.

[12] Kai-Hsun Chen, Huan-Ping Su, Wei-Chiu Chuang, Hung-Chang Hsiao, Wangda Tan, Zhankun Tang, Xun Liu, Yanbo Liang, Wen-Chih Lo, Wanqiang Ji, Byron Hsu, Keqiu Hu, HuiYang Jian, Quan Zhou, and Chien-Min Wang. Apache submarine: A unified machine learning platform made simple. In *European Workshop on Machine Learning and Systems (EuroMLSys)*, pages 101–108. ACM, 2022.

[13] Min-Chi Chiang and Jerry Chou. Dynamoml: Dynamic resource management operators for machine learning workloads. In *International Conference on Cloud Computing and Services Science (CLOSER)*, pages 122–132. SCITEPRESS, 2021.

[14] Chao Du, Zhifeng Gao, Shuo Yuan, Lining Gao, Ziyan Li, Yifan Zeng, Xiaoqiang Zhu, Jian Xu, Kun Gai, and Kuang-Chih Lee. Exploration in online advertising systems with deep uncertainty-aware learning. In *ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD)*, pages 2792–2801. ACM, 2021.

[15] Apache Software Foundation. Dynamic resource allocation, 2024. Spark provides a mechanism to dynamically adjust the resources your application occupies based on the workload. https://spark.apache.org/docs/latest/job-scheduling.html.

[16] Weihao Gao, Xiangjun Fan, Chong Wang, Jiankai Sun, Kai Jia, Wenzhi Xiao, Ruofan Ding, Xingyan Bin, Hui Yang, and Xiaobing Liu. Learning an end-to-end structure for retrieval in large-scale recommendations. In *ACM International Conference on Information and Knowledge Management (CIKM)*, pages 524–533. ACM, 2021.

[17] Priya Goyal, Piotr Dollár, Ross B. Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, large minibatch SGD: training imagenet in 1 hour. *CoRR*, abs/1706.02677, 2017.

[18] Diandian Gu, Yihao Zhao, Yinmin Zhong, Yifan Xiong, Zhenhua Han, Peng Cheng, Fan Yang, Gang Huang, Xin Jin, and Xuanzhe Liu. Elasticflow: An elastic serverless training platform for distributed deep learning. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 266–280. ACM, 2023.

[19] Anthony Hsu, Keqiu Hu, Jonathan Hung, Arun Suresh, and Zhe Zhang. Tony: An orchestrator for distributed machine learning jobs. In *USENIX Conference on Operational Machine Learning (OpML)*, pages 39–41. USENIX Association, 2019.

[20] Meiqi Hu, Chen Wu, and Bo Du. EMS-NET: efficient multi-temporal self-attention for hyperspectral change detection. In *IEEE International Geoscience and Remote Sensing Symposium (IGARSS)*, pages 6664–6667. IEEE, 2023.

[21] Qinghao Hu, Peng Sun, Shengen Yan, Yonggang Wen, and Tianwei Zhang. Characterization and prediction of deep learning workloads in large-scale GPU datacenters. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 104:1–104:15. ACM, 2021.

[22] Qinghao Hu, Meng Zhang, Peng Sun, Yonggang Wen, and Tianwei Zhang. Lucid: A non-intrusive, scalable and interpretable scheduler for deep learning training jobs. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 457–472. ACM, 2023.

[23] Jianbo Huang, Wenli Zhou, Ruoning Song, Fang Liu, Siye Wang, and Jun Liu. Design and performance modeling of a yarn-based gpu resource scheduling system. In *IEEE International Conference on Computer and Communications (ICCC)*, pages 1897–1902. IEEE, 2019.

[24] Po-Sen Huang, Xiaodong He, Jianfeng Gao, Li Deng, Alex Acero, and Larry P. Heck. Learning deep structured semantic models for web search using clickthrough data. In *ACM International Conference on Information and Knowledge Management (CIKM)*, pages 2333–2338. ACM, 2013.

[25] Changho Hwang, Taehyun Kim, Sunghyun Kim, Jinwoo Shin, and KyoungSoo Park. Elastic resource sharing for distributed deep learning. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 721–739. USENIX Association, 2021.

[26] Insu Jang, Zhenning Yang, Zhen Zhang, Xin Jin, and Mosharaf Chowdhury. Oobleck: Resilient distributed training of large models using pipeline templates. In *Symposium on Operating Systems Principles(SOSP)*, pages 382–395. ACM, 2023.

[27] Xianyan Jia, Le Jiang, Ang Wang, Wencong Xiao, Ziji Shi, Jie Zhang, Xinyuan Li, Langshi Chen, Yong Li, Zhen Zheng, Xiaoyong Liu, and Wei Lin. Whale: Efficient giant model training over heterogeneous gpus. In *USENIX Annual Technical Conference (ATC)*, pages 673–688. USENIX Association, 2022.

[28] Biye Jiang, Chao Deng, Huimin Yi, Zelin Hu, Guorui Zhou, Yang Zheng, Sui Huang, Xinyang Guo, Dongyue Wang, Yue Song, et al. XDL: An industrial deep learning framework for high-dimensional sparse data. In *International Workshop on Deep Learning Practice for High-Dimensional Sparse Data*, pages 1–9, 2019.

[29] Ziheng Jiang, Haibin Lin, Yinmin Zhong, Qi Huang, Yangrui Chen, Zhi Zhang, Yanghua Peng, Xiang Li, Cong Xie, Shibiao Nong, Yulu Jia, Sun He, Hongmin Chen, Zhihao Bai, Qi Hou, Shipeng Yan, Ding Zhou, Yiyao Sheng, Zhuo Jiang, Haohan Xu, Haoran Wei, Zhang Zhang, Pengfei Nie, Leqi Zou, Sida Zhao, Liang Xiang, Zherui Liu, Zhe Li, Xiaoying Jia, Jianxi Ye, Xin Jin, and Xin Liu. Megascale: Scaling large language model training to more than 10, 000 gpus. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 745–760. USENIX Association, 2024.

[30] Sofia Ira Ktena, Alykhan Tejani, Lucas Theis, Pranay Kumar Myana, Deepak Dilipkumar, Ferenc Huszár, Steven Yoo, and Wenzhe Shi. Addressing delayed feedback for continuous training with neural networks in CTR prediction. In *ACM Conference on Recommender Systems (RecSys)*, pages 187–195. ACM, 2019.

[31] Kubeflow Community. Kubeflow, 2024. Kubeflow makes artificial intelligence and machine learning simple, portable, and scalable. https://www.kubeflow.org/.

[32] Jiamin Li, Hong Xu, Yibo Zhu, Zherui Liu, Chuanxiong Guo, and Cong Wang. Lyra: Elastic scheduling for deep learning clusters. In *European Conference on Computer Systems (EuroSys)*, pages 835–850. ACM, 2023.

[33] Xiangru Lian, Binhang Yuan, Xuefeng Zhu, Yulong Wang, Yongjun He, Honghuan Wu, Lei Sun, Haodong Lyu, Chengjun Liu, Xing Dong, Yiqiao Liao, Mingnan Luo, Congfei Zhang, Jingru Xie, Haonan Li, Lei Chen, Renjie Huang, Jianying Lin, Chengchun Shu, Xuezhong

Qiu, Zhishan Liu, Dongying Kong, Lei Yuan, Hai Yu, Sen Yang, Ce Zhang, and Ji Liu. Persia: An open, hybrid system scaling deep learning-based recommenders up to 100 trillion parameters. In *ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD)*, pages 3288–3298. ACM, 2022.

[34] Feng Liang, Zhen Zhang, Haifeng Lu, Chengming Li, Victor Leung, Yanyi Guo, and Xiping Hu. Resource allocation and workload scheduling for large-scale distributed deep learning: A survey. *CoRR*, 2024.

[35] Zhuoran Liu, Leqi Zou, Xuan Zou, Caihua Wang, Biao Zhang, Da Tang, Bolin Zhu, Yijie Zhu, Peng Wu, Ke Wang, and Youlong Cheng. Monolith: Real time recommendation system with collisionless embedding table. In *Workshop on Online Recommender Systems and User Modeling co-located with the ACM Conference on Recommender Systems*, volume 3303 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2022.

[36] Kiran Kumar Matam, Hani Ramezani, Fan Wang, Zeliang Chen, Yue Dong, Maomao Ding, Zhiwei Zhao, Zhengyu Zhang, Ellie Wen, and Assaf Eisenman. QuickUpdate: a real-time personalization system for large-scale recommendation models. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 731–744. USENIX Association, 2024.

[37] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. Ray: A distributed framework for emerging AI applications. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 561–577. USENIX Association, 2018.

[38] Dheevatsa Mudigere, Yuchen Hao, Jianyu Huang, Zhihao Jia, Andrew Tulloch, Srinivas Sridharan, Xing Liu, Mustafa Ozdal, Jade Nie, Jongsoo Park, Liang Luo, Jie Amy Yang, Leon Gao, Dmytro Ivchenko, Aarti Basant, Yuxi Hu, Jiyan Yang, Ehsan K. Ardestani, Xiaodong Wang, Rakesh Komuravelli, Ching-Hsiang Chu, Serhat Yilmaz, Huayu Li, Jiyuan Qian, Zhuobo Feng, Yinbin Ma, Junjie Yang, Ellie Wen, Hong Li, Lin Yang, Chonglin Sun, Whitney Zhao, Dimitry Melts, Krishna Dhulipala, K. R. Kishore, Tyler Graf, Assaf Eisenman, Kiran Kumar Matam, Adi Gangidi, Guoqiang Jerry Chen, Manoj Krishnan, Avinash Nayak, Krishnakumar Nair, Bharath Muthiah, Mahmoud khorashadi, Pallab Bhattacharya, Petr Lapukhov, Maxim Naumov, Ajit Mathews, Lin Qiao, Mikhail Smelyanskiy, Bill Jia, and Vijay Rao. Software-hardware co-design for fast and scalable training of deep learning recommendation models. In *International Symposium on Computer Architecture (ISCA)*, pages 993–1011. ACM, 2022.

[39] Dheevatsa Mudigere, Yuchen Hao, Jianyu Huang, Andrew Tulloch, Srinivas Sridharan, Xing Liu, Mustafa Ozdal, Jade Nie, Jongsoo Park, Liang Luo, Jie Amy Yang, Leon Gao, Dmytro Ivchenko, Aarti Basant, Yuxi Hu, Jiyan Yang, Ehsan K. Ardestani, Xiaodong Wang, Rakesh Komuravelli, Ching-Hsiang Chu, Serhat Yilmaz, Huayu Li, Jiyuan Qian, Zhuobo Feng, Yinbin Ma, Junjie Yang, Ellie Wen, Hong Li, Lin Yang, Chonglin Sun, Whitney Zhao, Dimitry Melts, Krishna Dhulipala, K. R. Kishore, Tyler Graf, Assaf Eisenman, Kiran Kumar Matam, Adi Gangidi, Guoqiang Jerry Chen, Manoj Krishnan, Avinash Nayak, Krishnakumar Nair, Bharath Muthiah, Mahmoud khorashadi, Pallab Bhattacharya, Petr Lapukhov, Maxim Naumov, Lin Qiao, Mikhail Smelyanskiy, Bill Jia, and Vijay Rao. High-performance, distributed training of large-scale deep learning recommendation models. *CoRR*, abs/2104.05158, 2021.

[40] Derek G Murray, Jiří Šimša, Ana Klimovic, and Ihor Indyk. tf.data: A machine learning data processing framework. *Proceedings of the VLDB Endowment*, 14(12):2945–2958, 2021.

[41] Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, Xiaodong Wang, Udit Gupta, Carole-Jean Wu, Alisson G. Azzolini, Dmytro Dzhulgakov, Andrey Mallevich, Ilia Cherniavskii, Yinghai Lu, Raghuraman Krishnamoorthi, Ansha Yu, Volodymyr Kondratenko, Stephanie Pereira, Xianjie Chen, Wenlin Chen, Vijay Rao, Bill Jia, Liang Xiong, and Misha Smelyanskiy. Deep learning recommendation model for personalization and recommendation systems. *CoRR*, abs/1906.00091, 2019.

[42] Giang Nguyen, Stefan Dlugolinsky, Martin Bobák, Viet Tran, Álvaro López García, Ignacio Heredia, Peter Malík, and Ladislav Hluchỳ. Machine learning and deep learning frameworks and libraries for large-scale data mining: a survey. *Artificial Intelligence Review*, 52:77–124, 2019.

[43] Laurel J. Orr, Atindriyo Sanyal, Xiao Ling, Karan Goel, and Megan Leszczynski. Managing ML pipelines: Feature stores and the coming wave of embedding ecosystems. *Proc. VLDB Endow.*, 14(12):3178–3181, 2021.

[44] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Z. Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In *Conference*

on Neural Information Processing Systems (NeurIPS), pages 8024–8035, 2019.

[45] Aurick Qiao, Sang Keun Choe, Suhas Jayaram Subramanya, Willie Neiswanger, Qirong Ho, Hao Zhang, Gregory R. Ganger, and Eric P. Xing. Pollux: Co-adaptive cluster scheduling for goodput-optimized deep learning. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX Association, 2021.

[46] Steffen Rendle. Factorization machines. In *IEEE International Conference on Data Mining (ICDM)*, pages 995–1000. IEEE Computer Society, 2010.

[47] Alexander Sergeev and Mike Del Balso. Horovod: Fast and easy distributed deep learning in tensorflow. *CoRR*, abs/1802.05799, 2018.

[48] Chijun Sima, Yao Fu, Man-Kit Sit, Liyi Guo, Xuri Gong, Feng Lin, Junyu Wu, Yongsheng Li, Haidong Rong, Pierre-Louis Aublin, and Luo Mai. Ekko: A Large-Scale deep learning recommender system with Low-Latency model update. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 821–839. USENIX Association, 2022.

[49] Pawel Swietojanski, Jinyu Li, and Steve Renals. Learning hidden unit contributions for unsupervised acoustic model adaptation. *IEEE ACM Trans. Audio Speech Lang. Process.*, 24(8):1450–1463, 2016.

[50] Taegeon Um, Byungsoo Oh, Byeongchan Seo, Minhyeok Kweun, Goeun Kim, and Woo-Yeon Lee. Fastflow: Accelerating deep learning model training with smart offloading of input data pipeline. *Proc. VLDB Endow.*, 16(5):1086–1099, 2023.

[51] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Annual Conference on Neural Information Processing Systems NeurIPS*, pages 5998–6008, 2017.

[52] Qinlong Wang, Tingfeng Lan, Yinghao Tang, Bo Sang, Ziling Huang, Yiheng Du, Haitao Zhang, Jian Sha, Hui Lu, Yuanchun Zhou, Ke Zhang, and Mingjie Tang. DLRover-RM: Resource optimization for deep recommendation models trainin in the cloud. *Proc. VLDB Endow.*, 17(12):4130–4144, 2024.

[53] Zheng Wang, Yuke Wang, Boyuan Feng, Dheevatsa Mudigere, Bharath Muthiah, and Yufei Ding. EL-Rec: Efficient large-scale recommendation model training via tensor-train embedding table. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 70:1–70:14. IEEE, 2022.

[54] Wei Wei, Xubin Ren, Jiabin Tang, Qinyong Wang, Lixin Su, Suqi Cheng, Junfeng Wang, Dawei Yin, and Chao Huang. LLMRec: Large language models with graph augmentation for recommendation. In *ACM International Conference on Web Search and Data Mining (WSDM)*, pages 806–815. ACM, 2024.

[55] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, Fan Yang, and Lidong Zhou. Gandiva: Introspective cluster scheduling for deep learning. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 595–610. USENIX Association, 2018.

[56] Wencong Xiao, Shiru Ren, Yong Li, Yang Zhang, Pengyang Hou, Zhi Li, Yihui Feng, Wei Lin, and Yangqing Jia. AntMan: Dynamic scaling on GPU clusters for deep learning. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 533–548. USENIX Association, 2020.

[57] Lei Xie, Jidong Zhai, Baodong Wu, Yuanbo Wang, Xingcheng Zhang, Peng Sun, and Shengen Yan. Elan: Towards generic and efficient elastic training for deep learning. In *IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 78–88. IEEE, 2020.

[58] Chih-Chieh Yang and Guojing Cong. Accelerating data loading in deep neural network training. In *IEEE International Conference on High Performance Computing, Data, and Analytics (HiPC)*, pages 235–245. IEEE, 2019.

[59] Zhengyi Yang, Xiangnan He, Jizhi Zhang, Jiancan Wu, Xin Xin, Jiawei Chen, and Xiang Wang. A generic learning framework for sequential recommendation with distribution shifts. In *International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR)*, pages 331–340. ACM, 2023.

[60] Buyun Zhang, Liang Luo, Yuxin Chen, Jade Nie, Xi Liu, Shen Li, Yanli Zhao, Yuchen Hao, Yantao Yao, Ellie Dingqiao Wen, Jongsoo Park, Maxim Naumov, and Wenlin Chen. Wukong: Towards a scaling law for large-scale recommendation. In *International Conference on Machine Learning (ICML)*. OpenReview.net, 2024.

[61] Chengliang Zhang, Minchen Yu, Wei Wang, and Feng Yan. MArk: Exploiting cloud services for cost-effective, slo-aware machine learning inference serving. In *USENIX Annual Technical Conference (ATC)*, pages 1049–1062, 2019.

[62] Haoyu Zhang, Logan Stafman, Andrew Or, and Michael J. Freedman. SLAQ: quality-driven scheduling for distributed machine learning. In *Symposium on Cloud Computing (SoCC)*, pages 390–404. ACM, 2017.

[63] Yiqian Zhang, Yinfu Feng, Wen-Ji Zhou, Yunan Ye, Min Tan, Rong Xiao, Haihong Tang, Jiajun Ding, and Jun Yu. Multi-domain deep learning from a multi-view perspective for cross-border e-commerce search. In *AAAI Conference on Artificial Intelligence (AAAI)*, pages 9387–9395. AAAI Press, 2024.

[64] Hanyu Zhao, Zhi Yang, Yu Cheng, Chao Tian, Shiru Ren, Wencong Xiao, Man Yuan, Langshi Chen, Kaibo Liu, Yang Zhang, Yong Li, and Wei Lin. GoldMiner: Elastic scaling of training data pre-processing pipelines for deep learning. *ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 1(2):193:1–193:25, 2023.

[65] Laiping Zhao, Fangshu Li, Wenyu Qu, Kunlin Zhan, and Qingman Zhang. AITurbo: Unified compute allocation for partial predictable training in commodity clusters. In *International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*, pages 133–145. ACM, 2021.

[66] Mark Zhao, Niket Agarwal, Aarti Basant, Bugra Gedik, Satadru Pan, Mustafa Ozdal, Rakesh Komuravelli, Jerry Pan, Tianshu Bao, Haowei Lu, Sundaram Narayanan, Jack Langman, Kevin Wilfong, Harsha Rastogi, Carole-Jean Wu, Christos Kozyrakis, and Parik Pol. Understanding data storage and ingestion for large-scale deep recommendation model training: Industrial product. In *International Symposium on Computer Architecture (ISCA)*, pages 1042–1057. ACM, 2022.

[67] Guorui Zhou, Xiaoqiang Zhu, Chengru Song, Ying Fan, Han Zhu, Xiao Ma, Yanghui Yan, Junqi Jin, Han Li, and Kun Gai. Deep interest network for click-through rate prediction. In *ACM SIGKDD International Conference on Knowledge Discovery & Data Mining (KDD)*, pages 1059–1068. ACM, 2018.

[68] Jun Zhou, Ke Zhang, Feng Zhu, Qitao Shi, Wenjing Fang, Lin Wang, and Yi Wang. ElasticDL: A kubernetes-native deep learning framework with fault-tolerance and elastic scheduling. In *ACM International Conference on Web Search and Data Mining (WSDM)*, pages 1148–1151. ACM, 2023.

Chitu: Avoiding Unnecessary Fallback in Byzantine Consensus

Rongji Huang[1*], Xiangzhe Wang[1*], Xiaofeng Yan[1*], Lei Fan[1], Guangtao Xue[1,2], and Shengyun Liu[1,2#]

[1]*Shanghai Jiao Tong University*
[2]*Shanghai Key Laboratory of Trusted Data Circulation, Governance and Web3*

Abstract

Most Byzantine-Fault Tolerant (BFT) consensus protocols either pre-select a single leader with the help of additional timing assumptions (i.e., partially synchronous ones) or resort to random coins to achieve only probabilistic termination (i.e., asynchronous ones). The single leader may become a performance bottleneck and/or lead to availability problems, while probabilistic termination increases latency.

We re-consider the consensus problem from its first principles, where neither synchrony assumption or any designated role, nor randomization is intrinsic to consensus. We thus formally study a framework for designing robust BFT protocols with low latency: nodes first try to achieve consensus merely based on message exchange, but only resort to a fallback mechanism like random coin or leader election if correct nodes have divergent opinions on a proposal. We further present Chitu, an asynchronous DAG-based protocols following this framework. Chitu in the best case commits proposals in four message delays, even in the presence of faulty nodes and/or under asynchrony. In the worst case, Chitu still ensures predictable performance with $O(1)$ time complexity in expectation. Experimental results on Amazon EC2 show that Chitu achieves a significant reduction in latency compared to two representative DAG-based protocols that always put a leader or randomization on the execution path.

1 Introduction

Blockchains [8,69], or Byzantine-Fault Tolerance (BFT) state-machine replication (SMR) protocols [27, 44, 45, 80] are a promising approach to providing robust services for decentralized applications. Boosting the performance of BFT protocols is thus becoming a hot topic in recent years [16, 71, 77, 79]. At the core of SMR protocols is a consensus algorithm that reaches agreement among a group of replicas or *nodes*. Byzantine consensus protocols [58] further assume faulty nodes can exhibit arbitrary or malicious behaviors. Consensus protocols are roughly categorized into three types according to their timing assumptions: synchronous [34, 58, 67], partially synchronous [22, 27, 37, 80] and asynchronous [21, 32, 62, 66]. Synchronous or partially synchronous protocols rely on network synchrony to ensure safety or liveness property, while asynchronous variants resort to randomization to achieve only probabilistic termination due to FLP impossibility [40].

Many existing BFT protocols [22, 27, 44, 48, 56, 80] are leader-based and designed assuming the partial synchrony model, in which a single leader is elected by the help of network synchrony assumption. The leader is responsible for making a proposal and coordinating consensus process. If the leader (and the network) is stable, non-faulty nodes can reach consensus within *a fixed number of communication phases*, achieving promising performance. However, partially synchronous protocols rely on a complicated view synchronization mechanism [28, 70] to keep every correct node in sync with one another, and a predefined maximum network delay Δ to help trigger the replacement of a problematic leader (a.k.a., view change). During view change (which lasts at least Δ time), no available service can be provided. Even worse, if network synchrony assumption does not hold, partially synchronous protocols may be involved in consecutive view changes, forfeiting liveness property. Network problems can occur more often than not in Wide-Area-Network (WAN) [38], which is the typical setting for blockchain systems [48].

Asynchronous protocols, in contrast, have no synchrony assumptions and do not pre-select a leader [21, 25, 26, 32, 36, 47, 66]. Asynchronous protocols require every node to make a proposal in each consensus instance, and participate in several phases of message exchange. Finally, a coin-flipping phase is used to help non-faulty nodes converge and reach a final agreement. With probability one, every non-faulty node outputs the same set of committed proposals. Recently-proposed asynchronous protocols typically select a leader a posteriori in order to achieve $O(1)$ time complexity in expectation [32, 47, 53].

*These authors contributed equally and are listed in alphabetical order
#Corresponding author: shengyun.liu@sjtu.edu.cn

Although asynchronous protocols are becoming prevalent, they often incur more rounds of communications compared to their partially synchronous counterparts [46, 63]. We argue that the problem mainly comes from the coin-flipping phase when it is located on the critical path. The appeal of partially synchronous protocols is in their simplicity when the network is stable most of the time, with predictable latency between correct nodes, and when protocol execution path *skips* costly timeout event and view change mechanism, achieving *fast termination*.

Applying a fast path (often with optimistic assumptions) to normal-case operations is a widely-used approach for boosting system performance, and also has profound influence on consensus protocols [30, 56, 57, 62]. Actually, several up-to-date asynchronous protocols [63] adopt a fast path that mimics the normal case operations of partially synchronous protocols and a slow path that still provides liveness even under asynchrony. We can view them as combining assumptions and essential steps of both worlds. Therefore, the fast path of these protocols still relies on some special role (e.g., the single leader) and synchrony assumptions to move forward. In other words, their fast path possesses both advantages and disadvantages of partially synchronous protocols.

Meanwhile, we also found two recent works that enable fast termination in a pretty different way: Red Belly [30] (using DBFT [29]) under partial synchrony and MyTumbler [62] under asynchrony. Red Belly requires every node to make a proposal in each consensus instance. Unlike other partially synchronous counterparts, Red Belly uses a binary agreement (BA) protocol to determine the decision of each proposal (either committed or aborted). By doing so, Red Belly resorts to a lightweight coordinator in BA to break ties *only if* correct nodes are divergent on some proposal[1]. In a similar vein, MyTumbler (under asynchrony) skips random coins in its BA protocol if most correct nodes are consistent on a proposal.

If we consider both leader (coordinator) election and coin flipping a fallback mechanism that sidesteps FLP impossibility [40], Red Belly and MyTumbler can entirely skip the fallback via their fast paths. This departs from other fast path design in the *absence* of any special role. Although the above two mentioned protocols have such an appealing feature, Red Belly still puts timers on its fast path, and both of them use binary agreement (BA), meaning that they have to explicitly resolve ambiguity of at most n BA instances.

Inspired by Red Belly and MyTumbler, we first formally study a framework for designing robust BFT protocols with low latency: correct nodes first try to reach agreement merely based on message exchange (which we call the *fast path*), but only resort to a fallback mechanism like random coins or leader election when correct nodes have divergent opinions on some proposal. Figure 1 depicts different frameworks. Partially synchronous protocols (except Red Belly) pre-select a special role (such as the leader) with the help of additional timing assumptions, while asynchronous BFT protocols (except MyTumbler) either put randomization on the critical path or re-introduce synchrony assumptions and some special role in their fast path. In contrast, the framework we study has the following unique features:

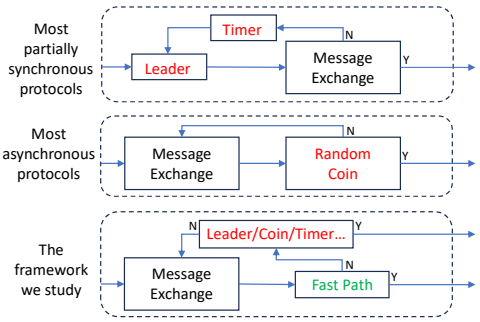


Figure 1: Frameworks for consensus protocols.

- In the fast path there is no special role (such as the leader) used or elected, and thus *no* synchrony assumption is re-introduced; and,
- there is no switch between fast and normal paths, as both can proceed concurrently and in a unified manner.

From a different viewpoint, the model of the fast path being studied in this paper could be considered a variant of fair schedulers [21]. Fair scheduler was proposed four decades ago, but was somehow overlooked from engineering perspectives. The main drawback of fair scheduler is that if the adversary controls message scheduling at will, the protocols no longer provide liveness. Hence, a *fallback* mechanism such as leader election and/or randomization is essential for mitigating such attacks targeting the fast path.

We name the framework we study Fair-Fallback. Following this framework, we further present Chitu, an asynchronous BFT protocol designed on top of a well-structured Directed Acyclic Graph (DAG). Each proposal in a DAG is represented as a vertex. DAG-based BFT protocols are becoming prevalent in permissionless blockchains [11, 32, 53, 60, 76]. They embed the consensus process in the dynamic construction of a DAG structure without involving additional messages, simplifying protocol design and implementation. Chitu is largely inspired by Tusk [32] and DAG-Rider [53], two recently proposed DAG-based protocols. Besides possessing the same $O(1)$ time complexity in expectation in the worst case, Chitu further bypasses the fallback mechanism (i.e., the single leader vertex in DAG) provided that most nodes have observed the same set of vertices in a round (i.e., a consensus instance), thus achieving fast termination.

Guided by the Fair-Fallback, the next challenge for Chitu is to find a way to commit as many proposals as possible through the fast path, which we found is a non-trivial task. After several attempts at devising and evaluating different commit rules and also according to the design of prior work [30, 62], we

[1] E.g., some nodes input 1 into a BA instance, while others input 0.

located a useful rule of thumb: *a node should wait as long as possible for a proposal to enter the fast path, provided that this wait does not introduce significant costs or violate liveness properties.* To this end, we further propose an adaptive wait mechanism that helps Chitu dramatically increase the chance of achieving fast termination.

Finally, we evaluate Chitu, Tusk and BullShark [76] on Amazon EC2 platform. BullShark is another DAG protocol designed under partial synchrony. The experimental result shows that in fault-free case, Chitu can reduces end-to-end latency, achieving up to 82.5% reduction compared to Tusk. Under crash faults, Chitu can still go through the fast path, while the performance of BullShark is significantly affected by the faulty node. We also mimic an adversarial situation under Byzantine attack, where each round is prolonged with extra waiting and the fast path is disabled. Even if all the proposals go through random coins in this case, the performance of Chitu degrades gracefully.

2 Background

2.1 Model

We focus on the Blockchain or state-machine replication (SMR) problem [73], a solution to which is a robust distributed system that coordinates a group of processes or nodes, in order to mask a bounded number of failures. SMR protocols order requests issued by clients, such that:

- (Safety) If two requests $req1$ and $req2$ are committed in the same position (or with the same block height in blockchain contexts), then $req1 = req2$;
- (Liveness) Eventually every request req issued by a correct client can be committed.

We target Byzantine faults, which may exhibit arbitrary behaviors even collusion attacks. For simplicity, we assume there are a group Π of $n = 3f+1$ nodes [37], in which at most f nodes may be faulty. Thus, a Byzantine quorum consists of $\lceil \frac{n+f+1}{2} \rceil = n - f = 2f + 1$ nodes. We assume reliable and authenticated channels between each pair of correct nodes, meaning that any message sent by a correct node will be eventually delivered by every correct receiver. Asynchronous protocols must rely on reliable channels to provide liveness, so does the fast path of Chitu.

2.2 Fast termination and the leader's bottleneck in partially synchronous protocols

Partially synchronous protocols are commonly separated into two sub-routines: the normal-case operations that allow a single leader (e.g., the primary in PBFT [27] or the leader in HotStuff [80]) to coordinate consensus and make its proposal (i.e., a batch of requests or transactions) committed, and the view-change operations that elect a new leader when, e.g., the previous one has crashed. The safety property of partially synchronous protocols is ensured by quorum-based techniques [78]. The liveness property, however, is ensured by further introducing network synchrony between the single leader and others. Such timing assumptions also help trigger the view-change operations. When the leader is correct and the network is stable, partially synchronous protocols can reach consensus within a fixed number of message exchange, achieving deterministic and *fast termination*. In summary,

- partially synchronous protocols have normal-case operations that rely on a single leader to reach consensus without involving time-related steps;
- the leader and other $2f$ correct nodes must remain in the same view for a sufficiently long period of time, such that consensus can be achieved before timeout.

Note that the second reason is critical in ensuring liveness and providing good performance. Otherwise, partially synchronous protocols may be involved in consecutive view changes.

When the leader is slow or has crashed, partially synchronous protocols incur more communication phases and additional timeout events for electing a new leader. We argue that the problem mainly stems from the existence of some special role, which is always on the critical path regardless of the system state.

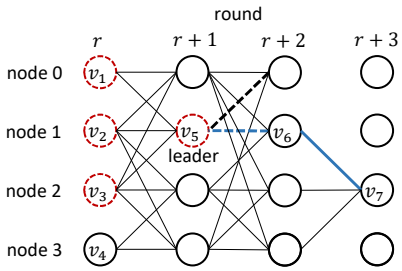


Figure 2: Example of commit rules in DAG-based protocols like Tusk [32] when $n = 4$ and $f = 1$. Each circle represents a vertex in the DAG. Dotted circles represent vertices committed by the leader (v_5), while dotted lines represent edges making the leader valid.

2.3 Randomization in asynchronous protocols

Asynchronous protocols [13, 32, 36, 47, 53, 66, 72] work in a somewhat leaderless manner, in the sense that no special role (such as the leader) is pre-selected by the help of timing assumptions. Asynchronous protocols are commonly round-based, where in each round every node is required to suggest a value and participate in several communication phases. The value a node suggests in a round is influenced by message exchange of previous rounds. To determine which value(s) can be committed, asynchronous protocols introduce a coin-flipping phase, which breaks ties and helps correct nodes converge.

Figure 2 depicts how Tusk [32], a recently proposed DAG-based asynchronous protocol, proceeds round by round. Each round corresponds to a new consensus instance. Since consecutive rounds are connected by edges, the execution of consecutive consensus instances is pipelined, similarly to the way HotStuff [80] chains consecutive proposals.

In each round r, a node i makes a proposal, i.e., a vertex in the DAG. Node i then propagates the vertex to others via a Byzantine Reliable Broadcast (BRB) protocol [21, 24]. BRB ensures that the delivered message (vertex) is the same for all correct nodes despite f Byzantine faults. BRB in Tusk takes three phases of message exchange. Each vertex of round $r+1$ has exactly $n-f$ edges connecting to vertices of round r. For example, vertex v_5 in Figure 2 connects to v_1, v_2 and v_3 (but not v_4). DAG is merely a partially ordered set, in which vertices are *not* totally ordered. To this end, Tusk generates a random coin in each odd round (e.g., round $r+1$ in Figure 2), by which correct nodes collectively select a leader vertex (e.g., v_5). Random coins can be instantiated using a deterministic threshold signature scheme [19]. If the leader vertex is *valid*, i.e., is connected by at least $f+1$ vertices of the subsequent round (e.g., v_5), it recursively commits all the ancestors in the DAG in a deterministic manner. Such a validity check is used to ensure there is a path (in the DAG) between a valid leader vertex and any subsequent leader vertex. For instance, if v_7 in Figure 2 is selected as the leader vertex of round $r+3$, there must be a path connecting v_7 and v_5 because v_5 is connected by $f+1$ vertices of round $r+2$ and v_7 must connect to $n-f$ vertices of round $r+2$. By ensuring this property, no one may miss or skip any valid leader vertex but commit them in the same order.

The random coin of round $r+1$ is generated when $n-f$ nodes have entered round $r+3$,[2] as the leader information should not be revealed in advance. This is to ensure with non-zero probability the selected leader vertex is valid. Otherwise, if an adversary that controls message delivery knows the coin in advance, it may always keep the corresponding leader vertex connected by less than $f+1$ vertices, forfeiting liveness properties. As the protocol proceeds round by round, with probability 1 some new leader vertex and its ancestors can be committed.

As we can observe, (1) compared to partially synchronous counterparts like PBFT, Tusk involves more rounds of message exchange, even in the absence of node/network failures; and, (2) random coins are always located on the execution path, thus making termination probabilistic. If the selected leader vertex is not valid, at least another two rounds are involved. In practice, such a delayed commit can happen when, e.g., the proposer of the leader vertex is slow or has crashed.

The design choices of Tusk (and most asynchronous protocols) are solely based on a pessimistic assumption: an adversarial scheduler can always control message scheduling at will. Hence, Tusk has only one path for committing proposals, i.e., by selecting a leader vertex with the help of random coins. In practice, such an omnipotent adversary does not (always) exist and the network performance is predictable most of the time [43, 61].

[2] At the time when most edges between rounds $r+1$ and $r+2$ are confirmed.

3 The Fair-Fallback Framework

3.1 Motivation and first principles of consensus

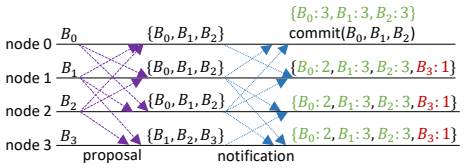
We simplify the discussion by focusing on one-shot consensus. Any solution to the multivalued consensus problem requires each *proposer* i to make a proposal B_i. After several phases of message exchange, correct nodes eventually *decide* on a single proposal or a subset of all proposals [65].

Generally, we can also consider any consensus protocol running m Binary Agreement (BA) instances (implicitly or explicitly), each of which determines the result of a given proposal. Eventually, each proposal is either committed (output 1) or aborted (output 0). Following the concept and terminology used in the seminal FLP impossibility [40], once a decision can be made, we say each proposal B_i becomes *univalent*, i.e., either *1-valent* or *0-valent*. Initially, every proposal is *bivalent*. For leader-based protocols, only a single proposal becomes 1-valent, and it is also the final decision. Other proposals are 0-valent. For leaderless variants such as asynchronous protocols, multiple proposals may become 1-valent and be included in the final decision.

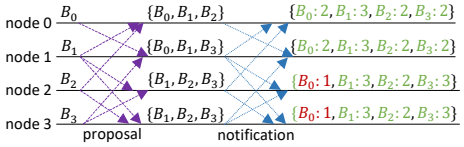
We observe that the general pattern discussed above neither introduces a leader or any special role, nor does it rely on random coins to break ties. In other words, *neither any special role (or the maximum network delay Δ) nor randomization is intrinsic to the consensus problem*. If correct nodes can reach agreement of each proposal merely based on message exchange, no additional assumptions or mechanisms should be put on the execution path. If we target the more practical multi-instance consensus problem (or SMR), where a series of consensus instances are sequentially ordered, either we can consider this order obtained out-of-band (e.g., pre-defined sequence numbers), or for DAG protocols the edges in DAG natively represent the partial order between proposals. In both cases, no special role exists in consensus.

3.2 A strawman protocol

We present a simple strawman protocol to better illustrate the fast path of our approach. To simplify our discussion, we further assume (1) each node acts both as a proposer and as a validator, and (2) there is no equivocation and every node is benign (i.e., correct or crashed), though we still have $n = 3f+1$ nodes. In §4 we give full-fledged Chitu which also tolerates Byzantine faults.



(a) Node 0 decides on B_0, B_1 and B_2, while every node selects B_0, B_1 and B_2 as the candidates.



(b) Nodes 0 and 1 select B_0, B_1, B_2 and B_3 as the candidates, while nodes 2 and 3 exclude B_0.

Figure 3: Example for the strawman protocol ($n = 4$).

First, ① each (correct) node i makes a proposal B_i and propagates B_i to others. As at most f nodes may be faulty, ② after receiving $n-f$ proposals, node i casts a vote on each proposal by notifying others of all the proposals received. Finally, ③ after receiving $n-f$ such notifications (again, the other f nodes may be faulty), node i selects the proposals that have each obtained $f+1$ votes as the *candidates* for final decisions.

For each proposal B, there are three cases, where the first two are univalent:

- (**1-valent**) If at least $n-f$ nodes have notified of B, we say B is 1-valent;
- (**0-valent**) If at least $n-f$ nodes have *not* notified of B (if B exists), we say B is 0-valent; and,
- (**bivalent**) If neither of the above conditions is met, we say B is bivalent.

Fast path. If for each proposal B, B is either 1-valent or 0-valent, we can safely commit all the 1-valent proposals and abort 0-valent ones. This works because every node in step ③ will select the same candidates (i.e., only 1-valent ones). Even with further communication and/or any additional mechanism, correct nodes will eventually commit the same set of proposals. For instance, in Figure 3a, node 0 immediately commits B_0, B_1 and B_2 upon observing they are 1-valent (as included in $n-f$ notifications) and B_3 is 0-valent (as not included in $n-f$ notifications). Other nodes will only select B_0, B_1 and B_2 (but not B_3) as the candidates for further negotiations.

On the contrary, if any proposal B is bivalent, more communication and/or additional mechanism is needed in order to determine the final result of each bivalent proposal. For instance, in Figure 3b, B_1, B_2 and B_3 are 1-valent, but B_0 is bivalent. Actually, the main challenge for any (multi-instance) consensus protocol is to obtain a total order on committed proposals. Thus, the fast path of our protocol may fail due to a possible discrepancy in the final order. That is why a fallback mechanism (like leader election or random coin) is needed.

3.3 The framework

There are several ways to reach final agreement on bivalent proposals. We can repeat the notification step described in §3.2 (i.e., step ②), where each node only notifies of the selected candidates. If with *non-zero* probability each repeat results in a univalent status, the protocol can itself achieve probabilistic termination. Such a model, a.k.a., fair scheduler, was formally described in the seminal work of Bracha and Toueg [21].

However, if Byzantine nodes precisely control the delivery speed of their own messages, they may always keep their own proposals bivalent (e.g., node 0 and proposal B_0 in Figure 3b), thus breaking liveness properties. To mitigate this problem, we must rely on a fallback mechanism to break ties. We first integrated a light-weight leader into the fast path. In contrast to partially synchronous protocols, we do not rely on synchrony assumptions (or any timer) but only leverage a pre-selected leader to resolve ambiguity, which provides probabilistic termination if Byzantine nodes only manipulate the delivery speed of their own messages but not that of correct nodes.

In the worst case, if the adversary also controls message scheduling of correct nodes, some proposal may forever be bivalent and the pre-selected leader may not work. To defend against such a powerful adversary, existing asynchronous protocols explicitly introduce random coins to make correct nodes converge. Similarly, we further integrate random coins into the fast path and obtain Chitu (in a similar vein as Tusk, see §2.3), if the problem needs to be dealt with.

Maybe more interestingly, if we assume the adversary can control message scheduling but cannot (always) break synchrony assumptions, we can instead integrate Δ (the maximum network delay) into the fast path and obtain a partially synchronous protocol. Thanks to the simplicity of DAG structure, we can readily apply the fast path to Bullshark [76] by pre-selecting a leader in a round-robin fashion and waiting Δ time for each leader vertex. We may even remove timeouts (but keep leaders) and obtain an asynchronous protocol under fair schedulers [21], which refrains from using a burdensome random number generation protocol (e.g., threshold signature). Since they share very similar features with Chitu, in this paper we only focus on Chitu. Note that in all above-mentioned protocols a fallback mechanism (i.e., random coins or pre-selected leaders) is needed *only if* some proposals are bivalent. The framework we use is depicted in Figure 4.

Generalization of the fast path. There can be different ways to instantiate the fast path. In §3.2 (and §4) we only give one concrete solution. Generally, assume the conditions for the fast path to commit a proposal are denoted as $\mathcal{CON}_{\mathcal{C}}$ (e.g., 1-valent ones in §3.2), while the conditions for the fast path to abort a proposal are $\mathcal{CON}_{\mathcal{A}}$ (e.g., 0-valent ones in §3.2). Further assume the conditions for the fallback to commit a proposal are $\mathcal{CON}_{\mathcal{F}}$. Then, the following two invariants must

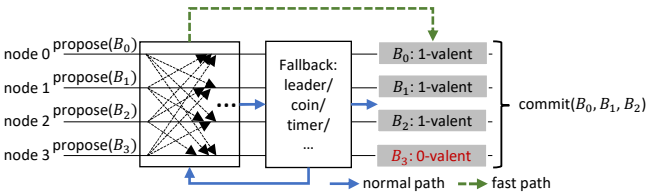


Figure 4: The Fair-Fallback framework.

hold:

- If any proposal B satisfies $\mathcal{CON}_{\mathcal{C}}$, then B must also satisfy $\mathcal{CON}_{\mathcal{F}}$; and,
- If any proposal B satisfies $\mathcal{CON}_{\mathcal{A}}$, then B must *not* satisfy $\mathcal{CON}_{\mathcal{F}}$.

These two variants guarantee that even if some nodes go through the fast path, while others proceed by the fallback, they still obtain the same result for each proposal.

There are two remaining questions: (1) how to ensure safety properties under Byzantine faults; and, (2) how to efficiently integrate the fast path into a fallback mechanism, such that there is no switch between two paths and each path commits the same set of proposals. For the first question, every message in our strawman protocol (i.e., proposal and notification) is disseminated through a Byzantine Reliable Broadcast (BRB) protocol, such that a Byzantine sender cannot make correct nodes deliver distinct messages. For the second question, our full-fledged solution (see §4) is inspired by recently-proposed DAG protocols [32, 53, 76]. The fallback mechanism (i.e., the selected leader vertices) is embedded into the dynamic construction of DAG, enabling consensus to be achieved without sending extra messages, so is the fast path. With the same DAG and the same leader selected by random coins, no matter whether nodes proceed through the fast path or the fallback, they commit the same set of (1-valent) proposals.

4 The Chitu Protocol

Chitu is inspired by Tusk [32] and DAG-Rider [53], both of which adopt DAG as the underlying structure. Chitu differs from Tusk and DAG-Rider in that it is further enhanced by a fast path relying on no special role. In the best case, Chitu takes only four communication phases to commit a proposal. In the following, we first discuss the main idea of Chitu in §4.1. We then describe the detail of Chitu in §4.2-§4.5 and sketch its proof in §4.6. We postpone its pseudocode and correctness proof to Appendices A and B due to space limit.

4.1 Main idea

Chitu dynamically constructs a DAG in a round-by-round manner. In each round, every (correct) node proposes a vertex in the DAG and propagates it to others via a Byzantine Reliable Broadcast (BRB) protocol (see §4.2 for details). BRB guarantees that the sender can make at most one vertex delivered in a round. Hence, each round has at most $n = 3f+1$ and at least $n-f = 2f+1$ vertices delivered via BRB. A vertex v in round r has at least $n-f$ edges connecting to vertices associated with round $r-1$. For ease of explanation, we first give the following definition:

Definition 4.1 (Observe)**.** *We say vertex v in round r observes vertex u in round $r' < r$, if there is a path (in the DAG) from v to u.*

The first challenge for Chitu is to embed the message pattern of our strawman approach (§3.2) into a DAG structure, as consensus instances in DAG are naturally pipelined. To this end, a vertex v in round r simultaneously represents a proposal (of round r) and a notification of the proposals (of round $r-1$) observed by v. Besides, as discussed in §2.3, the selected leader vertices in DAG solely determine the set of committed (or 1-valent) proposals, which are ancestors of leaders. When the fast path is enabled, each leader must be able to distinguish between 1-valent and 0-valent vertices and commits only 1-valent ones.

Assume vertices u and u' are in round r and vertex v is in round $r+2$. We note that when u is observed by $n-f$ vertices of round $r+1$ (denoted as group $\mathcal{G}$), at least $f+1$ of those are observed by v. This is because every vertex in round $r+2$ (e.g., v) observes $n-f = 2f+1$ vertices, among which at least $f+1$ are in $\mathcal{G}$. On the contrary, if at least $n-f$ vertices of round $r+1$ have not observed u', there exist *no* $f+1$ vertices (of round $r+1$) observing u'. The above-mentioned properties provide us a way to distinguish between 1-valent and 0-valent vertices.

We say a node i *decides* a round r, if every proposal of round r has been decided univalent from i's perspective. Round r can be decided in either of the following two ways:

1. (fast path) when vertices of round $r+1$ help decide round r; and,
2. (normal path) when a selected leader of round $r' > r+1$ helps decide round r.

To ensure the safety property, the following invariant must hold:

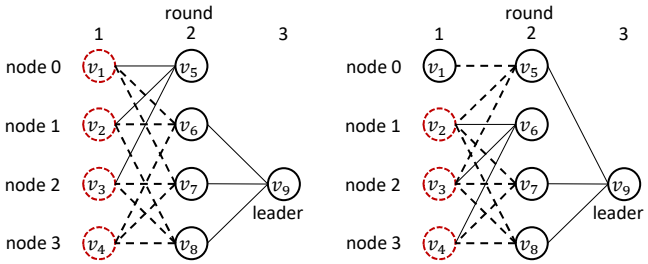
Invariant 4.1. *For any round r, if a group of vertices $\mathcal{V}$ is decided 1-valent by node i via the fast path, while a group of vertices $\mathcal{V}'$ is decided 1-valent by node j via the normal path, then $\mathcal{V} = \mathcal{V}'$.*

We then give the following definition.

Definition 4.2 (Strongly observe)**.** *We say vertex v in round $r' \geq r+2$ strongly observes vertex u in round r, if there are (at least) $f+1$ vertices of round $r+1$ connecting v and u.*

Correspondingly, we have the following important invariants:

Invariant 4.2. *if vertex u in round r is observed by $2f+1$ vertices of round $r+1$, then every vertex of round $r' \geq r+2$ strongly observes u.*



(a) Case I: all 1-valent (b) Case II: some 0-valent (v_1)

Figure 5: Example for Chitu fast path ($n = 4$). Dotted circles denote 1-valent vertices, while dotted lines denote edges enabling the leader vertex to commit the same set of vertices.

Invariant 4.3. *if vertex u' in round r is observed by less than $f+1$ vertices of round $r+1$, then no vertex of round $r' \geq r+2$ strongly observes u'.*

Invariants 4.2 and 4.3 play crucial roles in designing Chitu. For each vertex u in round r, if *either* u is observed by at least $2f+1$ vertices of round $r+1$, *or* u cannot be observed by $f+1$ vertices of round $r+1$, round r can be decided. In this case, if node i decides round r but node j (temporarily) skips round r and decides $r+2$, any vertex in round $r+2$ can distinguish between those that are strongly observed and those not, and *only decide strongly observed ones in round r*. Generally, the leader of round $r' = r+2k$ ($k \in \mathbb{N}^+$) can help decide round r in an indirect way. Figure 5 gives an example.

4.2 DAG construction

Each vertex v represents a proposal broadcast by a single node. It contains basic information such as a round number r, a source which identifies the node who created v, a block of transactions, and, most importantly, a set of edges which refer to vertices of the last round. Each node maintains a local copy of the DAG according to messages it receives. A vertex v is added into the DAG if v is delivered by BRB. A node i enters round $r+1$ and makes a new proposal if node i has delivered $n-f$ vertices of round r.

Reliable Broadcast (BRB). In Chitu reliable broadcast protocol, a vertex goes through two communication phases. We denote two types of messages by VAL and PREPARE. Each node i first broadcasts a VAL message when starting a new round r, containing a new vertex v proposed by i. Upon receiving VAL message or $f+1$ PREPARE messages for the same vertex v, node i checks the validity and broadcasts a signed PREPARE message. As an optimization, $f+1$ PREPARE messages guarantee that the corresponding VAL message is received by at least one correct node and hence can be received by i eventually. Upon receiving $n-f$ PREPARE messages for the same vertex v, v can be delivered, i.e. added into the local DAG. Note that if node i receives v from node j but has not delivered some vertex v' connected by v, it can ask j for v' that must be in j's local DAG. After each of $n-f$ vertices in round r collects $n-f$ PREPARE messages, node i can advance to the next round (§4.4 presents an optimization that delays the advancement).

Weak edges. As $n-f$ nodes may quickly move forward, vertices proposed by other slow nodes may be constantly ignored. To solve this problem, Chitu can also leverage *weak edges* proposed in DAG-rider [53], enabling vertex v in round r to weakly connect to vertex v' in round $r' < r-1$. Then, v' can be committed if v is committed. Since vertices proposed by a single node are chained and totally ordered, each vertex only needs to connect to at most $n = 3f+1$ vertices in previous rounds, each of which is from a different node.

4.3 Commit rules

Each node decides vertices round by round merely based on its local view of the DAG. For a vertex v in round r, it connects at least $n-f$ vertices of round $r-1$, which can be considered the proposer of v votes for vertices connected by v but vetoes others not.

Fast path. Through the fast path, round $r+1$ helps decide round r when node i delivers $n-f$ or more vertices of round $r+1$. We denote three statuses for a vertex v in round r:

- *1-valent*: v is 1-valent if at least $n-f$ vertices of round $r+1$ observe v.
- *0-valent*: v is 0-valent if at least $n-f$ vertices of round $r+1$ do not observe v.
- *bivalent*: v is bivalent if it is possible that more than f but less than $n-f$ vertices of round $r+1$ observe v.

Both 1-valent and 0-valent are univalent, while bivalent indicates that v is temporarily unsettled and may need to wait for decisions of subsequent rounds. The local DAG of node i changes dynamically, so the status of each vertex may also change as i keeps delivering new vertices. However, 1-valent and 0-valent are incompatible, meaning that if some correct node decides vertex v 1-valent, no correct node may ever decides v 0-valent (but v can be bivalent temporarily). *If every vertex v in round r is decided univalent*, node i can *decide* round r with all the 1-valent vertices (due to Invariants 4.2 and 4.3 in §4.1).

Normal path. Chitu selects leaders by random coins (like Tusk) to survive malicious schedulers that can totally control message delivery. Leader vertices are used to (recursively) decide preceding rounds, if the fast path does not work.

In the same vein as DAG-Rider and Tusk, we say a leader vertex v in round r is *valid*, if (1) v is observed by at least $f+1$ vertices of round $r+1$, or (2) v is observed by the next valid leader vertex in round $r' = r+2k$ ($k \in \mathbb{N}^+$). Note that in Chitu leader vertices in odd (even) rounds only decide vertices in odd (even) rounds, i.e., odd and even rounds are separated. This is because two leader vertices in consecutive rounds, e.g., round r and $r+1$, may not have a path in the DAG. With a

valid leader vertex v in even (odd) round r, a node decides all the previous even (odd) rounds that have not be decided. Like Tusk, these rounds are decided recursively. Each round is decided either by v, or by v' that is made valid by v (i.e., observed by v). In each round, every vertex strongly observed by v (or v') is decided 1-valent while the others are decided 0-valent. Such a rule ensures that *every round is decided either by the fast path or by the same leader vertex*.

Finally, node i commits round r if (1) all the preceding rounds are committed; and (2) round r is decided. That is to say, each node must commit rounds sequentially, though odd and even rounds are decided independently. Once round r is committed, all the ancestors of the 1-valent vertices of round r are also committed indirectly, thanks to DAG structure.

Leader selection. When $n-f$ vertices in round $r+1$ are delivered, node i broadcasts its signature share (for threshold signatures) and later determines the leader vertex of round r. With the help of the leader vertex, node i tries to decide round $r-2$ and all the preceding odd (or even) rounds recursively. Finally, if round r and every preceding round $r' < r$ are decided, node i commits rounds sequentially till round r. Figure 6 depicts an example for Chitu commit rules.

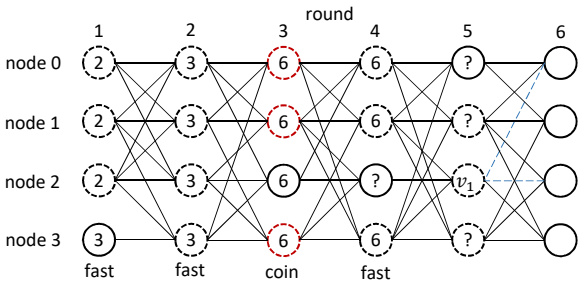


Figure 6: An example for commit rules of Chitu. Dotted circles denote strongly observed vertices in each round. The number in each circle denotes the round after which the corresponding vertex is committed. Rounds 1 and 2 are decided through the fast path, and thus their strongly observed vertices are committed immediately. In round 1, the vertex proposed by node 3 is committed indirectly by round 2, as it is observed by the strongly observed vertices of round 2. Since round 3 is decided through random coins, even if round 4 is decided through the fast path, it cannot be committed until the leader vertex v_1 of round 5 is selected, which decides round 3. In round 3, the vertex proposed by node 2 can also be committed because it is observed by a strongly observed vertex (node 0) in round 4.

4.4 Adaptive wait

Motivation. Consider the DAG construction stated in §4.2, each vertex is, on average, connected by $n-f$ vertices of the next round, which is exactly the 1-valent condition via the fast path. However, due to discrepancy in latency between nodes, some vertices may be connected by more than $n-f$ edges while others are not. Those connected by more than f but less than $n-f$ edges cannot be decided via the fast path. To mitigate this problem, intuitively more edges are needed to help those vertices become 1-valent, *by nodes trying to wait for more than $n-f$ vertices delivered to advance to the next round*. It is irrational to simply (re-)introduce a timeout mechanism (i.e. waiting Δ time) here, for a deterministic timeout is always an inevitable barricade on the fast path. It is also difficult to quantify Δ in unstable asynchronous networks. Besides, we must guarantee that such a wait will not prevent correct nodes from proceeding.

Adaptive mechanism. We observe that any vertex v received by $f+1$ nodes can be propagated to every correct node eventually (even with f faults). We say this kind of vertices are *pre-accepted*. In our adaptive wait mechanism, a node i keeps waiting for every pre-accepted vertex v to be delivered (through BRB), *so long as* the sender of v does not broadcast two distinct vertices in a round. Such an adaptive mechanism creates a gap between when a node i delivers $n-f$ vertices of round r and when i enters round $r+1$, effectively increasing the chance of connecting more vertices in round r. Besides, since i keeps receiving new messages while waiting, i may notice more pre-accepted vertices to wait for.

To guarantee that every pre-accepted vertex v can be delivered or an equivocation can be detected, node i needs to forward the VAL and $f+1$ PREPARE messages for v to all the other nodes while waiting. This forwarding scheme enables other (correct) nodes to vote for v or detect the malicious behavior of the sender of v. In the first case, all correct nodes eventually collect $n-f$ PREPARE messages for v; in the second case, all correct nodes stop waiting for v and enter the next round if no other pre-accepted vertex exists.

4.5 Performance analysis

We first analyze the performance of Chitu and Tusk (see Table 1). Chitu has a fast path that can quickly commit all vertices in 2 rounds. Specifically, when f nodes crash, Chitu always goes through the fast path. Through random coins, the probability to select a valid leader vertex is at least $\frac{f+1}{3f+1} > \frac{1}{3}$. Thus, in expectation, there exists a valid leader vertex in every three odd (even) rounds. Chitu can commit a round r only if both odd and even rounds after r select a valid leader vertex. Therefore, Chitu commits a round in every six rounds through random coins. Tusk has the same expected time complexity (through random coins), though it selects a leader vertex in every odd round. Chitu introduces an all-to-all message pattern in its Byzantine Reliable Broadcast (BRB), and thus has $O(n^2)$ message complexity and 2 message delays per round. As for communication complexity, each vertex v in round $r+1$ must connect to at least $2f+1$ vertices in round r, each of which contains $2f+1$ signatures. The complexity of each VAL messages is $O(n^2)$. Every node is allowed to propose

a new vertex in each round, thus the total communication complexity is $O(n^4)$ as the expected time complexity is $O(1)$. Tusk has the same communication complexity due to the same reason.

Table 1: Performance analysis of Chitu and Tusk. Note that the worst case indicates time complexity (message delays) in expectation.

	Best case	Normal case	Worst case	Message complexity	Communication complexity
Chitu	4	9	15	$O(n^3)$	$O(n^4)$
Tusk	14.5	14.5	25	$O(n^2)$	$O(n^4)$

Furthermore, Table 2 compares Chitu, MyTumbler and Red-Belly regarding their time complexities (message delays), as they all adopt the Fair-Fallback Framework.

Table 2: Performance analysis of the protocols following the Fair-Fallback framework.

	Best case	Worst case	Timing model
Chitu	4	$O(1)$	asynchrony
MyTumbler	4	$O(\log_2 n)$	asynchrony
Red Belly	4	-	partial synchrony

MyTumbler uses SuperMA, a multi-valued agreement protocol to commit proposals in three message delays in the best case, and another one message delay to pass timestamps for execution. If an adversary totally controls message scheduling, however, MyTumbler explicitly runs n asynchronous binary agreement instances, each within a SuperMA instance and has $O(1)$ time complexity in expectation, resulting in at least $O(\log_2 n)$ time complexity in total. Red Belly also commits proposals in four message delays in the best case, i.e., Propose, Echo, Ready and Decide. Red Belly further uses DBFT, a partially synchronous binary agreement protocol, to resolve ambiguity. DBFT selects a weak coordinator to suggest a value. It is not straightforward or fair to compare the worst-case performance between partially synchronous (Red Belly) and asynchronous (Chitu and MyTumbler) protocols as they target different timing models. In summary, the key advantages of Chitu over MyTumbler are the worst-case performance and simplicity (thanks to DAG). Unlike Red Belly, Chitu provides liveness under asynchrony and has no timers on the execution path.

4.6 Sketch of proof

Safety. First, BRB guarantees that any node can make at most one vertex delivered in a round and correct nodes will not deliver different vertices proposed by the same node. Then, since Chitu commits vertices round by round, we prove that in each round the same set of vertices are decided 1-valent, no matter via the normal path or the fast path. Assume nodes i and j decide $\mathcal{V}$ and $\mathcal{V}'$ in round r, respectively.

Case I: Both i and j decide round r via the normal path. W.l.o.g, assume i decides $\mathcal{V}$ by the leader vertex v_1 in round $r_1 \geq r+2$, while j decides $\mathcal{V}'$ by the leader vertex v_2 in round $r_2 \geq r_1+2$. v_1 is observed by $f+1$ vertices of round r_1+1, and v_2 observes at least $2f+1$ vertices in round r_1+1. Since $(f+1)+(2f+1) > 3f+1$, v_1 is observed by v_2. Recursively, j must decide round r by the leader vertex v_1 but not v_2.

Case II: i decides round r via the normal path, while j decides round r via the fast path. Assume i decides $\mathcal{V}$ by the leader vertex v_l in round $r' \geq r+2$, which observes at least $2f+1$ vertices in round $r+1$. For any $v \in \mathcal{V}'$, v is observed by at least $2f+1$ vertices in round $r+1$. Since these two groups of $2f+1$ vertices intersect at $f+1$ vertices, by Definition 4.2 v is strongly observed by v_l, i.e., $v \in \mathcal{V}$. In contrast, for any $v' \notin \mathcal{V}'$, v' is obeserved by less than $f+1$ vertices in round $r+1$, and thus v' cannot be strongly observed by v_l, i.e., $v' \notin \mathcal{V}$. Therefore, $\mathcal{V} = \mathcal{V}'$.

Case III: Both i and j decide round r via the fast path. Assume there exists a vertex $v \in \mathcal{V}$ but $v \notin \mathcal{V}'$. In i's view v is observed by at least $2f+1$ vertices in round $r+1$, while in j's view v is not observed by at least $2f+1$ vertices in round $r+1$. Since these two groups of $2f+1$ vertices intersect at $f+1$ vertices, at least one correct node proposes two different vertices in round $r+1$. A contradiction. For each vertex, any two correct nodes must decide it with the same univalent status, and thus all 1-valent vertices and their causal histories are committed in the same order.

Liveness. We first argue that for every round r, there are at least $f+1$ vertices strongly observed by the next valid leader vertex v_l in round $r+2k$ ($k \in \mathbb{N}^+$). At least $2f+1$ vertices in round $r+1$ are observed by v_l, denoted by $\mathcal{G}$. The total number of edges $\mathcal{G}$ provides to round r is at least $(2f+1)|\mathcal{G}|$. By Definition 4.2, a vertex in round r is strongly observed by v_l if observed by at least $f+1$ vertices in $\mathcal{G}$. In round r, if only f vertices are strongly observed by v_l and observed by all vertices in $\mathcal{G}$ using the most edges, and the rest $2f+1$ vertices are observed by only f vertices in $\mathcal{G}$, the total number of edges is $f|\mathcal{G}|+(2f+1)f \leq f|\mathcal{G}|+|\mathcal{G}|f = 2f|\mathcal{G}| < (2f+1)|\mathcal{G}|$. Thus, there are at least $f+1$ vertices strongly observed by v_l, i.e., satisfying the rule of valid leaders. Since the adversary cannot know the leader of round r until $2f+1$ vertices of round $r+1$ are delivered, it is impossible to control their edges to impede it becoming valid. Thus, the probability to select a valid leader is at least $\frac{f+1}{3f+1} > \frac{1}{3}$. As the protocol runs round by round, with probability 1 the leader of some round r is valid, and then all rounds $r-2k$ ($k \in \mathbb{N}^+$) are decided.

5 Performance Evaluation

We implement Chitu in Golang, using noise [7] for asynchronous networking. We use SHA256 to compute hash values and Ed25519 to sign and verify sig-

natures, implemented by Golang crypto library. Boneh-Lynn-Shacham (BLS) threshold signatures are used to construct random coins, computed by bls-eth-go-binary [2]. GoGo Protobuf [5] library is used for serialization. The code is available at https://github.com/Decentralized-Computing-Lab/ChituBFT.

We compare Chitu to Tusk [32] and BullShark [76], two representative DAG-based BFT protocols, with open-source implementations [3, 6]. Tusk and BullShark have the same structured DAG provided by Narwhal [32] but with different consensus protocols. Narwhal introduces additional worker nodes to disseminate blocks separately, leaving the primary nodes only running consensus protocol.

We run experiments on AWS EC2, with t3.2xlarge instances with 8 vCPUs, 32 GiB RAM and 5 Gbps Network burst bandwidth, running Ubuntu 22.04.4 LTS. Nodes and clients are deployed over 5 different regions across the globe: Ohio, Singapore, Tokyo, Canada (Central) and Frankfurt. The average RTT among 5 regions is 135 ms. For each node, there is a client running in the same region and sending a given number of requests to it per 50 ms (i.e. client rate). Each node gathers requests into blocks and proposes them when a new vertex is generated. We measure end-to-end latency as the time elapsed from when the client sends its request till when the client receives the confirmation from the same node. In all experiments, we set request size to 1000 B, which represents the trend towards larger transactions for Blockchain applications such as Bitcoin [1] and Ethereum [4].

5.1 Fault-free performance

We first compare the performance of Chitu, Tusk and BullShark when there is no fault. We gradually increase the rate that clients send requests until the system is saturated. We run experiments at different scales ($n = 4$, $n = 10$ and $n = 100$). When $n = 4$, nodes and clients are deployed in Ohio, Singapore, Tokyo and Frankfurt.

Figures 7 to 9 show the results. The end-to-end latency of Chitu is around 500 ms when the system is not saturated. Compared to Tusk, Chitu achieves up to 82.5% (440 ms versus 2519 ms) and 82.2% (485 ms versus 2729 ms) reduction in latency when each client sends 100 requests for every 50 ms, when $n = 4$ and $n = 10$ respectively. This is due to three reasons. First, most vertices in Chitu are committed via the fast path at small scales, where in the best case only four communication phases (i.e. two DAG rounds) are taken and random coins are bypassed. Second, in the normal path, the probability to select a valid leader is higher in Chitu than in Tusk and BullShark, for more edges are connected between rounds thanks to the wait mechanism. Third, Chitu trades off message complexity for less communication phases per round. With the moderately large number of nodes, the latency of Chitu is still much lower than that of Tusk and BullShark, with around 50% vertices committed via the fast path when $n = 100$.

Limitation. The percentage of fast-path commits becomes lower as the scale grows. Assume the evolvement of each vertex is an independent event, and the probability of a vertex being connected by more than $2f$ or less than $f+1$ vertices (i.e., 1-valent or 0-valent) is p. As the number of nodes grows, the probability that a round goes through the fast path is thus p^n, impeding larger-scale deployments. A possible solution to mitigate this problem is to select a subset of nodes as proposers while leaving others as validators, a typical way permissionless blockchains adopt to achieve scalability [42, 49]. We however leave this direction to future work.

BullShark has lower latency than Tusk since BullShark preselects a leader in each odd round and can commit vertices within 3.5 DAG rounds, while Tusk has to go through random coins and selects a valid leader every six rounds in expectation. Chitu has the advantage over BullShark not only in taking fewer rounds to commit vertices via the fast path, but also the leaderless feature and the way of advancing rounds. Nodes in BullShark additionally wait for a predefined leader to become valid (or at most Δ time), which may take longer time when the proposer of such a vertex is slow.

Wait mechanism. To further analyze the effects of the wait mechanism, we compare the average and 95%ile latency, and the percentage of fast-path commits, of Chitu with and without the wait mechanism. Figure 10 shows the result when $n = 4$ at four request rates corresponding to the points highlighted in Figure 7. With the wait mechanism, almost all vertices are committed through the fast path when the system is not saturated, achieving 99.5% when the request rate is 100 requests per 50 ms. In contrast, without the wait mechanism, only 12% vertices are committed through the fast path at the same request rate, with 1.75x latency (772 ms vs. 440 ms). This is mainly because the normal path takes eight communication phases to commit vertices. As the request rate grows, each vertex contains more requests and takes more time to propagate, which directly increases latency. Without the wait mechanism, it becomes more difficult to propagate vertices proposed by slow nodes, and hence they can be decided 0-valent via the fast path more easily. Thus, the percentage of fast-path commits becomes higher under slightly heavier workloads, but still much lower than that with the wait mechanism.

The wait mechanism helps a node observe as many vertices as possible, except those propagated by very slow nodes. These vertices may be observed by other $f+1$ but less than $n-f$ nodes also due to the wait mechanism, leading to the failure of the fast path. Thus, with the wait mechanism, the percentage of fast-path commits becomes a little lower under the heavier workload (but still around 92.5%). Moreover, as the scale grows, the probability of achieving fast termination without the wait mechanism declines rapidly. When $n = 10$, almost no round goes through the fast path without the wait mechanism, even as the request rate increases, due to the similar limitation discussed above. Therefore, the wait mechanism

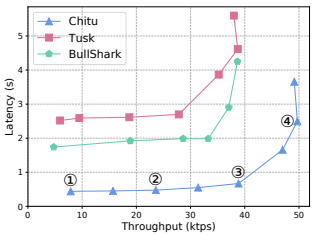


Figure 7: $n = 4$, fault-free.

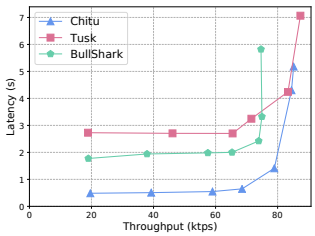


Figure 8: $n = 10$, fault-free.

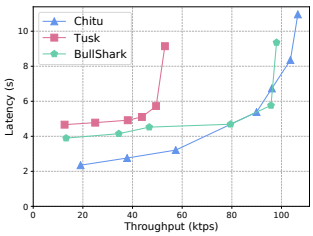


Figure 9: $n = 100$, fault-free.

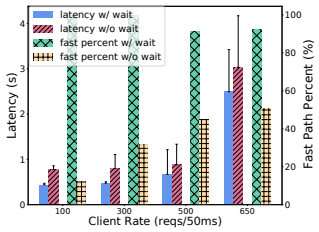


Figure 10: Performance with and without wait mechanism.

is critical for enabling the fast path.

5.2 Performance under crash faults

We then measure the performance of Chitu, Tusk and BullShark when disabling f nodes in advance, as there is no separation between normal case and view change in these protocols, i.e., Tusk, BullShark and Chitu have no failover to execute. The faulty nodes may crash or purposely keep silent. We run the experiments at different scales ($n = 4$, $n = 10$ and $n = 100$).

Figure 11 to 13 depict the results. Chitu has lower latency and higher peak throughput in this scenario. When f nodes crash, each vertex can observe only $n - f$ vertices of the last round. This implies that each vertex are observed by exactly $n - f$ vertices of the next round, and hence Chitu can completely skip random coins and terminate deterministically in this particular scenario, as no vertex is bivalent. The fast path takes four communication phases to determine that the rest f vertices (if exist) cannot be observed by more than f vertices, i.e., they must be 0-valent. However, in this scenario, each node needs to deliver vertices from all the others to advance to the next round, which may not be the fastest $n - f$ nodes in the fault-free scenario. Therefore, the latency of Chitu under the light workload slightly increases to around 550 ms at small scales. The peak throughput of all the protocols degrades in this case, as f faulty nodes cannot make proposals anymore. Nonetheless, it is noteworthy that the peak throughput of Chitu decreases by the least and is higher than the other two. This is because the wait mechanism cannot be triggered when there are only $n - f$ vertices in each round, and there is no need to forward the votes for pre-accepted vertices.

Compared to Chitu, Tusk and BullShark suffer a larger increase in latency under crash faults. Tusk always relies on random coins to terminate. With f faults, the selected leader is less likely to be valid, for with probability $\frac{1}{3}$ it is faulty. As for BullShark, such a high-latency problem even gets worse. BullShark requires each node to additionally wait Δ time, during which the pre-selected leader can be delivered. However, if the proposer is faulty (with probability $\frac{1}{3}$), such a timeout is located on the critical path. The default value of Δ is set to 5 sec in BullShark's open-source implementation. The average latency of BullShark increases to more than 5 sec at all scales. By this experiment we show the benefit of offloading the fallback mechanism when correct nodes can reach agreement leaderlessly.

5.3 Performance under Byzantine faults

Byzantine nodes may badly affect the performance of Chitu, especially by purposely disabling the fast path. Since it is difficult to manipulate message scheduling in practice, we evaluate Chitu in a typical scenario: (1) the fast path is disabled; (2) the wait mechanism is maliciously exploited by the adversary, so that each node has to deliver n vertices to advance to the next round; and, (3) Byzantine nodes disseminate their proposals with no payload and do not participate in message exchange of any proposal.

We run the experiment with $n = 10$, where one node in Canada Central and two nodes in Frankfurt are Byzantine faulty. Figure 15 depicts the results. When the system is not saturated (Figure 15a), the average latency under Byzantine faults is about 2x more than that under crash faults. This is mainly because the normal path takes 2x more communication phases than the fast path. The peak throughput (Figure 15b) moderately degrades under Byzantine faults compared to the one with f crash nodes. This is because it takes more bandwidth overhead for correct nodes to deliver the vertices proposed by Byzantine nodes, i.e. more PREPARE messages broadcast and forwarded. However, the wait mechanism also plays a role in synchronization, making slow nodes catch up with the pace of fast nodes. Therefore, the measured degradation of the peak throughput is acceptable, decreased by 5.78% (from 65.7 ktps to 61.9 ktps).

Note that one Byzantine node is sufficient to disable the fast path if the node can make its proposal bivalent (i.e., observed by more than f but less than $n - f$ vertices of the next round). In reality, initiating such an attack is a bit tricky because the Byzantine node must send its proposal at an appropriate time. Consider the all-to-all pattern of the Chitu broadcast, its adaptive wait mechanism, and even gossip-like communication in large-scale blockchains, the attack is feasible but not trivial. Nonetheless, even if the fast path is turned off, Chitu still has the fallback that provides acceptable performance.

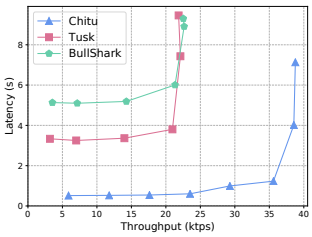


Figure 11: $n = 4$, 1 fault.

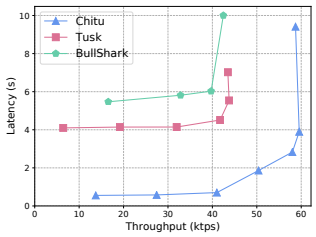


Figure 12: $n = 10$, 3 faults.

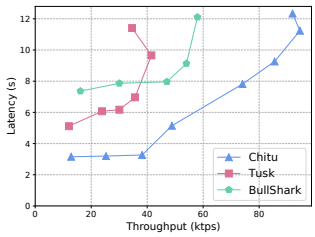


Figure 13: $n = 100$, 33 faults.

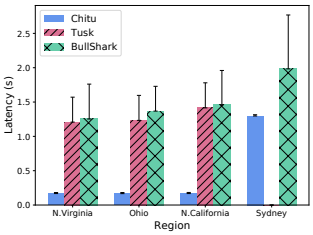
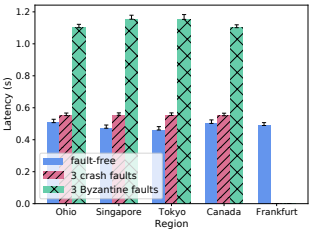
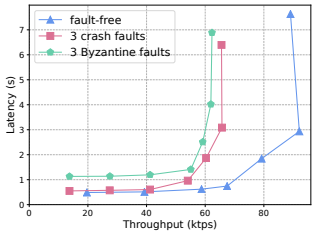


Figure 14: Performance with skewed distribution.



(a) latency on different regions



(b) latency vs. throughput

Figure 15: Performance under Byzantine faults ($n = 10$).

5.4 Performance with skewed distribution

Finally, we measure the performance of Chitu, Tusk and BullShark with more skewed node distribution. We run the experiment with $n = 4$ nodes in N.Virginia, Ohio, N.California and Sydney respectively. Three of them are in the US, with lower round-trip latency among one another, while Sydney is far away from them. We set the request rate to 50 requests per 50 ms (not saturated) and measure the average and 95%ile latency of each region.

The results are presented in Figure 14. In this particular setting, three nodes in the US are capable of pushing forward DAG construction, while the node in Sydney fails to keep pace with others. Even if Chitu has the wait mechanism, almost no vertices from Sydney are observed by those in the US, as the physical distance brings too much delay. Similarly, Tusk can move on with only three nodes in the US and hence has no data in Sydney. In contrast, BullShark and Chitu further leverage weak edges, which can connect to "outdated" vertices of previous rounds, though with a significantly increased latency.

The latency of Chitu is still lower than that of Tusk and BullShark in all regions. The reason is similar to the case when f nodes crash, where the fast path is enabled in Chitu, but proposals from the node in Sydney are committed with the help of weak edges. Tusk also outperforms BullShark in N. Virginia, Ohio and N. California. This is because BullShark periodically selects the node in Sydney as a leader, which delays the progress of the whole system. This result further demonstrates the effectiveness of the fast path in Chitu, as it relies on no special role.

6 Related Work

Partially synchronous protocols. A big family of BFT protocols are designed under partial asynchrony, where additional timing assumptions are used for electing a special role [17, 22, 27, 44, 56, 76, 80]. We argue that they prematurely rely on leader election and timing assumptions to resolve potential divergences among nodes, which are not necessary when correct nodes can quickly reach consensus in a leaderless manner. Our framework can also be applied to partially synchronous protocols, e.g., integrating the fast path with Bullshark [76] and treating leader election as a fallback mechanism.

Classical asynchronous protocols. Asynchronous protocols ensure safety and liveness properties despite network asynchrony. Asynchronous protocols can be traced back to the 1980s, when the seminal works of Rabin [72] and Ben-Or [13] were proposed. HoneyBadgerBFT [66] is considered the first practical solution, which makes use of a reliable broadcast protocol [20] and a randomized binary agreement protocol [68] to construct a common subset among all correct nodes. Since then, numerous variants [36, 46, 47, 79] are proposed, which greatly improve the performance of asynchronous protocols. However, they still put randomization on execution paths, and thus only achieve probabilistic termination. One exception is Bolt [63], a mechanism that explicitly switches between a fast pah (i.e., Fastlane in the original paper) and a pessimistic path. Bolt however reintroduces timer to trigger switching to pessimistic path. A similar idea of fast termination in Chitu can be applied to classical asynchronous protocols, which we leave for future work.

Certified DAGs. In contrast to classical protocols, DAG-based variants [11, 32, 53, 76] embed consensus into DAG construction, which not only reduces the number of messages needed, but greatly simplifies the design and implementation. Chitu, by leveraging the similar structure, also possesses the simplicity feature. In contrast to DAG-Rider [53] and Tusk [32], Chitu has the fast path that can bypass random coins. Autobahn [43] is a recently proposed DAG protocol

that incorporates a partially synchronous consensus mechanism into an asynchronous DAG construction layer, such that Autobahn can both reduce latency (compared to existing DAG protocols) and maintain the seamlessness feature of DAG protocols. Autobahn still relies on a correct leader to quickly reach consensus, while Chitu goes through the fast path in a pure leaderless manner. Shoal [75] and Shoal++ [9] leverage a reputation mechanism to select fast and stable nodes as anchors (leaders), in order to eliminate timeouts in most cases. Shoal++ further introduces a fast commit rule to quickly commit anchors. Sailfish [74] allows to select leader vertices in every round (rather than once every two or more rounds), thus reducing end-to-end latency.

Uncertified DAGs. Best Effort Broadcast (or uncertified DAG) helps reduce the number of message exchanges per round compared to certified DAG, but complicates fault handling logic as malicious nodes may send distinct proposals to different nodes. BBCA-CHAIN [64] leverages a variant of Byzantine Consistent Broadcast (BCB) named BBCA to broadcast only leader blocks, while non-leader blocks can be propagated by Best Effort Broadcast. Mahi-Mahi [51] manually configures the number of leader slots for each round and selects leaders using a global coin (randomization). Mysticeti [10] is a partially synchronous protocol that achieves three message delays in the best case. Mysticeti and Chitu share similarities in distinguishing between proposals that are observed or supported by $2f+1$ subsequent proposals and that are not observed by $2f+1$ proposals, but Mysticeti still waits timeout for a primary block of each round in order to provide liveness. In contrast to uncertified DAGs, Chitu follows the paradigm of certified DAG (like Tusk and Bullshark) and has no special role in its fast path but treats every proposal equally. We envision that by integrating the idea of Chitu into an uncertified DAG, we may further reduce commit delays for both paths.

Fast path. In recent years, numerous BFT variants [10, 18, 31, 74, 75] are proposed with a fast path. Most of them rely on a leader or some special role to coordinate consensus. Their fast path inevitably assumes the correctness of the single leader and puts Δ on the critical path, thus incurring additional waiting time if such a role crashes or be partitioned. Besides, more message delays may be introduced if they resort to a new leader to take over and move forward. The framework we study in this paper, in contrast, can quickly skip faulty or slow nodes in a responsive manner. There are three exceptions, which together motivate our study: Red Belly [30] under partial synchrony model, and HashGraph [11] and MyTumbler [62] under asynchrony model. Hashgraph uses a gossip protocol to disseminate proposals and derives consensus results from a DAG of degree two. Although the general idea of the fast path of Chitu shares similarities with Hashgraph, it may incur exponential latency [32] when some nodes are Byzantine. Therefore, integrating a fallback mechanism is critical to mitigating adverse effects on performance during an attack. Red Belly still puts timers on the fast path for aborted proposals, while Chitu can quickly terminates even if some proposals are 0-valent. MyTumbler sequences proposals based on physical timestamps and bypasses random coins when correct nodes input the same value into a binary agreement protocol. Our idea also shares similarity with MyTumbler in skipping random coins if correct nodes are convergent on a proposal, but MyTumbler relies on a binary agreement protocol to reach consensus. In the worst case, however, the time complexity of MyTumbler grows to at least $O(\log n)$ [14, 47]. Dumbo-NG [41] is an asynchronous protocol that concurrently runs transaction dissemination and asynchronous agreement, mitigating the tension between throughput and latency.

Other related work. Some BFT protocols [12, 33, 52, 59] make use of trusted hardware to prevent equivocation, so as to ensure both safety and liveness with only $n = 2f+1$ nodes. Such a model can further simplify the design of Chitu. Although we only gave the solution in permissioned settings, Chitu would be a promising approach also for permissionless blockchains [23, 39, 42, 69] when integrating with a Sybil [35] resistance measure such as Proof-of-Work [50] or Proof-of-Stake [54]. As Chitu resorts to all-to-all communication to disseminate and collect votes (PREPARE messages), its communication complexity can be further reduced by using a gossip protocol [15, 71] or threshold signatures [55].

7 Conclusion

In this paper we discussed a generic framework for reducing latency in BFT consensus protocols. We then presented Chitu, a DAG-based protocol following this framework. Chitu not only retains the simplicity feature of DAG protocols, but can also bypass expensive random coins and achieve consensus efficiently, when most nodes are in consistency about their observations. We implemented Chitu and conducted extensive experiments on Amazon EC2 platform. Through evaluation we showed that Chitu can effectively reduce latency in several typical scenarios, compared to existing DAG protocols. Our framework opens avenues for future work. Other partially synchronous or asynchronous protocols, or protocols under crash fault model, can also adopt a similar idea to achieve fast termination.

Acknowledgments

We are very grateful to our shepherd, Zihao Zhang, and the anonymous reviewers for their insightful comments and guidance. This work was supported by the National Natural Science Foundation of China (grant no. 62372293) and the Shanghai Action Plan for Science, Technology and Innovation (grant no. 24BC3201300).

References

[1] Bitcoin transaction size. https://bitcoinvisuals.com/chain-tx-size.

[2] bls-eth-go-binary. https://github.com/herumi/bls-eth-go-binary.

[3] Bullshark source code. https://github.com/MystenLabs/narwhal.

[4] Ethereum statistics. https://ycharts.com/indicators/sources/etherscan.

[5] gogoprotobuf. https://github.com/gogo.

[6] Narwhal and tusk. https://github.com/facebookresearch/narwhal.

[7] noise package. https://github.com/perlin-network/noise.

[8] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, Srinivasan Muralidharan, Chet Murthy, Binh Nguyen, Manish Sethi, Gari Singh, Keith Smith, Alessandro Sorniotti, Chrysoula Stathakopoulou, Marko Vukolić, Sharon Weed Cocco, and Jason Yellick. Hyperledger fabric: A distributed operating system for permissioned blockchains. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, New York, NY, USA, 2018. Association for Computing Machinery.

[9] Balaji Arun, Zekun Li, Florian Suri-Payer, Sourav Das, and Alexander Spiegelman. Shoal++: High throughput dag bft can be fast!, 2025.

[10] Kushal Babel, Andrey Chursin, George Danezis, Anastasios Kichidis, Lefteris Kokoris-Kogias, Arun Koshy, Alberto Sonnino, and Mingwei Tian. Mysticeti: Reaching the limits of latency with uncertified dags, 2024.

[11] Leemon Baird. The swirlds hashgraph consensus algorithm: Fair, fast, byzantine fault tolerance. *Swirlds, Inc. Technical Report SWIRLDS-TR-2016*, 1, 2016.

[12] Johannes Behl, Tobias Distler, and Rüdiger Kapitza. Hybrids on steroids: Sgx-based high performance bft. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, page 222–237, New York, NY, USA, 2017. Association for Computing Machinery.

[13] Michael Ben-Or. Another advantage of free choice (extended abstract): Completely asynchronous agreement protocols. In *Proceedings of the Second Annual ACM Symposium on Principles of Distributed Computing*, PODC '83, page 27–30, New York, NY, USA, 1983. Association for Computing Machinery.

[14] Michael Ben-Or and Ran El-Yaniv. Resilient-optimal interactive consistency in constant time. *Distributed Computing*, 16(4):249–262, 2003.

[15] N. Berendea, H. Mercier, E. Onica, and E. Riviere. Fair and efficient gossip in hyperledger fabric. In *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*, pages 190–200, Los Alamitos, CA, USA, dec 2020. IEEE Computer Society.

[16] Alysson Bessani, Eduardo Alchieri, João Sousa, André Oliveira, and Fernando Pedone. From byzantine replication to blockchain: Consensus is only the beginning. In *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 424–436, 2020.

[17] Alysson Bessani, João Sousa, and Eduardo E.P. Alchieri. State machine replication for the masses with bft-smart. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 355–362, 2014.

[18] Erica Blum, Jonathan Katz, Julian Loss, Kartik Nayak, and Simon Ochsenreither. Abraxas: Throughput-efficient hybrid asynchronous consensus. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, CCS '23, page 519–533, New York, NY, USA, 2023. Association for Computing Machinery.

[19] Dan Boneh, Ben Lynn, and Hovav Shacham. Short signatures from the weil pairing. In Colin Boyd, editor, *Advances in Cryptology — ASIACRYPT 2001*, pages 514–532, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[20] Gabriel Bracha. Asynchronous byzantine agreement protocols. *Inf. Comput.*, 75(2):130–143, nov 1987.

[21] Gabriel Bracha and Sam Toueg. Asynchronous consensus and broadcast protocols. *J. ACM*, 32(4):824–840, October 1985.

[22] Ethan Buchman, Jae Kwon, and Zarko Milosevic. The latest gossip on BFT consensus. *CoRR*, abs/1807.04938, 2018.

[23] Vitalik Buterin et al. A next-generation smart contract and decentralized application platform. *white paper*, 3(37), 2014.

[24] Christian Cachin, Rachid Guerraoui, and Luís Rodrigues. *Introduction to reliable and secure distributed programming*. Springer Science & Business Media, 2011.

[25] Christian Cachin, Klaus Kursawe, and Victor Shoup. Random oracles in constantinople: Practical asynchronous byzantine agreement using cryptography. *Journal of Cryptology*, 18(3):219–246, 2005.

[26] Ran Canetti and Tal Rabin. Fast asynchronous byzantine agreement with optimal resilience. In *Proceedings of the Twenty-Fifth Annual ACM Symposium on Theory of Computing*, STOC '93, page 42–51, New York, NY, USA, 1993. Association for Computing Machinery.

[27] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Trans. Comput. Syst.*, 20(4):398–461, 2002.

[28] Pierre Civit, Muhammad Ayaz Dzulfikar, Seth Gilbert, Vincent Gramoli, Rachid Guerraoui, Jovan Komatovic, and Manuel José Ribeiro Vidigueira. Byzantine consensus is $\theta(n^2)$: The dolev-reischuk bound is tight even in partial synchrony! Number 11, pages 1:11–1:19, Hannover, 2022. Dagstuhl Publishing.

[29] T. Crain, V. Gramoli, M. Larrea, and M. Raynal. Dbft: Efficient leaderless byzantine consensus and its application to blockchains. In *2018 IEEE 17th International Symposium on Network Computing and Applications (NCA)*, pages 1–8, 2018.

[30] T. Crain, C. Natoli, and V. Gramoli. Red belly: A secure, fair and scalable open blockchain. In *2021 2021 IEEE Symposium on Security and Privacy (SP)*, pages 1501–1518, Los Alamitos, CA, USA, may 2021. IEEE Computer Society.

[31] Xiaohai Dai, Bolin Zhang, Hai Jin, and Ling Ren. Parbft: Faster asynchronous bft consensus with a parallel optimistic path. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, CCS '23, page 504–518, New York, NY, USA, 2023. Association for Computing Machinery.

[32] George Danezis, Lefteris Kokoris-Kogias, Alberto Sonnino, and Alexander Spiegelman. Narwhal and tusk: A dag-based mempool and efficient bft consensus. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, page 34–50, New York, NY, USA, 2022. Association for Computing Machinery.

[33] Jérémie Decouchant, David Kozhaya, Vincent Rahli, and Jiangshan Yu. Damysus: Streamlined bft consensus leveraging trusted components. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, page 1–16, New York, NY, USA, 2022. Association for Computing Machinery.

[34] D. Dolev and H. R. Strong. Authenticated algorithms for byzantine agreement. *SIAM Journal on Computing*, 12(4):656–666, 1983.

[35] John R. Douceur. The sybil attack. In Peter Druschel, Frans Kaashoek, and Antony Rowstron, editors, *Peer-to-Peer Systems*, pages 251–260, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[36] Sisi Duan, Michael K. Reiter, and Haibin Zhang. Beat: Asynchronous bft made practical. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 2028–2041, New York, NY, USA, 2018. Association for Computing Machinery.

[37] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323, April 1988.

[38] Vitor Enes, Carlos Baquero, Tuanir França Rezende, Alexey Gotsman, Matthieu Perrin, and Pierre Sutra. State-machine replication for planet-scale systems. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery.

[39] Ittay Eyal, Adem Efe Gencer, Emin Gun Sirer, and Robbert Van Renesse. Bitcoin-ng: A scalable blockchain protocol. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 45–59, Santa Clara, CA, March 2016. USENIX Association.

[40] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, April 1985.

[41] Yingzi Gao, Yuan Lu, Zhenliang Lu, Qiang Tang, Jing Xu, and Zhenfeng Zhang. Dumbo-ng: Fast asynchronous bft consensus with throughput-oblivious latency. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, CCS '22, page 1187–1201, New York, NY, USA, 2022. Association for Computing Machinery.

[42] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling byzantine agreements for cryptocurrencies. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 51–68, New York, NY, USA, 2017. Association for Computing Machinery.

[43] Neil Giridharan, Florian Suri-Payer, Ittai Abraham, Lorenzo Alvisi, and Natacha Crooks. Autobahn: Seamless high speed bft. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, SOSP '24, page 1–23, New York, NY, USA, 2024. Association for Computing Machinery.

[44] G. Golan Gueta, I. Abraham, S. Grossman, D. Malkhi, B. Pinkas, M. Reiter, D. Seredinschi, O. Tamir, and A. Tomescu. Sbft: A scalable and decentralized trust infrastructure. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 568–580, 2019.

[45] Rachid Guerraoui, Nikola Knežević, Vivien Quéma, and Marko Vukolić. The next 700 bft protocols. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, page 363–376, New York, NY, USA, 2010. Association for Computing Machinery.

[46] Bingyong Guo, Yuan Lu, Zhenliang Lu, Qiang Tang, Jing Xu, and Zhenfeng Zhang. Speeding dumbo: Pushing asynchronous BFT closer to practice. *IACR Cryptol. ePrint Arch.*, page 27, 2022.

[47] Bingyong Guo, Zhenliang Lu, Qiang Tang, Jing Xu, and Zhenfeng Zhang. Dumbo: Faster asynchronous bft protocols. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, CCS '20, page 803–818, New York, NY, USA, 2020. Association for Computing Machinery.

[48] Suyash Gupta, Sajjad Rahnama, Jelle Hellings, and Mohammad Sadoghi. Resilientdb: Global scale resilient blockchain fabric. *Proc. VLDB Endow.*, 13(6):868–883, feb 2020.

[49] Timo Hanke, Mahnush Movahedi, and Dominic Williams. DFINITY technology overview series, consensus system. *CoRR*, abs/1805.04548, 2018.

[50] Markus Jakobsson and Ari Juels. *Proofs of Work and Bread Pudding Protocols(Extended Abstract)*, pages 258–272. Springer US, Boston, MA, 1999.

[51] Philipp Jovanovic, Lefteris Kokoris Kogias, Bryan Kumara, Alberto Sonnino, Pasindu Tennage, and Igor Zablotchi. Mahi-mahi: Low-latency asynchronous bft dag-based consensus, 2024.

[52] Rüdiger Kapitza, Johannes Behl, Christian Cachin, Tobias Distler, Simon Kuhnle, Seyed Vahid Mohammadi, Wolfgang Schröder-Preikschat, and Klaus Stengel. Cheapbft: Resource-efficient byzantine fault tolerance. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, page 295–308, New York, NY, USA, 2012. Association for Computing Machinery.

[53] Idit Keidar, Eleftherios Kokoris-Kogias, Oded Naor, and Alexander Spiegelman. All you need is dag. In *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing*, PODC'21, page 165–175, New York, NY, USA, 2021. Association for Computing Machinery.

[54] Sunny King and Scott Nadal. Ppcoin: Peer-to-peer crypto-currency with proof-of-stake. *self-published paper, August*, 19(1), 2012.

[55] Eleftherios Kokoris Kogias, Dahlia Malkhi, and Alexander Spiegelman. Asynchronous distributed key generation for computationally-secure randomness, consensus, and threshold signatures. New York, NY, USA, 2020. Association for Computing Machinery.

[56] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: speculative Byzantine fault tolerance. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*. ACM, 2007.

[57] Leslie Lamport. Generalized consensus and paxos. Technical Report MSR-TR-2005-33, March 2005.

[58] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, July 1982.

[59] Dave Levin, John R. Douceur, Jacob R. Lorch, and Thomas Moscibroda. TrInc: Small trusted hardware for large distributed systems. In *6th USENIX Symposium on Networked Systems Design and Implementation (NSDI 09)*, Boston, MA, April 2009. USENIX Association.

[60] Chenxing Li, Peilun Li, Dong Zhou, Zhe Yang, Ming Wu, Guang Yang, Wei Xu, Fan Long, and Andrew Chi-Chih Yao. A decentralized blockchain with high throughput and fast confirmation. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 515–528. USENIX Association, July 2020.

[61] Shengyun Liu, Paolo Viotti, Christian Cachin, Vivien Quéma, and Marko Vukolic. Xft: Practical fault tolerance beyond crashes. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, page 485–500, USA, 2016. USENIX Association.

[62] Shengyun Liu, Wenbo Xu, Chen Shan, Xiaofeng Yan, Tianjing Xu, Bo Wang, Lei Fan, Fuxi Deng, Ying Yan, and Hui Zhang. Flexible advancement in asynchronous bft consensus. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 264–280, New York, NY, USA, 2023. Association for Computing Machinery.

[63] Yuan Lu, Zhenliang Lu, and Qiang Tang. Bolt-dumbo transformer: Asynchronous consensus as fast as the pipelined bft. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, CCS '22, page 2159–2173, New York, NY, USA, 2022. Association for Computing Machinery.

[64] Dahlia Malkhi, Chrysoula Stathakopoulou, and Maofan Yin. Bbca-chain: Low latency, high throughput bft consensus on a dag. In *Financial Cryptography and Data Security: 28th International Conference, FC 2024, Willemstad, Curaçao, March 4–8, 2024, Revised Selected Papers, Part I*, page 51–73, Berlin, Heidelberg, 2025. Springer-Verlag.

[65] Darya Melnyk and Roger Wattenhofer. Byzantine agreement with interval validity. In *2018 IEEE 37th Symposium on Reliable Distributed Systems (SRDS)*, pages 251–260, 2018.

[66] Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. The honey badger of bft protocols. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, page 31–42, New York, NY, USA, 2016. Association for Computing Machinery.

[67] Nenad Milošević, Daniel Cason, Zarko Milošević, and Fernando Pedone. How Robust Are Synchronous Consensus Protocols? In Silvia Bonomi, Letterio Galletta, Etienne Rivière, and Valerio Schiavoni, editors, *28th International Conference on Principles of Distributed Systems (OPODIS 2024)*, volume 324 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 20:1–20:25, Dagstuhl, Germany, 2025. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[68] Achour Mostefaoui, Hamouma Moumen, and Michel Raynal. Signature-free asynchronous byzantine consensus with t < n/3 and o(n2) messages. In *Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing*, PODC '14, page 2–9, New York, NY, USA, 2014. Association for Computing Machinery.

[69] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. Technical report, Manubot, 2019.

[70] Oded Naor, Mathieu Baudet, Dahlia Malkhi, and Alexander Spiegelman. Cogsworth: Byzantine view synchronization. *CoRR*, abs/1909.05204, 2019.

[71] Ray Neiheiser, Miguel Matos, and Luís Rodrigues. Kauri: Scalable bft consensus with pipelined tree-based dissemination and aggregation. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, page 35–48, New York, NY, USA, 2021. Association for Computing Machinery.

[72] Michael O. Rabin. Randomized byzantine generals. In *24th Annual Symposium on Foundations of Computer Science (sfcs 1983)*, pages 403–409, 1983.

[73] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, 22(4):299–319, December 1990.

[74] Nibesh Shrestha, Rohan Shrothrium, Aniket Kate, and Kartik Nayak. Sailfish: Towards improving the latency of dag-based bft. Cryptology ePrint Archive, Paper 2024/472, 2024. https://eprint.iacr.org/2024/472.

[75] Alexander Spiegelman, Balaji Arun, Rati Gelashvili, and Zekun Li. Shoal: Improving dag-bft latency and robustness. In *Financial Cryptography and Data Security: 28th International Conference, FC 2024, Willemstad, Curaçao, March 4–8, 2024, Revised Selected Papers, Part I*, page 92–109, Berlin, Heidelberg, 2025. Springer-Verlag.

[76] Alexander Spiegelman, Neil Giridharan, Alberto Sonnino, and Lefteris Kokoris-Kogias. Bullshark: Dag bft protocols made practical. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, CCS '22, page 2705–2718, New York, NY, USA, 2022. Association for Computing Machinery.

[77] Chrysoula Stathakopoulou, Matej Pavlovic, and Marko Vukolić. State machine replication scalability made simple. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, page 17–33, New York, NY, USA, 2022. Association for Computing Machinery.

[78] M. Vukolic. *Quorum Systems: With Applications to Storage and Consensus*. 2012.

[79] Lei Yang, Seo Jin Park, Mohammad Alizadeh, Sreeram Kannan, and David Tse. DispersedLedger: High-Throughput byzantine consensus on variable bandwidth networks. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 493–512, Renton, WA, April 2022. USENIX Association.

[80] Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan Gueta, and Ittai Abraham. Hotstuff: Bft consensus with linearity and responsiveness. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, PODC '19, page 347–356, New York, NY, USA, 2019. Association for Computing Machinery.

Algorithm 1: Data structure and utilities for node i.

Local variables:
struct vertex v:
$v.r$ - the round of v in the DAG
$v.s$ - the node that proposes v
$v.txs$ - a list of transactions
$v.edges$ - a set of at least $n-f$ vertices (of round $v.r-1$) linked by v
$DAG[*]$ - an array of sets of vertices with Quorum Certificate (QC)
$txsToPropose$ - a queue, to which node i enqueues valid transactions from clients

1: **function** path(u,v)
2: **return** $\exists$ a sequence of $k \in \mathbb{N}$ vertices $v_1, v_2, \cdots, v_k$ **s.t.** $v_1 = u, v_k = v$, and $\forall j \in [1 \dots k-1] : v_j \in v_{j+1}.edges$

3: **function** createVertex(r)
4: **wait until** $\neg\, txsToPropose$.empty()
5: $v.r \leftarrow r$
6: $v.s \leftarrow i$
7: $v.txs \leftarrow txsToPropose$.dequeue()
8: $v.edges \leftarrow DAG[r-1]$
9: **return** v

10: **function** getLeaderVertex(r)
11: $j \leftarrow$ chooseLeader(r) ▷ use a random coin
12: **if** $\exists v \in DAG[r]$ **s.t.** $v.s = j$ **then**
13: **return** v
14: **else return** $\perp$

Algorithm 2: Chitu DAG construction for node i.

Local variables: $round \leftarrow 1$

1: **procedure** propose(r)
2: $v \leftarrow$ createVertex(r) ▷ see Alg. 1 Line 3
3: **broadcast** $\langle \text{VAL}, v, r \rangle$

4: **upon receiving** $\langle \text{VAL}, v, r \rangle$ from node p of round r **or** $f+1$ $\langle \text{PREPARE}, v, r \rangle_{\sigma_*}$ for the first time **do**
5: **broadcast** $\langle \text{PREPARE}, v, r \rangle_{\sigma_i}$

6: **upon receiving** $n-f$ $\langle \text{PREPARE}, v, r \rangle_{\sigma_*}$ **do**
7: $DAG[r] \leftarrow DAG[r] \cup \{v\}$
8: **if** $|DAG[r]| = n-f$ **then**
9: advanceRound(r)

10: **procedure** advanceRound(r)
11: **if** $r \geq round$ **then**
12: $round \leftarrow r+1$
13: propose$(round)$

Algorithm 3: Chitu commit rules for node i.

Local variables: $decided[*] \leftarrow \{\}$

1: **procedure** tryFastPath(r)
2: $one \leftarrow \{v \in DAG[r] : |\{v' \in DAG[r+1] : \text{path}(v,v')\}| \geq n-f\}$
3: $zero \leftarrow \{v \in DAG[r] : |\{v' \in DAG[r+1] : \neg\text{path}(v,v')\}| \geq n-f\}$
4: **if** $|one| + |zero| = n$ **then**
5: $decided[r] \leftarrow one$

6: **procedure** tryLeaderPath(r)
7: $v \leftarrow$ getLeaderVertex(r) ▷ see Alg. 1 Line 10
8: **if** $v \neq \perp$ **and** $|\{v' \in DAG[r+1] : \text{path}(v,v')\}| \geq f+1$ **then** ▷ valid
9: decideLeaderPath(v,r)

10: **procedure** decideLeaderPath(v_l, r_l)
11: $r \leftarrow r_l - 2$
12: $decided[r] \leftarrow \{v \in DAG[r] : |\{v' \in DAG[r+1] : \text{path}(v,v') \wedge \text{path}(v', v_l)\}| \geq f+1\}$
13: $v \leftarrow$ getLeaderVertex(r)
14: **if** $v \neq \perp$ **and** path(v, v_l) **then**
15: decideLeaderPath(v,r)
16: **else**
17: decideLeaderPath(v_l, r)

A Chitu Pseudocode

A.1 Data Structure and Utilities

The data structures and basic utilities of Chitu are specified in Algorithm 1. Each vertex v represents a proposal broadcast by a single node. It contains basic information such as a round number r, a source which identifies the node who created v, a block of transactions, and, most importantly, a set of edges which refer to vertices of the last round. Each node maintains a local copy of the DAG according to messages it delivers. For each node i, we denote its local view of the DAG as $DAG[*]$. $DAG[r]$ for $r \in \mathbb{N}^+$ stores a set of vertices that nodes generated in round r and delivered by i. $txsToPropose$ is a queue that stores valid transactions from clients.

Function path(u,v) is used to check whether there is a path from vertex u to vertex v in the DAG (Line 1), while Function createVertex(r) is used to create a vertex with basic information (Line 3). Function getLeaderVertex(r) computes the selected leader of round r and returns the vertex if it is delivered (Line 10). Otherwise, it returns $\perp$. The leader of round r is determined by a random coin.

A.2 DAG Construction

In Chitu Reliable Broadcast protocol, a vertex goes through two communication phases. The pseudocode is given in Algorithm 2. We denote two types of messages by VAL and PREPARE. Node i first broadcasts a VAL message when starting a new round r, containing a new vertex v proposed by it (Line 1-3). Upon receiving VAL message or $f+1$ PREPARE messages for the same vertex v, node i checks the validity and broadcasts a signed PREPARE message (Line 4-5). Here, $f+1$ PREPARE messages guarantee that the corresponding VAL message is received by at least one correct node and hence will be received by i definitely. Upon receiving $n-f$ PREPARE messages for the same vertex v, v can be delivered, i.e. added into the local DAG (Line 6-7). Note that if node i receives v from node j but has not delivered some vertex v' connected by v, it can ask j for v' that must be in j's local DAG. Procedure advanceRound(r) is called after each of $n-f$ vertices in round r collects $n-f$ PREPARE messages, i.e., when $|DAG[r]| = n-f$ (Line 8-9).

A.3 Commit Rules

The pseudocode of commit rules for node i is given in Algorithm 3. Every time a new vertex is added in round $r+1$, Procedure tryFastPath(r) is called to check whether round r satisfies the rules for the fast path (Line 1-5). Moreover, i also tries to decide round $r-2$ and its preceding odd (or even) rounds the with the help of the leader vertex in round r(Line 6-9). When there exist $n-f$ vertices in round $r+1$, node i can

broadcast its signature share (for threshold signatures) and later determine the leader vertex of round r (Line 7). Procedure decideLeaderPath(v_l, r_l) recursively decides all the preceding odd (or even) rounds by leaders (Line 10-17). Finally, if round r and all rounds $r' < r$ are decided, node i commits rounds sequentially to round r.

B Chitu Correctness Proof

We formally prove the safety and liveness properties of Chitu.

B.1 Safety

We first give the agreement proof of the normal path in Chitu.

Lemma B.1. *If a leader vertex v in round r is valid, then any valid leader vertex v' in the future round $r+2k$ $(k \in \mathbb{N}^+)$ has a path to v.*

Proof. The leader vertex v is valid only if v is observed by at least $f+1$ vertices in round $r+1$. Since every vertex must connect to at least $2f+1$ vertices in the previous round, there are at least $2f+1$ vertices in round $r+1$ observed by v' in round $r+2$. Considering the vertices in round $r+1$ have an intersection by $(2f+1)+(f+1) > 3f+1$, v' must have a path via at least 1 vertex in round $r+1$ to v. By induction, any valid leader vertex v' in round $r+2k$ $(k \in \mathbb{N}^+)$ must have a path to v. □

Lemma B.2. *Let v_1 and v_2 be the valid leader vertices in round r and $r+2k$ $(k \in \mathbb{N}^+)$, respectively. If a correct node enters the decide phase by v_1 before v_2, then no other correct nodes enter the decide phase by v_1 after v_2.*

Proof. We prove this lemma by contradiction. Since v_1 is selected to decide before v_2, there is no directed path from v_1 to v_2 in the DAG. If some correct node enters the decide phase by v_1 after v_2, then there is no path from v_2 to v_1. However, by Lemma B.1, there must exist a path from one of them to the other. A contradiction. □

Theorem B.1 (Agreement of the normal path)**.** *For each vertex, any two correct nodes decide it to the same univalent status through the normal path.*

Proof. If a correct node enters the decide phase by some valid leader vertex v in round r, by Lemma B.1 all valid leader vertices in round $r-2k$ $(k \in \mathbb{N}^+)$ are learned. Then, with Lemma B.2, any two correct nodes enter the decide phase by all valid leader vertices in the same sequence, for odd rounds and even rounds respectively. Since it can be seen as pipelining to decide odd rounds and even rounds, the decide results on these two are independent. Therefore, any two correct nodes decide each round by the same leader vertex, and hence decide the same univalent status on each vertex in each round. □

Theorem B.1 guarantees the agreement of the normal path in Chitu. Then we prove the agreement of the fast path and the normal path.

Lemma B.3. *If a correct node considers a vertex v in round r 1-valent through the fast path, then no other correct nodes consider v 0-valent through the fast path, and vice versa.*

Proof. We prove this lemma by contradiction. a correct node considers v 1-valent through the fast path only if at least $2f+1$ vertices in round $r+1$ observe v. Assume v is considered 0-valent by some correct node through the fast path, then at least $2f+1$ vertices in round $r+1$ do not observe v. The number of vertices in round $r+1$ is $(2f+1)+(2f+1) > 3f+1$. A contradiction. □

Lemma B.4. *If a vertex v in round r is considered 1-valent through the fast path, then v is strongly observed by the next valid leader vertex v_l in round $r+2k$ $(k \in \mathbb{N}^+)$ through the normal path.*

Proof. We prove this lemma by contradiction. v is considered 1-valent only if at least $2f+1$ vertices in round $r+1$ observe v. Denote by S the set of vertices in round $r+1$ observed by v_l, $|S| \geq 2f+1$. Assuming v is not strongly observed by v_l, by Definition 4.2 there are less than $f+1$ vertices in S that observe v. Thus, at least $f+1$ vertices in S do not observe v. The number of vertices in round $r+1$ is $(2f+1)+(f+1) > 3f+1$. A contradiction. □

Lemma B.5. *If a vertex v in round r is considered 0-valent through the fast path, then v cannot be strongly observed by the next valid leader vertex v_l in round $r+2k$ $(k \in \mathbb{N}^+)$ through the normal path.*

Proof. We prove this lemma by contradiction. v is considered 0-valent only if at least $2f+1$ vertices in round $r+1$ do not observe v. Denote by S the set of $2f+1$ vertices in round $r+1$ observed by v_l. Assuming v is strongly observed by v_l, by Definition 4.2 there are at least $f+1$ vertices in S that observe v. The number of vertices in round $r+1$ is $(2f+1)+(f+1) > 3f+1$. A contradiction. □

We give the total agreement of Chitu through different commit paths:

Theorem B.2 (Agreement of different paths)**.** *For each vertex, any two correct nodes decide it to the same univalent status.*

Proof. Let i and j be two correct nodes. We prove this theorem in three cases.

Case I: Both i and j decide vertex v through the fast path. By Lemma B.3, v is decided to the same status by i and j.

Case II: i decides vertex v through the fast path while j decides v through the normal path. By Lemma B.4 and Lemma B.5, if v is considered 1-valent (0-valent) by i through the fast path, then v can (cannot) be strongly observed by the

next valid leader vertex in j's view. Thus, v can be decided to the same status as i by j through the normal path.

Case III: Both i and j decide vertex v through the normal path. By Theorem B.1, v is decided to the same status by i and j. □

Theorem B.3 (Total order)**.** *Any two correct nodes commit vertices with the same order.*

Proof. Chitu commits vertices round by round consecutively. By Theorem B.2, any two correct nodes have the same decide result on each vertex, and hence can commit all vertices that are decided 1-valent with the same pre-defined order in each round. In the meantime, the causal history of these vertices are also committed with the same order. Therefore, Chitu satisfies total order. □

B.2 Liveness

Lemma B.6. *For every round r, there are at least $f+1$ vertices strongly observed by the next valid leader vertex v_l in round $r+2k$ ($k \in \mathbb{N}^+$).*

Proof. A vertex must connect to at least $2f+1$ vertices in the previous round. Denote by S the set of vertices in round $r+1$ observed by v_l, $|S| \geq 2f+1$. The total number of edges S provide to round r is at least $(2f+1)|S|$. By Definition 4.2, a vertex in round r is strongly observed by v_l if it is observed by at least $f+1$ vertices in S. There are at most $3f+1$ vertices in round r. Therefore, if only f vertices in round r are strongly observed by v_l, which are observed by all vertices in S using the most edges, and the rest $2f+1$ vertices in round r are observed by only f vertices in S, the maximum total number of edges is $f|S|+(2f+1)f \leq f|S|+|S|f = 2f|S| < (2f+1)|S|$. Thus, there are at least $f+1$ vertices strongly observed by v_l. □

Lemma B.6 shows that for every round r there are at least $f+1$ vertices each observed by at least $f+1$ vertices in round $r+1$. This connection feature is the same as the rules of valid leaders, i.e., a leader vertex v_l in round r is valid if v_l is observed by at least $f+1$ vertices in round $r+1$. Then the following lemma is simply derived.

Lemma B.7. *For every round r, there are at least $f+1$ vertices that satisfy the rules of valid leaders.*

Theorem B.4 (Termination of leader selection)**.** *In expectation, Chitu selects a valid leader vertex for odd (even) rounds every 6 rounds in the DAG under an asynchronous adversary.*

Proof. To select a leader vertex in round r, the random coin is generated when $2f+1$ vertices in round $r+1$ are added to the DAG. Only after the DAG of the next round is fixed can the adversary identify the leader of the round before, so that it is impossible to manipulate the edges connected to the leader. Since by Lemma B.7 at least $f+1$ vertices in each round satisfy the rules of valid leaders and the coin is uniformly distributed, the probability to select a valid leader in each selection is at least $\frac{f+1}{3f+1} > \frac{1}{3}$. Thus, in expectation, there exists a valid leader every 3 selections. Since each selection takes 2 rounds, and Chitu selects leaders for odd rounds and even rounds respectively, in expectation, a valid leader for odd (even) rounds is selected every 6 rounds in the DAG. □

Lemma B.8. *If a leader vertex v in round r is valid, then any vertex in round $r-2k$ ($k \in \mathbb{N}^+$) is decided to a univalent status.*

Proof. By Lemma B.1, all valid leaders in round $r-2k$ ($k \in \mathbb{N}^+$) can be learned, and hence all vertices in round $r-2k$ ($k \in \mathbb{N}^+$) are decided to a univalent status. □

Theorem B.5 (Termination)**.** *In networks with random message delays, in expectation, Chitu decides each vertex to a univalent status within less than 5 rounds in the DAG.*

Proof. Let v_l be the leader vertex in round r selected through random coins. Since message delays are distributed uniformly, each vertex in round $r+1$ connects to at least $2f+1$ vertices in round r independently of other vertices in round $r+1$. Therefore, the probability that each vertex in round $r+1$ observes v_l is at least $\frac{2f+1}{3f+1} > \frac{2}{3}$. Considering v_l is valid if it is observed by at least $f+1$ vertices in round $r+1$, the probability that an selected leader can be valid is at least $\sum_{k=f+1}^{2f+1} \binom{2f+1}{k} \left(\frac{2}{3}\right)^k \left(\frac{1}{3}\right)^{2f+1-k}$. Since this probability increases as f grows, we can compute that the minimum probability is $\frac{12}{27} + \frac{8}{27} = 0.74$ when $f=1$. Thus, in expectation, an selected leader is valid every $\frac{1}{0.74} = 1.35$ selections, i.e. $1.35 \times 2 = 2.7$ rounds, at most. Then, by Lemma B.8 each vertex in odd (even) rounds can be decided by the nearest subsequent valid leader in odd (even) rounds. In expectation, each vertex is decided by a valid leader after $\frac{1\times 1+0.35\times 1.35}{1.35} \times 2 = 2.18$ rounds. Combining the above results, in expectation, Chitu decides each vertex to a univalent status within $2.7+2.18 = 4.88 \approx 5$ rounds (or less when $f>1$) in the DAG. Moreover, if the fast path is achieved, each vertex is decided every 2 rounds in the DAG, which reduces the expectation of rounds to decide each vertex further. □

Theorem B.6 (Validity)**.** *In each round, at least one vertex proposed by a correct node is decided 1-valent.*

Proof. By Lemma B.6, for each round, at least one vertex from a correct node can be strongly observed by the subsequent valid leader and decided 1-valent. Therefore, the adversary can not fully occupy the 1-valent vertices. Chitu satisfies validity. □

Fast Distributed Transactions for RDMA-based Disaggregated Memory

Haodi Lu, Haikun Liu*, Yujian Zhang, Zhuohui Duan, Xiaofei Liao, Hai Jin, Yu Zhang
National Engineering Research Center for Big Data Technology and System,
Service Computing Technology and System Lab/Cluster and Grid Computing Lab,
School of Computer Science and Technology, Huazhong University of Science and Technology, China
Email: {haodilu, hkliu, yj8023xx, zhduan, xfliao, hjin, zhyu}@hust.edu.cn

Abstract

Memory disaggregation has emerged as a promising datacenter architecture since it improves memory utilization and scalability. However, it is usually costly to process distributed transactions in disaggregated memory systems due to relatively high latency of remote memory accesses. In this paper, we present HDTX, a high-performance distributed transaction system for RDMA-based disaggregated memory. We advocate three novel designs. First, we propose a *fast commit protocol* (FCP) to minimize network round trips by coalescing different phases of distributed transaction processing. Second, we propose an RDMA-enabled offloading mechanism to reduce data transfers across computing and memory nodes by carefully orchestrating different RDMA primitives. Third, we propose decentralized priority-based locking to schedule mission-critical transactions, and thus further reduce the latency of distributed transactions. Experimental results show that HDTX reduces the latency of distributed transactions by up to 88.3% and 72.1%, and improves the throughput by up to 2.08× and 84.7%, compared with RDMA-based distributed transaction systems–FaRM and FORD, respectively.

1 Introduction

Resource disaggregation has emerged as one fundamental shift that we organize and manage different types of resources (e.g. CPU, memory) in datacenters. It has attracted extensive interests in both academia [42, 53, 54] and industry [9, 17] in recent years. Under resource disaggregation, monolithic servers are disaggregated into dedicated computing and memory nodes which are typically connected by high-speed networks such as *Remote Direct Memory Access* (RDMA) or *Compute Express Link* (CXL). The computing nodes run programs using a small local DRAM buffer backed with a large remote memory pool, while the memory nodes are usually equipped with very limited computing units for memory allocations and network initialization/connectivity. By pooling and sharing different resources, this resource disaggregation architecture substantially improves resource utilization, and facilitates resource scaling and failure isolation.

When correlated objects are concurrently accessed in the *disaggregated memory* (DM) pool, the computing nodes have to exploit *distributed transactions* (dtxns) to guarantee data integrity and consistency. However, most previous RDMA-based dtxn systems [28, 48] are designed for conventional monolithic servers in which computing and memory resources are tightly coupled. They are not applicable to DM systems because memory nodes usually have very limited computing resource to process costly operations during dtxn processing, such as buffer polling [12], locking [48], and data copying [28]. Recently, the state-of-the-art FORD [53] has been proposed particularly for processing dtxns in DM systems. However, it still incurs relatively high latency and high network bandwidth consumption. Moreover, dtxn processing systems should offer low latency for mission-critical applications. This low-latency requirement further increases the difficulty of deploying dtxn processing systems in the DM architecture due to the following challenges.

C.1: multiple network round trips (RTT) caused by multi-phase dtxn processing. Traditional dtxn systems usually exploit *optimistic concurrency control* (OCC) [46] to achieve lock-free dtxn processing on read-only data, and adopt the *primary-backup replication* (PBR) mechanism [28, 48] to guarantee high availability. Thus, existing dtxn systems [12, 49] generally process a single dtxn in five phases. Specifically, coordinators execute the dtxn in the *Execution* phase, serialize the dtxn in the *Locking* and *Validation* phases, and finally commit the dtxn in the *Commit Backup* and *Commit Primary* phases. Each phase usually incurs an RTT. FORD [53] has optimized the dtxn protocol particularly for DM systems, but still requires four phases to process a dtxn. These multiple phases often cause high latency of dtxn processing.

C.2: inefficient data synchronization in dtxn commit phases. Most previous dtxn processing systems require two phases to commit backups and primaries sequentially [12,

*Haikun Liu is the corresponding author.

49]. To reduce the latency, previous proposals [29] [34] rely on CPUs of memory nodes to synchronize the latest data within backups and primaries locally. However, these schemes are not applicable to DM systems since memory nodes do not have sufficient CPU resource to perform the costly data synchronization. Thus, they have to send the log and the latest data from computing nodes to all memory nodes in two rounds of data transfers. This causes substantial pressure on the RDMA bandwidth when the network load is high.

C.3: limited CPU resource in DM nodes for scheduling mission-critical dtxns. For mission-critical dtxns, their execution priority usually should be enhanced to guarantee low latency. Conventional dtxn systems [11, 35] exploit priority-based locking [22] to schedule different dtxns globally using CPUs on storage nodes. However, they cannot be directly adopted in DM systems because memory nodes usually do not have sufficient CPU resource for lock management and dtxn scheduling. Thus, it is challenging to schedule dtxns globally by DM nodes.

In this paper, we present HDTX, a high-performance distributed transaction system for the RDMA-based DM architecture. Through an in-depth analysis of the characteristics of dtxns and one-sided RDMA primitives, we propose three novel designs to address the above three challenges, respectively. Specifically, we make the following contributions.

For C.1, we propose a *fast commit protocol* (FCP) to minimize network round trips by coalescing the *Validation* and *Commit* phases of dtxns. To commit a dtxn earlier while still guaranteeing data consistency, FCP adopts the redo log and a visibility control technique [29, 34, 53] to coalesce three phases: *Validation*, *Commit Backup*, and *Commit Primary*. At first, FCP replaces *Commit Backup* and *Commit Primary* phases with one *Commit* phase and one asynchronous *Release* phase. Moreover, since there is no data dependency between the *Validation* phase and the *Commit* phase, FCP also coalesces these two phases to further reduce network round trips. Thus, the computing node only requires one round trip to commit a dtxn after the *Execution* and *Locking* phases.

For C.2, we propose RDMA-enabled data synchronization offloading to reduce RDMA bandwidth consumption. In the *Release* phase of our fast commit protocol, the computing node should write the latest data to the datastore, update the version of data, and release the lock on memory nodes. Since the redo log containing the latest data has been stored in memory nodes during the *Commit* phase, HDTX can offload these operations (i.e., data synchronization) to *RDMA network interface controller* (RNIC) via a series of `RDMA Write` and `RDMA Atomic` primitives. By orchestrating these primitives with special `RDMA Wait` and `RDMA Enable` primitives, the RNICs of memory nodes can perform the *Release* phase autonomously. In this way, HDTX can avoid additional transfers of the latest data across computing and memory nodes, mitigating the bandwidth contention of RDMA networks.

For C.3, we propose decentralized priority-based locking to reduce the latency of mission-critical applications. During the *Locking* phase, each computing node exploits the `RDMA Fetch-and-Add (FAA)` primitive to request write locks for a read-write set. For mission-critical dtxns, the computing node can set a high priority to the lock request so that it has more opportunities to acquire the required locks. In a global view, our decentralized priority-based locking mechanism can schedule latency-sensitive dtxns without involving memory nodes' CPUs.

We implement HDTX [23] and evaluate it with several benchmarks that simulate real-world workloads[1]. Compared with two typical RDMA-enabled dtxn systems–FaRM [12] and FORD [53], experimental results show that HDTX reduces the latency of dtxns by up to 88.3% and 72.1%, reduces the 99th percentile latency by up to 82.7% and 60.9%, and improves the dtxn throughput by up to 2.08× and 84.7% on average, respectively.

2 Background

2.1 High-Performance RDMA

RDMA networks are widely deployed in *high-performance computing* (HPC) environments and data centers because they offer high-throughput and low-latency remote memory accesses via bypassing traditional I/O stacks. Applications can leverage one-sided RDMA operations, such as `Read/Write` primitives, to directly access remote memory, without the involvement of remote CPUs. Also, applications can use two-sided RDMA operations, such as `Send/Recv` primitives, to communicate with remote CPUs for more complex operations. There are also a number of `RDMA Atomic` primitives, such as `RDMA Compare-and-Swap (CAS)` and `RDMA FAA`. The former can atomically modify 8-byte data under specific conditions, and the latter can atomically add 8-byte data to a specific value. To perform RDMA operations, an application must first construct a *completion queue* (CQ) and a *queue pair* (QP) which comprises a send queue and a receive queue. It then appends RDMA operations to the QP and notifies the RNIC for execution. Once the RDMA operation completes, the RNIC generates an acknowledgment message in the CQ to inform the application.

Besides data transmission, the RDMA driver also supports ordering control via `RDMA Fence` flags or `RDMA Wait` primitives, which can maintain the operation order within a send queue and across different QPs, respectively. Specifically, an RDMA operation with an `RDMA Fence` flag is halted on the sender's RNIC till all previous operations within the same send queue are completed. In contrast, the `RDMA Wait` can block its subsequent RDMA operations till a specific RDMA operation is completed. The `RDMA Enable` primitive allows the RNIC to fetch the metadata of pending RDMA operations

[1]The source code and experiments in this paper are available at Github [23].

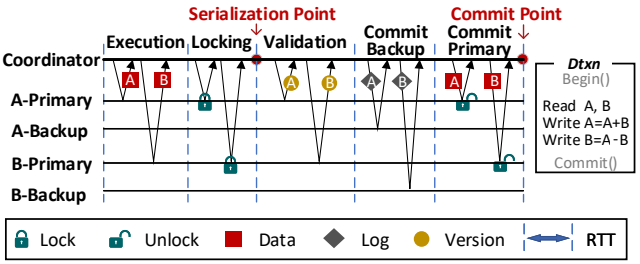


Figure 1: Dtxn processing based on OCC and PBR. Each data (A and B) has two replicas on memory nodes.

in the same work queue, and enables them to be executable. Thus, `RDMA Wait` and `RDMA Enable` can be orchestrated to perform a set of pending RDMA operations autonomously in an event-driven manner [31, 38].

2.2 RDMA-based Distributed Transactions

Distributed transactions are commonly used to ensure data consistency when multiple objects are accessed concurrently. There are two kinds of concurrency control mechanisms for dtxn systems: pessimistic and optimistic. The former locks all objects involved in a transaction to guarantee data consistency in high-contention scenarios, while the latter only locks the write set of a transaction to improve the parallelism of dtxn processing in low-contention scenarios.

Most RDMA-based dtxn systems [13, 28, 48] exploit both OCC and PBR to achieve high performance and high availability, respectively. Figure 1 illustrates a typical dtxn protocol. Upon receiving a dtxn request, the coordinator (i.e., the computing node) fetches the read set from multiple memory nodes and executes the transaction locally. After the *Execution* phase, the coordinator tries to commit the dtxn through four phases. If the read-write data are locked successfully in the *Locking* phase, and the coordinator verifies that the read set has not been modified by other dtxns before the *Validation* phase, it writes the dtxn output to logs on all backups for failure recovery. Once the coordinator receives all acknowledgments (ACKs), it updates and unlocks the data on primary nodes, and finally commits the dtxn. If the validation or any write operation fails, the coordinator should unlock the write set and abort the dtxn. Overall, five RTTs are required to commit a single dtxn in five phases.

3 Motivation

For most online services, dtxn processing systems should offer both high throughput and low latency. However, existing RDMA-based dtxns do not work efficiently in DM systems. By carefully analyzing existing dtxn protocols and ordering constraints of batched RDMA primitives, we find that there are several opportunities for optimizing dtxns in DM systems.

Table 1: Work request operation ordering

First Operation \ Second Operation	Send	Write	Read	Atomic (FAA/CAS)
Send	#	#	#	#
Write	#	#	#	#
Read	F	F	#	F
Atomic (FAA/CAS)	F	F	#	F

*F: Order is maintained only if the second operation has set a Fence flag.
*#: Order is always maintained.

Constraints of RDMA operation ordering. One-sided RDMA supports remote memory accesses without involving remote CPUs. It is applicable to DM systems but poses a number of constraints on dtxn processing. Two RDMA primitives that are batched and delivered from the same send queue should obey multiple operation ordering rules outlined in the *InfiniBand Trade Association* (IBTA) Specification [15,43], as shown in Table 1. Specifically, if the first RDMA operation is an `RDMA Send/Write` and the second RDMA operation can be any type, these two primitives are executed sequentially and thus can be completed within a single RTT. If the first operation is an `RDMA Read/FAA/CAS` and the second operation is not an `RDMA Read`, the sender's RNIC may execute the second operation earlier, causing unpredictable issues. Thus, an `RDMA Fence` flag should be attached to the second RDMA operation to guarantee the ordering but incurs additional network latency. Since RDMA operations for data synchronization among replicas should be ordered to guarantee data consistency, the dtxn processing protocol should be carefully designed to fully exploit different RDMA primitives.

Opportunities for reducing network round trips. Since typical dtxn systems require five RTTs to commit a transaction, as shown in Figure 1, previous studies [48, 53] have proposed several RDMA-based optimizations. The coordinator can combine the *Execution* and *Locking* phases, or the *Locking* and *Validation* phases to reduce network round trips. In addition, the coordinator can also update both primary and backup replicas in a single RTT, at the risk of suffering lower recovery performance upon failures. Despite these optimizations, previous proposals still require three or more RTTs to commit a dtxn. Fortunately, we find that it is possible to further reduce network round trips by carefully orchestrating the logging mechanism and the primary/backup *Commit* phases. By decoupling the data synchronization operation from the primary/backup *Commit* phases, we have an opportunity to coalesce the *Validation* and *Commit* phases, allowing a dtxn to be committed within two RTTs (Section 4.2).

Opportunities for RDMA-enabled data synchronization offloading. Most dtxn systems adopt undo or redo logs to guarantee data consistency in case of failures. In traditional dtxn systems [28,48], data nodes rely on their CPUs to update the data using redo logs, or to generate undo logs using the original data. However, these logging mechanisms are not applicable to DM systems due to lack of CPU resource on

DM nodes. The state-of-the-art dtxn system–FORD [53] uses undo logs for the DM, but its coordinator has to send the log and the latest data to all primaries/backups in two phases of dtxns. Since the coordinator should read the original data from memory nodes and then write it to the undo log, FORD incurs additional network bandwidth consumption due to logging. Although the *multi-version concurrency control* (MVCC) [37, 44] mechanism can avoid the additional data transfer incurred by logging, it requires more computing resource for garbage collection in memory nodes. By revisiting redo logs and `RDMA Wait/Enable` primitives, we find that it is possible to save the RDMA bandwidth by replacing remote memory copying with local memory copying. Since the redo log has been sent to memory nodes, we exploit an `RDMA Wait` and an `RDMA Write` to update the data by copying the redo log locally, avoiding remote memory copying and unnecessary RDMA bandwidth consumption (Section 4.3).

Opportunities for scheduling mission-critical dtxns. Most dtxn systems face a risk of high tail latency due to lock contention and deadlocks. Specifically, some RDMA-based dtxn systems typically acquire locks via `RDMA CAS` primitives with blind retries [12, 53], leading to unpredictable performance and starvation issues. Some proposals [36, 51] schedule tasks in a first-in-first-out manner, and thus also fail to accelerate mission-critical dtxns. Priority-based locking can mitigate these problems by scheduling latency-sensitive dtxns. However, existing priority-based locking mechanisms are inapplicable to the DM architecture because of limited computing resource on memory nodes. Fortunately, since the `RDMA FAA` primitive can perform a read-modify-write operation in a single RTT, it offers an opportunity to design a decentralized locking mechanism for flexible dtxn scheduling. By carefully designing the data structure of priority-based locks and the lock acquisition mechanism, mission-critical dtxns are given a high priority to acquire locks, and thus reduce lock contention with other dtxns. (Section 4.4).

4 Design of HDTX

4.1 Overview

Figure 2 shows the overview of HDTX. The coordinators in the computing pool are responsible for receiving and processing transactions requested by clients. The memory pool stores application data in hash tables. The coordinators (i.e., computing nodes) directly access the remote memory nodes through RDMA networks.

At first, HDTX initializes the datastore and establishes network connections with one CPU thread in the memory pool (①). Meanwhile, the computing pool caches the metadata and the configuration of the datastore for remote data accesses. After HDTX is initialized, coordinators in the computing pool are ready to serve clients' requests (②). Specifically, the coordinator initiates a transaction with `Tx_begin()`. Then, it uses

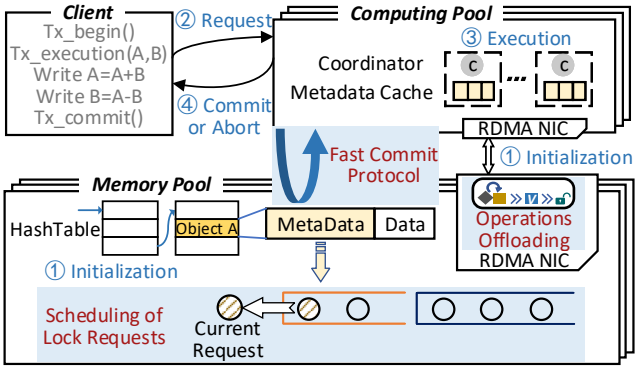


Figure 2: The system overview of HDTX

the `Tx_execution()` to fetch and lock the latest data from the memory pool with priority-based locking (Section 4.4), and executes the transaction locally (③). We note that the coordinator can perform the `Tx_execution()` multiple times to fetch and lock data that is newly-added to the read-write set during the dtxn execution. Once the execution is completed, the coordinator exploits our fast commit protocol (Section 4.2) to commit the transaction using `Tx_commit()`, and returns the transaction output to the client (④).

4.2 Fast Commit Protocol

In this section, we elaborate two key designs that enable fast dtxn commits while still guaranteeing data consistency.

4.2.1 Combining Backup and Primary Commit Phases

When the computing node commits a transaction and responds to the client, the latest data should be persisted in all memory nodes to guarantee consistency. As shown in Figure 1, traditional transaction systems require two RTTs to write the log and data sequentially to backups and primaries, respectively. This ensures fast failure recovery but incurs high latency. In traditional monolithic servers, a straightforward approach for fast dtxn commits can be achieved by combining *Backup Commit* and *Primary Commit* phases within a single RTT. Specifically, the coordinator writes logs to all primaries/backups, and then makes the write set invisible by setting the visibility bit as "0". This visibility control [29, 34, 53] can guarantee data consistency during the combined *Commit* phase. Once receiving ACKs from all primaries/backups, the coordinator can commit the dtxn, and then asynchronously executes a *Release* phase, which is responsible for updating the data and its version, making the write set visible, and releasing locks.

Unfortunately, the above approach incurs additional RTTs when applying to an RDMA-enabled DM architecture due to the ordering rules of RDMA primitives. In the following, we discuss how the primary/backup *Commit* and *Release* phases should be implemented in the DM architecture when using

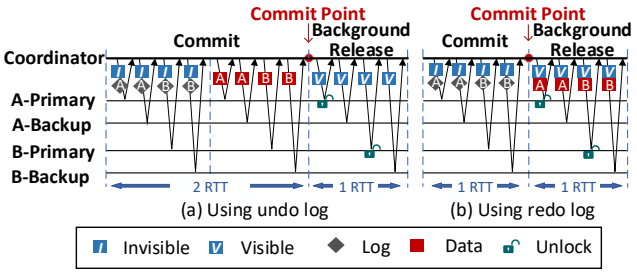


Figure 3: Commit a dtxn using (a) undo log and (b) redo log after the *Validation* phase

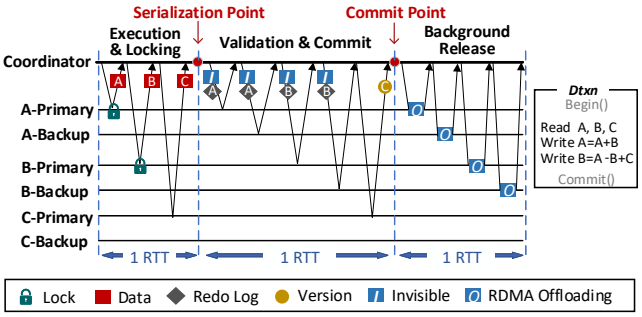


Figure 4: The workflow of dtxn processing in HDTX

undo and redo logs, respectively.

(a) **Using undo log**. The undo log contains the old version of the data and its metadata, and thus the computing node can in-place write the latest data to all memory nodes before committing the dtxn. As illustrated in Figure 3(a), after sending the undo log, the coordinator marks the locked data as invisible via `RDMA Atomic`, and then in-place updates the data via `RDMA Write` to commit the transaction. These two steps should be performed sequentially. Otherwise, other transactions may read the data being updated, causing inconsistent data accesses. However, according to the ordering rules, the coordinator must add `RDMA Fence` flag to the `RDMA Write` to maintain the order. This operation brings an additional RTT to the *Commit* phase. Even though the undo log can be propagated to all primaries/backups during the *Validation* phase, at least two RTTs are required to synchronize the latest data among all memory nodes. **Thus, dtxn systems using undo logs are unable to achieve fast commits in DM systems.**

(b) **Using redo log**: The redo log contains the transaction output (i.e. the latest data) and its metadata. Thus, the computing node can commit the transaction after writing the redo log to all primaries/backups, and then asynchronously updates the data using the redo log. Unlike undo logs, the order of propagating the latest data with redo logs and the visibility control do not affect the read consistency during the *Commit* phase. Thus, the coordinator can simply write the redo log to all primaries/backups via the `RDMA Write`, and then marks the locked data as invisible via the `RDMA Atomic`, as illustrated in Figure 3(b). The locked data can be in-place updated asynchronously in the *Release* phase, which is not on the critical path of dtxns. Since the order of batched `RDMA Write` and `RDMA Atomic` in the *Commit* phase is always maintained, as shown in Table 1, they can be performed sequentially in only one RTT. **Thus, we adopt the redo log and the visibility control mechanism to minimize network round trips during the *Commit* phase.**

4.2.2 Combining Validation and Commit Phases

The *Validation* phase checks whether the read-only set is updated by other dtxns during the *Execution* and *Locking* phases. The *Commit* phase propagates redo logs of the write set to all primaries/backups. Since there is no data dependency between the *Validation* phase and the *Commit* phase, HDTX combines these two phases to further reduce network round trips. In this way, HDTX can shorten the lock duration and enable fast dtxn commits. However, when there are high read-write conflicts, HDTX may rollback the *Commit* phase due to unsuccessful *Validation*. Fortunately, since redo logs can be discarded or overwritten directly, the rollback of the *Commit* phase incurs rather low performance overhead. Thus, this design can significantly accelerate dtxn processing particularly for workloads with low read-write contention.

4.2.3 Dtxn Workflow in HDTX

The *Execution* and *Locking* phases can be also combined to reduce network round trips [7,53]. Putting these optimizations together, Figure 4 illustrates how HDTX processes a dtxn using our fast commit protocol in DM systems. *It mainly includes the following three phases but can commit a dtxn within only two RTTs.*

(1) *Execution* & *Locking* phase. The coordinator issues batched `RDMA FAA(Atomic)` and `RDMA Read` primitives to lock and fetch the read-write set from primaries in one RTT. Since the order of these two primitives is always maintained according to the operation ordering rules, the coordinator can obtain the write lock and the latest data within one network round trip.

(2) *Validation* & *Commit* phase. The coordinator issues an `RDMA Read` to fetch the visibility bit and the version number of read-only data for validation. Meanwhile, it issues batched `RDMA Write` and `RDMA Read` to persistently write the redo log to all replicas, and uses `RDMA FAA(Atomic)` to atomically mark the locked data as invisible. Once the coordinator receives acknowledgments from all replicas and verifies that there is no conflict, it can commit the dtxn. We note that the visibility control can provide a consistent view among replicas when in-place updating the write set. Therefore, other coordinators only need to check the data version and visibility bits rather than the write locks, enabling concurrent accesses to read-only data. If the coordinator detects a missing acknowledgment or a failure before the commit point, it rolls back the dtxn. In this case, it should notify all replicas to reset visibility bits and release write locks.

Table 2: A summary of different dtxn systems

Dtxn Systems	Phases of dtxn protocol	# RTTs to commit
FaRM [12, 39]	*Execution + Locking + Validation + Commit Backup + Commit Primary*	5 RTTs
FaSST [28]	*Execution&Locking + Validation + Log + Commit Backup + Commit Primary*	5 RTTs
DrTM+H [48]	*Execution + Locking&Validation + Commit Backup + Commit Primary*	4 RTTs
FORD [53]	*Execution&Locking (UndoLog) + Validation + Commit + Background Release*	3 RTTs
HDTX	*Execution&Locking + Validation&Commit (RedoLog) + Background Release*	2 RTTs

(3) *Release* phase. The coordinator should update the locked data, increase the version number, make the locked data visible, and release the write lock. The coordinator issues an `RDMA Send` primitive to asynchronously conduct all these operations via RDMA-enabled offloading, as described in Section 4.3. If a failure occurs in this phase, the coordinator handles the failure according to the status of the uncompleted dtxn (Section 4.5).

We summarize previous dtxn protocols in Table 2. FaRM [12, 13] and FaSST [28] both require 5 RTTs to process a dtxn. DrTM+H [48] combines the *Locking* and *Validation* phases, and thus processes a dtxn with 4 RTTs. FORD combines the *Execution* and *Locking* phases, and performs its *Release* phase in the background, and thus can commit a dtxn in 3 RTTs. Our HDTX only requires two phases to commit a dtxn, and then performs the *Background Release* phase asynchronously.

When the validation fails due to read-write conflicts, all dtxn systems abort the current dtxn. In this case, both HDTX and other systems must release all acquired locks via RDMA operations. Since both HDTX and FORD send logs to memory nodes before the Validation phase, they both waste RDMA bandwidth for the log transmission upon a dtxn abort. Thus, the performance overhead of the rollback in HDTX is comparable to that of FORD.

4.3 RDMA-enabled Release Phase Offloading

We next consider how to efficiently execute the *Release* phase of dtxns on DM nodes. As discussed in Section 3, it is costly to send the latest data and the log in two rounds of data transfers in DM systems. We observe that the in-place updating of the write set can be efficiently achieved by RNICs on memory nodes, because it only involves data movement within each memory node when using redo logs. Thus, we leverage `RDMA Wait` and `RDMA Enable` primitives to sequentially offload all operations in the *Release* phase to RNICs of memory nodes.

Figure 5 illustrates how the memory node's RNIC can "copy" the redo log to update the data locally via RDMA-enabled operation offloading. When HDTX is initialized, two work queues are created particularly for RDMA offloading in the memory nodes' RNIC. Specifically, work queue 1 is paired with a work queue in the computing node to receive RDMA

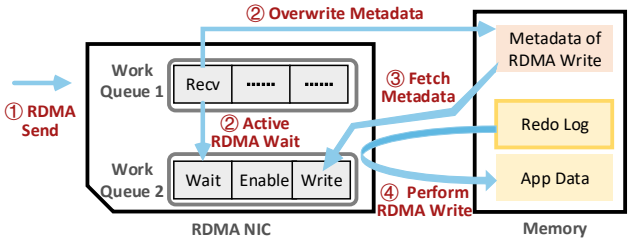


Figure 5: Offloading the data copying operation to the memory node's RNIC via `RDMA Wait/Enable` primitives

offloading messages. The memory node sequentially issues three RDMA primitives, i.e., `RDMA Wait`, `RDMA Enable`, and `RDMA Write` to work queue 2. At this time, the source and destination addresses of the `RDMA Write` are set as NULL in its metadata. Besides the addresses, the metadata of RDMA primitives also contains the type of RDMA operation, access permissions, the data size, and so on. These items can be used by the RNIC on the memory node to perform `RDMA Write` operations. Then, the memory node issues an `RDMA Recv` to work queue 1. The destination address of the `RDMA Recv` is the memory region that stores the metadata of the `RDMA Write`. Since the `RDMA Wait` in work queue 2 is configured to wait for the work completion event of the first position in work queue 1, the RNIC should monitor the completion event of `RDMA Recv`, and then continue the pending RDMA primitives blocked by the `RDMA Wait`.

When the redo log in the memory node should be copied to update the application data locally, the computing node sends the source and destination addresses of the data copying operation to the memory node via an `RDMA Send` (①), which consumes the `RDMA Recv` in work queue 1 and overwrites the metadata of the `RDMA Write` with the content of the `RDMA Send`. Meanwhile, the RNIC generates a work completion event of work queue 1, which activates the pending `RDMA Wait` in work queue 2 (②). Then, the RNIC continues to process the other two pending RDMA primitives in work queue 2. Since the `RDMA Write` is accompanied by an `RDMA Enable`, the RNIC fetches the metadata of the `RDMA Write` from the memory node, enabling it to be executable (③). Finally, the memory node's RNIC performs the `RDMA Write` (④) to copy data from the redo log to the application's data area, without the intervention of the memory node's CPUs.

To offload all operations in the *Release* phase to the RNIC, the memory node has to issue two `RDMA Writes` and one `RDMA FAA` to its RNIC and blocks them via `RDMA Wait/Enable` primitives during the initialization of HDTX. Figure 6 shows all operations in the *Release* phase executed by RNICs on memory nodes. After responding the dtxn output to clients, the coordinator issues an `RDMA Send` to activate all blocked RDMA operations for each memory node. Then, the memory nodes' RNICs overwrite the metadata of blocked RDMA operations according to the content of the `RDMA Send`, including the address of redo logs, the addresses

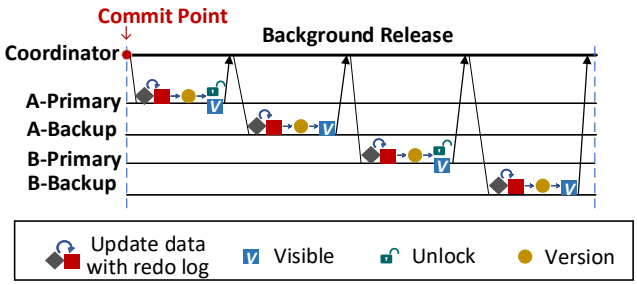


Figure 6: Operations executed by RNICs in the *Release* phase

of the data and its metadata, and the data size. At this time, memory nodes' RNICs are aware of the source and destination addresses of the pending data copying operation, and perform the first blocked `RDMA Write` to update the data with redo logs. Similarly, HDTX leverages the second blocked `RDMA Write` to offload the version updating. The blocked `RDMA FAA` primitive is used to modify visibility bits for all primaries/backups, and to release locks for all primaries. Since the RDMA-enabled offloading in the *Release* phase would consume a set of RDMA primitives in the two work queues, HDTX supplements these RDMA primitives in batches for upcoming dtxns using a background thread. In this way, all operations in the *Release* phase can be offloaded to the RNIC, without any intervention of the memory node's CPUs.

4.4 Decentralized Priority Locking

Most dtxn processing systems are concerned with the latency of individual transactions. To meet users' requirements, HDTX leverages decentralized priority-based locking to support flexible dtxn scheduling, i.e., assigning a high priority to dtxns that are mission-critical or are about to violate their latency requirements soon.

4.4.1 Lock Presentation

To manage and schedule lock requests for dtxns, we construct two queues assigned different priorities for each lock in memory nodes. Specifically, we queue requests with the same priority in a *first-in-first-out* (FIFO) manner using Lamport's Bakery algorithm [32], and rely on coordinators to determine the order of requests in different queues.

The Lamport's Bakery algorithm is applicable to the lock management in a DM architecture since it allows computing nodes to collect the status of locks on memory nodes, without involving memory nodes' CPUs. When a computing node requests a lock, it receives an incremental token from the memory node. The memory node records the number of lock requests and the token of the current computing node that holds the lock. If the token received by the computing node matches the token recorded by the memory node, the computing node acquires the lock. Based on this algorithm, we manage lock requests using two FIFO queues, i.e., a normal

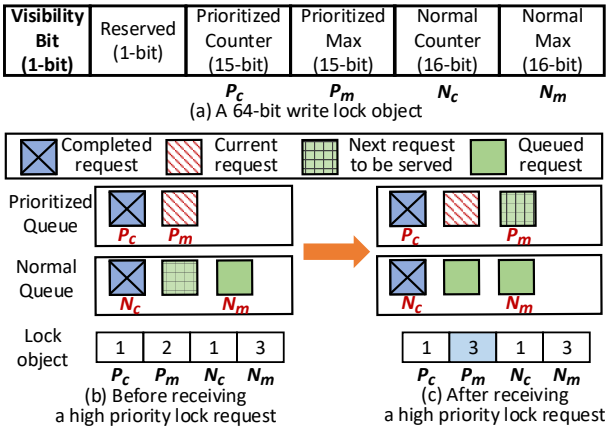


Figure 7: The write lock representation and an illustration of the decentralized priority-based locking

queue and a prioritized queue, to achieve more flexible dtxn scheduling. The former and latter queues are assigned with a low priority and a high priority, respectively.

Figure 7(a) shows the 64-bit representation of a priority-based lock object. For each lock, we construct a normal queue using a pair of $< N_c,\ N_m >$, which denotes the token of the current dtxn holding the lock and the number of dtxns requesting the lock, respectively. We also use a pair of $< P_c, P_m >$ to construct the prioritized queue, which works similarly to the normal queue. Once the current lock is released, the pending requests in the prioritized queue can acquire the lock ahead of requests in the normal queue. If a dtxn fails to acquire a lock for several times, it would retry with a high-priority lock request so that the priority of dtxns can be dynamically adjusted to mitigate the risk of latency requirement violations.

Figure 7(b) and (c) illustrate how the status of the lock object and two queues change when a high-priority lock request is received. As shown in Figure 7(b), there are two lock requests in the prioritized queue (P_m=2). The first request in the prioritized queue is completed (P_c=1), while the second request has acquired the lock. In the normal queue, there are three lock requests (N_m=3), and the first request is completed (N_c=1). If the current request completes, the second request in the normal queue can acquire the lock. When a new lock request with a high priority arrives, as shown in Figure 7(c), the number of requests in the prioritized queue increases from two to three (P_m=3). According to our priority-based locking algorithm, if the current request completes, the third request in the prioritized queue instead of the second request in the normal queue can acquire the lock. In this way, our priority-based locking mechanism can reduce the latency of mission-critical dtxns by assigning them a high priority.

4.4.2 Handling Priority-based Locking

HDTX leverages `RDMA FAA` primitives for locking operations, including lock acquisition and release. The `RDMA CAS` primitive is also used for special cases, such as handling lock over-

Table 3: Key notations and procedures used in HDTX

Function	Description
L(segment)	returns the value of *segment* in the 64-bit lock object *L*. The *segment* is one of P_c, P_m, N_c, and N_m.
READ(lock)	fetches the value of the *lock* via an RDMA Read.
FAA(lock, segment, value)	atomically adds *value* to a *segment* of *lock* via an RDMA FAA and receives the original value of *lock*.
CAS(lock, value)	compares and swaps the value of *lock* if it equals to the given *value* via an RDMA CAS, and receives the original value of *lock*.
CheckOverflow(L)	checks whether there is an overflow or an error on the lock *L*, i.e. $L(P_m) > maximum$ or $L(N_m) > maximum$, and returns *True* if an overflow occurs.

flows and deadlocks. Upon receiving a dtxn request, the coordinator performs the *Execution & Locking* phase to fetch and lock the read-write set. The coordinator determines whether it acquires the lock by comparing the returned token with the current value of the lock object. Once the coordinator acquires all locks required by the dtxn, it begins the next phase to validate the version and commit the dtxn. In the *Release* phase, the coordinator releases write locks via the `RDMA FAA`. We elaborate these locking operations in the following.

Algorithm 1: Normal Lock Acquisition

```
   input  : the requested lock object
   output : Success or Failure
1  L_old = FAA(lock, N_m, 1);
2  L = L_old;
3  do
4      if CheckOverflow(L) = True then
5          Repeat CAS(lock, L_old) till it succeeds or the lock
             resets;
6          return Failure;
7      end
8      if L(P_c) = L(P_m) AND L(N_c) = L_old(N_m) then return
         Success;
9      L = READ(lock);
10     Sleep ( L_old(N_m) - L(N_c) + L(P_m) - L(P_c) ) μs;
11 while true;
```

Lock Acquisition: Algorithm 1 and Algorithm 2 show pseudocodes for lock acquisitions in HDTX. All notations and procedures used in these algorithms are listed in Table 3. For **(a) normal requests**, as shown in Algorithm 1, the coordinator adds "1" to N_m and obtains its token from the returned value via `RDMA FAA`. Then, it polls the lock till there are no pending requests in these two queues (lines 3-11 in Algorithm 1). To mitigate the overhead of busy polling, the coordinator waits an interval that is proportional to the number of pending high-priority requests ($L(P_m)$ - $L(P_c)$) and the preceding normal requests ($L_{old}(N_m)$ - $L(N_c)$) before attempting to fetch the lock value again (line 10 in Algorithm 1). For **(b) high-priority requests**, as shown in Algorithm 2, the coordinator adds "1" to P_m and receives the current lock value. According to the number of pending requests, there are three scenarios. (b.1) If both normal and prioritized queues are empty, the coordinator acquires the lock directly (lines 6-7). (b.2) If there are only pending requests in the normal queue, implying that a normal request already acquires the lock, the coordinator waits for the lock till the current request is completed (lines 8-12). (b.3) Otherwise, if there are pending requests in the prioritized queue, the coordinator waits for their completions (lines 14-20).

Algorithm 2: High-Priority Lock Acquisition

```
   input  : the requested lock object
   output : Success or Failure
1  L_old = FAA(lock, P_m, 1);
2  if CheckOverflow(L_old) = True then
3      Repeat CAS(lock, L_old) till it succeeds or the lock resets;
4      return Failure;
5  end
6  if L_old(P_c) = L_old(P_m) then
7      if L_old(N_c) = L_old(N_m) then return Success;
8      do
9          L = READ(lock);
10         CheckOverflow(L);// lines 2-5
11         if L(N_c) = L_old(N_c) + 1 then return Success;
12     while true;
13 else
14     do
15         L = READ(lock);
16         CheckOverflow(L); // lines 2-5
17         if L(P_c) = L_old(P_m) then return Success;
18         Sleep ( L_old(P_m) - L(P_c) ) μs;
19     while true;
20 end
```

Lock Release: After a transaction is committed or aborted, the coordinator asynchronously releases the write lock. This is achieved by adding "1" to N_c or P_c via the `RDMA FAA`. Subsequently, other coordinators waiting for the lock can acquire it according to our lock acquisition algorithm.

Handling Lock Overflows: HDTX stores the 1-bit visibility flag and the 63-bit write lock in an 8-byte value, allowing atomic modifications via `RDMA Atomic` primitives. To detect overflows, the highest bit of each segment is reserved as a canary value [45,51]. Therefore, the maximum of the segment representing the normal queue is 32768. For the prioritized queue, its maximum value is 16384 because the highest bit of the lock is reserved as the visibility bit. When any coordinator finds that its received token is larger than the maximum, it withdraws its request via the `RDMA CAS` (lines 4-7 in Algorithm 1 and lines 2-5 in Algorithm 2). If the current coordinator holding the lock detects an overflow, it releases the lock by resetting all segments to "0" via `RDMA CAS`. Then, the lock overflow is resolved and queued coordinators can retry their lock requests.

4.5 Handling Deadlocks and Failures

HDTX assumes a non-Byzantine failure model in which servers may fail-stop but never exhibit arbitrary faulty behaviors. Like previous systems [25, 47, 51], HDTX extends the lease mechanism [21] to handle various failures. In detail, coordinators calculate an elapsed time locally to monitor the lease expiration for each lock. The coordinator resets the elapsed time to zero once it acquires the lock, or detects an update on N_c or P_c or the visibility bit. The coordinator holding locks can process the dtxn only if all leases are not expired. The coordinator waiting for locks assumes that a failure occurs if the elapsed time of any lock exceeds a threshold. The failure detection relies on an assumption that clocks advance at nearly the same speeds [19, 51]. Fortunately, commercial networks [33] with a clock drift of ± 0.5 ns per microsecond is sufficient for our mechanism.

Deadlocks. In HDTX, deadlocks only occur during the *Execution & Locking* phase. The coordinator assumes a deadlock if the data is readable (i.e., the visibility bit is "1") and the elapsed time is twice greater than the lease expiration time. Similar to DSLR [51], any coordinator detecting a deadlock atomically hands over the lock to its successor via an `RDMA CAS`. Subsequently, it releases all acquired locks and reapplies for high-priority or normal locks later according to the client's latency requirement. Also, the queued requests can retry to acquire the lock. Based on our experimental results, we set the lease expiration time to 1 millisecond by default.

Failures of Computing Nodes. To facilitate failure recovery of computing nodes, the coordinator records the metadata of all objects accessed by a dtxn in the redo log. If a computing node fails (1) during the *Execution & Locking* phase: other coordinators assume a deadlock and atomically hand over the lock. Thus, dtxns executed on the failed coordinator can be safely aborted, and then clients can issue dtxn requests to other coordinators. If a computing node fails (2) after the *Execution & Locking* phase: other coordinators assume a failure of a computing node when the locked data is invisible and the elapsed time is twice greater than the lease expiration time. According to the redo log, another computing node is assigned to check the status of uncompleted dtxns and take over the failed node. (2.1) If any object involved in the dtxn is updated or the client has been acknowledged that the dtxn is committed, the newly-assigned computing node continues the *Release* phase. (2.2) Otherwise, it is safe to abort the dtxn directly. In all cases, when the newly-assigned computing node begins to recover dtxns, the lock lease of coordinators that are suspected to be failed is certain to expire, preventing the suspected coordinator from updating the data.

Failures of Memory Nodes. When any coordinator identifies a failure of a memory node, a computing node is assigned to replicate the replica from other memory nodes. (1) If a memory node fails before the *Release* phase, coordinators can simply terminate their dtxns and release their locks. (2) If a memory node fails during the *Release* phase, coordinators can commit dtxns normally regardless of the failed node, and notify the newly-assigned computing node to recover the replica with the latest version.

Failures of Networks. Network failures result in network partitions, where HDTX allows only the primary partition [6, 20] to serve client requests for strong consistency. When the network connectivity restores, coordinators holding locks yet outside the primary partition cannot continue their dtxns due to the expired lease, thereby guaranteeing data consistency.

4.6 Datastore Implementation

During the cluster initialization, HDTX allocates multiple memory regions on the DM pool to facilitate RDMA operations. Memory nodes use hash tables to store key-value objects in these memory regions. Each hash table entry includes metadata and data of an object. The metadata contains the write lock, the version number, the key, the object size, and the offset in the remote memory region, etc. In HDTX, we set the maximum object size to 1 KB since it satisfies most OLTP workloads. For variable-sized objects, HDTX stores their data and metadata in different areas and accesses the data using pointers. However, a coordinator can still fetch the data and its metadata in a single round trip by issuing two batched `RDMA Read` primitives.

To access an object, the coordinator calculates the address of the hash bucket according to the key and retrieves all entries to locate the object. It then caches the metadata locally, allowing subsequent read requests to directly fetch the object. The coordinator can cache the metadata without a capacity limitation since a GB-scale cache is commonly deployed on computing nodes [41, 53]. Thus, most dtxns can fetch and lock the write set within one RTT if there is no lock contention.

4.7 Discussions

Correctness and Serializability. Similar to traditional dtxn systems using OCC, HDTX locks the read-write set and validates the read-only set to guarantee data consistency. Specifically, the locking mechanism avoids conflicted accesses to the read-write set, while the validation phase can guarantee that the read-only set has not been updated by other dtxns at the serialization point. Unlike existing dtxn systems, HDTX combines *Validation* and *Commit* phases. The coordinator validates the version of the read-only set and transfers the redo logs without modifying the application data. Therefore, a validation failure requires HDTX to roll back the dtxn by releasing the locks and marking data visible. This avoids data inconsistency because the data is unreadable upon validation failures. Since HDTX only commits the dtxn when the version validation and the redo log propagation are completed, the combination of these two phases does not compromise the correctness and serializability of dtxns.

Optimization for Write-Only Dtxns. In HDTX, coordinators acquire locks and read data to perform the transaction logic in the *Execution&Locking* phase. For normal read-modify-write dtxns, coordinators must (1) read data from memory nodes and generate redo logs during the *Execution&Locking* phase, and (2) then send redo log to memory nodes to commit dtxns during the *Validation&Commit* phase. Thus, it requires two phases to commit a read-modify-write dtxn. In contrast, for write-only dtxns, coordinators can directly generate and send redo logs to memory nodes, and meanwhile try to lock the write-set in the first RTT. This process is similar to a write operation in a KV store system. If locks are acquired successfully, the coordinator can commit the dtxn, and then update the data and release the locks asynchronously. In this case, it is possible to commit a write-only dtxn in a single RTT.

Consistency of Reading Lock Objects. HDTX can guarantee data consistency when reading a lock object with an `RDMA Read` primitive. According to Intel's Programming Manual [1], Intel processors can guarantee that the memory operation of reading or writing 8 bytes will always be carried out atomically. The `RDMA Read` operation also relies on CPU's integrated memory controller to access main memory [26, 50]. Thus, like previous work [19, 51], we use the `RDMA Read` to read a lock object atomically.

5 Evaluation

5.1 Experimental Setup

Testbed. We conduct our experiments in an RDMA-capable cluster with 5 servers. All servers are equipped with 128 GB DRAM, 1 TB Intel Optane DCPMM, two-socket Intel Xeon Gold 6230 processors with 20 cores running at 2.10 GHz, and a Mellanox ConnectX-3 40/56 GbE network interface controller. To set up the DM system, we use two servers as memory nodes for 2-way replication, and each node is restricted to use only one CPU core for cluster initialization. All memory nodes use Intel Optane DCPMMs in the *App Direct Mode* to achieve fast durability for the datastore. If not specified otherwise, one server is used as the computing node in which multiple threads are initialized to execute dtxns in parallel. Each computing node communicates with the memory nodes via the RDMA *Reliable Connection* (RC) mode. When the redo logs are written to the disaggregated persistent memory, we exploit the one-sided RDMA Flush [14, 16] (i.e., RDMA read-after-write mechanism) to guarantee the durability of dtxns. Specifically, we use an `RDMA Write` to transfer the log, and an `RDMA Read` to read the last byte of the log for validation. Since the batched `RDMA Write` and `RDMA Read` primitives are strictly ordered, as shown in Table 1, the RDMA read-after-write mechanism can guarantee remote data persistence within one RTT.

Benchmarks. We use three typical OLTP workloads, TPC-C [5], SmallBank [3], and TATP [4], as macro-benchmarks to evaluate the performance of different dtxn systems. **1) TPC-C** simulates a complex ordering system. It consists of 9 tables, with the maximum object size of 672 bytes. There are 92% read-write transactions in this benchmark. We generate 20 warehouses by default. **2) SmallBank** simulates a banking application. It contains 2 tables, and all objects are 16 bytes. There are about 85% read-write transactions in this workload. **3) TATP** is a telecom application. It contains 4 tables, and the maximum object size is 48 bytes. About 80% of its transactions are read-only. Besides the macro-benchmarks, we also develop micro-benchmarks [23] to evaluate the effectiveness of our three individual techniques. In these micro-benchmarks, the hash tables in the memory nodes store one million key-value pairs, where the key size is 8 bytes and the default value size is 40 bytes.

Methodology. In all dtxn systems, the coordinator retries transactions in case of failures and deadlocks [49, 51], and reports a transaction abort to the client only when the transaction becomes time-out ultimately. We set the default timeout as 1 millisecond to guarantee a low transaction abort rate, with a low performance degradation. We measure the throughput by counting the number of committed transactions per second, and measure the latency by recording the interval between two committed transactions. All experimental results are the average of three trials.

Comparisons. We compare HDTX with two OCC-based RDMA-enabled dtxn systems– FORD [53] [18] and FaRM [39] [12]. The state-of-the-art FORD is designed particularly for DM systems, while Microsoft's FaRM is an in-memory distributed computing platform that uses RDMA-based RPCs to process dtxns using memory nodes' CPUs. To evaluate FaRM in a DM environment, we adopt a derivation of FaRM in which the coordinators use one-sided RDMA primitives to access remote data [18]. This DM-compatible FaRM directly accesses data in the persistent memory pool instead of using a circular buffer to receive RDMA messages, and thus saves memory space and avoids the cost of data copying. Thus, we compare HDTX with the DM-compatible FaRM rather than the original FaRM.

For a fair comparison, all systems adopt optimizations in FORD, including coroutines [28], outstanding requests, and doorbell batching [27]. Coroutines are lightweight, user-space functions that manage their own execution flow cooperatively and switch contexts without kernel involvement. We use multiple coroutines in a single thread as coordinators to fully utilize CPU resources when waiting for RDMA ACKs.

5.2 Macro-benchmarks

We use 16 threads, with each thread launching 7 worker coroutines as coordinators. This setup enables all dtxn systems to achieve near-optimal performance using a single computing node. Figure 8 shows the end-to-end performance of each

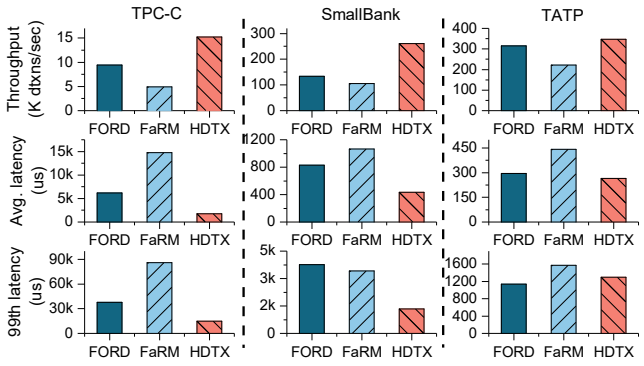


Figure 8: The end-to-end performance of different workloads

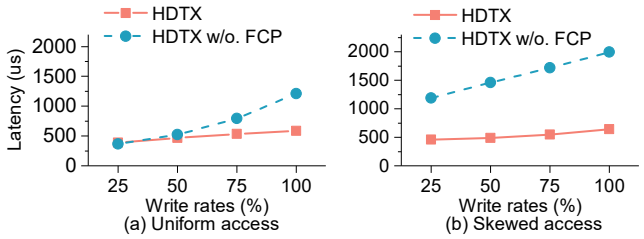


Figure 9: The effectiveness of *fast commit protocol* (FCP)

dtxn system using three OLTP workloads. Compared with FORD and FaRM, HDTX significantly reduces the average latency of TPC-C by 72.1% and 88.3%, improves its throughput by 84.7% and 2.08×, and reduces its 99th percentile latency by 60.9% and 82.7%, respectively. For SmallBank, HDTX also improves the throughput, and reduces the average latency and the tail latency significantly. For TATP, HDTX and FORD show comparable performance because most dtxns in TATP are read-only and they allow all backup memory nodes to serve read requests for high throughput, while coordinators in FaRM can only read data from primary memory nodes.

FORD outperforms FaRM because it exploits the visibility control technique to accelerate the *Commit* phase, and thus reduces network round trips of dtxns. HDTX can further reduce the latency by combining more dtxn processing phases in the proposed fast commit protocol, and can also significantly reduce the latency by scheduling dtxns according to their latency requirements. Thus, HDTX is particularly efficient for dtxns with high read-write conflicts and write-write lock contention, such as TPC-C and SmallBank.

5.3 Micro-benchmarks

In this section, we evaluate the effectiveness and efficiency of the proposed three techniques in HDTX individually.

***Fast Commit Protocol* (FCP).** First, we demonstrate the benefit of FCP using uniform and skewed [8] access distributions. Figure 9 shows the average latency of HDTX with/without FCP at different write rates. We initiate 64 coordinators to serve client requests, with each coordinator accessing two objects per dtxn. HDTX w/o. FCP shows a significant growth of latency when the write rate increases. Particularly, HDTX can reduce the average latency of dtxns by up to 67.7% compared with HDTX w/o. FCP for the skewed access distribution. Although FCP may incur the rollback of the commit phase due to read-write conflicts, FCP still demonstrates its effectiveness for the scenario using skewed access distributions under heavy read-write conflicts, as shown in Figure 9(b). This implies that the benefit of FCP is much higher than the overhead of the rollback. Overall, HDTX exploits the fast commit protocol to minimize the network rounds trips and reduce the average lock duration, and thus can significantly improve the performance of write-intensive workloads.

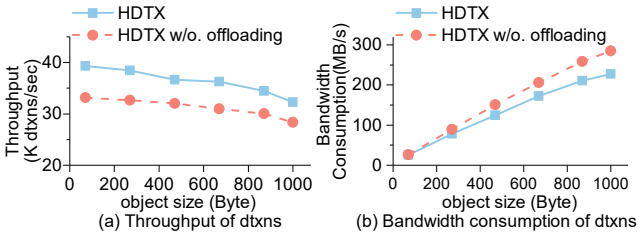


Figure 10: The effectiveness of release phase offloading

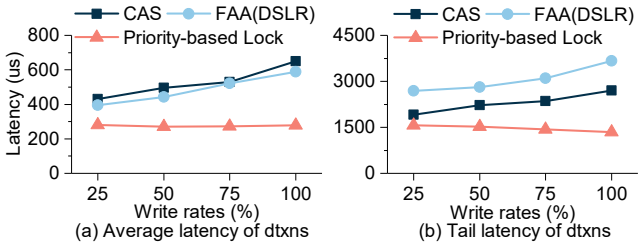


Figure 11: The effectiveness of priority-based locking

RDMA-enabled Release Phase Offloading. Second, to demonstrate the impact of the *Release* phase offloading on RDMA bandwidth consumption, we evaluate this design under high network load by running a background RDMA communication task. Figure 10 shows the throughput and RDMA bandwidth consumption of dtxns when the object size increases from 64 bytes to 1 KB. Each coordinator accesses four objects in a dtxn. Without enabling the offloading mechanism, the coordinator has to send the latest data from coordinators to memory nodes to update the write set during the *Release* phase. Compared with HDTX w/o. offloading, HDTX reduces the RDMA bandwidth consumption by up to 19.1%, and improves the throughput by up to 18.5%. Although `Wait/Enable` primitives may slow down the RNIC pipeline, the net benefit of the RDMA offloading is still considerable because our design can avoid an extra data transfer by offloading the data synchronization operations to memory nodes' RNICs.

Priority-Based Locking for Mission-Critical Dtxns. Third, we demonstrate the effectiveness of the priority-based locking mechanism. We compare it with the `RDMA CAS`-based

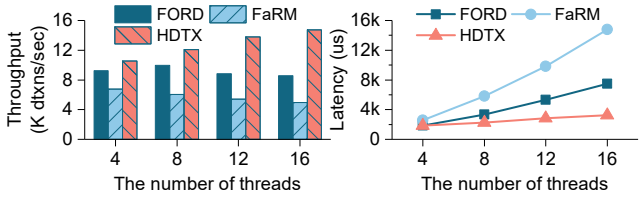


Figure 12: The throughput and the average latency of TPC-C vary with the number of threads

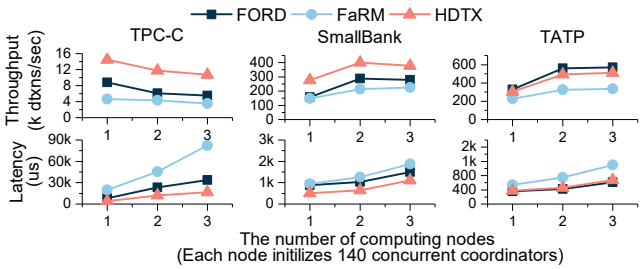


Figure 13: The throughput and the average latency of different applications vary with the number of computing nodes

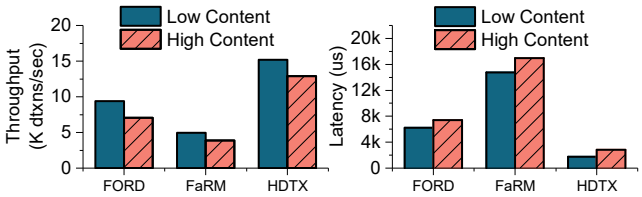


Figure 14: The throughput and average latency in low-contention and high-contention scenarios

locking mechanism adopted by FORD [53] and DrTMR [7], and the `RDMA FAA`-based locking mechanism proposed in DSLR [51]. We randomly select 20% of requests as user-specified mission-critical dtxns with a high priority, and evaluate the average and tail latency of these dtxns under different write rates with a skewed access distribution. As shown in Figure 11, HDTX using priority-based locking reduces the average latency of mission-critical dtxns by 57.1% and 52.8%, and reduces the tail latency by 50.2% and 63.3% compared with `RDMA CAS`-based and `RDMA FAA`-based mechanisms, respectively. For the tail latency, the `RDMA CAS`-based mechanism outperforms the `RDMA FAA`-based mechanism because it aborts and retries dtxns immediately once the coordinator fails to acquire all write locks, while the `RDMA FAA`-based mechanism must wait for a given lease time. Most importantly, these two mechanisms can not schedule user-specified dtxns. In contrast, our priority-based locking mechanism can significantly reduce the latency of read-write dtxns by assigning a high priority, and thus is particularly effective for scheduling mission-critical dtxns.

5.4 Sensitivity Studies

The Number of Threads. Figure 12 shows the throughput and average latency of TPC-C in different dtxn systems when the number of threads in the computing node increases from 4 to 16. Each thread initializes 7 worker coroutines as coordinators. HDTX shows a significant growth of throughput and a slight growth of latency. In contrast, the latency of FaRM increases significantly since FaRM commits a dtxn with five RTTs in a DM architecture. FORD outperforms FaRM because it optimizes the dtxn commits by exploiting the visibility control and undo logs to synchronize the latest data. However, FORD still requires three RTTs to commit a transaction. Since a longer lock duration would increase the potential of lock conflicts, FORD suffers from throughput degradation when the number of threads exceeds 8. When 16 threads are used, HDTX significantly improves the throughput by 72.7% and 1.98×, and reduces the latency by 56.7% and 78.1%, respectively, compared with FORD and FaRM. This implies that HDTX has much better performance scalability when more coordinators are used.

The Scale of Computing Nodes. Figure 13 shows the throughput and average latency of TPC-C, SmallBank, and TATP in different dtxn systems when the number of computing nodes increases. Each computing node initializes 20 threads with 7 worker coroutines per thread. Thus, the three computing nodes can initialize at most 420 concurrent coordinators. Since TPC-C has more read-write conflicts than SmallBank and TATP, the throughput of TPC-C decreases in all dtxn systems when the number of concurrent coordinators is larger than 140 (one computing node). Specifically, when we use 3 computing nodes to initialize a total of 420 concurrent coordinators for TPC-C, HDTX significantly improves the throughput by 81.8% and 2.06×, and reduces the latency by 64.1% and 79.9%, compared with FORD and FaRM, respectively. In this scenario, the high read-write contention causes more validation failures, and thus both FORD and HDTX consume more RDMA bandwidth to write logs. Specifically, FORD writes the undo log before the *Validation* phase, while HDTX writes the redo log during the *Validation* phase. However, since HDTX enables fast transaction commits with fewer execution phases, HDTX shows much higher performance than FORD. Since SmallBank and TATP have fewer read-write conflicts, all dtxn systems can improve the throughput by using more concurrent coordinators, but HDTX offers lower latency than FORD and FaRM. Overall, HDTX shows much better scalability than FORD and FaRM for large-scale concurrent dtxns.

Data Contention. Figure 14 shows the throughput and average latency of TPC-C under low and high contention of data accesses. We can reduce the number of warehouses to increase data contention. In the high-contention scenario, dtxns are more likely to access the same object, resulting in frequent lock contentions and validation failures. However, even when the number of warehouses decreases from 20 to 8,

we find that the failure rate of the *Validation* phase in HDTX only increases from 8.1% to 9.8%. Compared with FORD and FaRM, HDTX reduces the average latency of dtxns by 61.8% and 83.4%, and improves dtxn throughput by 83.2% and 2.3×, respectively, under the high-contention scenario. Since TPC-C has 92% read-write transactions with the maximum object size of 672 bytes, HDTX is particularly effective for handling this application with high contention because our priority-based locking mechanism can significantly reduce lock conflicts between dtxns. Moreover, HDTX can reduce the RDMA bandwidth consumption for large objects by offloading the data synchronization and accelerate the dtxn processing significantly using the fast commit protocol. Thus, HDTX can still achieve high-performance dtxn processing even in high-contention scenarios.

6 Related Work

RDMA-based Distributed Transaction Systems. A number of studies have exploited fast RDMA networks to improve the performance of distributed transactions. Microsoft's FaRM [39] [12] exploits OCC and PBR protocols to support RDMA-based dtxn processing. FaSST [28] adopts RDMA-based RPCs to process complicated messages with remote CPUs, and thus decreases the number of RTTs for dtxn processing. DrTM+H [48] proposes a hybrid approach that exploits one-sided or two-sided RDMA primitives to perform different phases of a transaction. Zeus [30] exploits an ownership protocol to transform dtxns into local ones, and thus can accelerate workloads with high data locality. DINT [55] exploits eBPF technologies to transparently offload some operations of dtxns to the kernel space, without kernel modifications. It eliminates the performance overhead of network stacks while guaranteeing security. However, since these systems are designed for traditional monolithic servers equipped with both computing and memory resources, they cannot be applied to DM systems directly. Xenic [40] offloads some functions of the dtxn protocol to SmartNICs to reduce PCIe operations. FORD [53] exploits one-sided RDMA to achieve fast dtxn processing for DM systems, and guarantees remote persistence with low network overhead. Motor [52] proposes contiguous version tuples to construct a multi-versioning datastore in a DM environment. It eliminates the logging overhead while allowing coordinators to concurrently read data during modification. Inspired by these studies, HDTX further optimizes the performance of dtxn for one-sided RDMA-based DM systems via three novel designs, including the fast commit protocol, RDMA-enabled offloading, and decentralized priority-based locking. By carefully orchestrating these techniques, HDTX can minimize the RTTs for dtxn processing, significantly improve the throughput, and reduce the latency of dtxns.

Management of Distributed Locks. Lock management is a fundamental technique for preventing race conditions and ensuring data consistency in distributed systems. In traditional distributed locking schemes [2, 22, 24], data nodes' CPUs act as a central point to enable flexible scheduling of shared object accesses. To alleviate the CPU load in data nodes, some studies [28, 53] exploit one-sided `RDMA CAS` primitives to request locks in a blind-retry manner. To avoid the starvation problem caused by `RDMA CAS`, a decentralized mutex lock mechanism [10] is proposed to schedule lock requests in a FIFO manner. N-CoSED [36] uses two 32-bit segments of a 64-bit lock object to support shared and exclusive access modes. However, they require additional communications between coordinators to propagate the ownership of the lock along the FIFO queue. DSLR [51] exploits the Lamport's Bakery algorithm to avoid write starvation and to eliminate communication among coordinators, and thus improves the performance of dtxns. We go further to challenge an intuitive understanding that decentralized locks are hard to support priority-based scheduling, and propose decentralized priority-based locking to schedule transactions at each computing node. In this way, HDTX can significantly reduce the latency of mission-critical transactions in DM systems.

7 Conclusion

Distributed transactions usually are processed inefficiently in disaggregated memory systems due to high latency of remote memory accesses. This paper presents HDTX, a high-performance dtxn system that delivers fast dtxn services and supports flexible dtxn scheduling in the DM architecture. We first conduct an in-depth analysis of the characteristics of dtxns and one-sided RDMA primitives. Based on our analysis, we propose a fast commit protocol to commit dtxns within minimum network round trips and significantly reduce the latency of dtxns. We further propose an RDMA-enabled offloading mechanism to reduce data transfers across computing and memory nodes by carefully orchestrating different RDMA primitives. In addition, we propose a decentralized priority-based locking mechanism to schedule mission-critical dtxns, thereby significantly reducing the tail latency of dtxns. Experimental results demonstrate that HDTX provides higher throughput and lower latency than state-of-the-art RDMA-based dtxn systems, such as FaRM and FORD.

Acknowledgments

We appreciate our shepherd Roberto Palmieri and anonymous reviewers' constructive comments for improving the quality of this paper. This work is supported jointly by the National Key Research and Development Program of China under grant 2022YFB4500303, National Natural Science Foundation of China (NSFC) under grants 62332011, 62302178, NSFC-RGC under grant 62461160333, and Natural Science Foundation of Hubei Province under grant 2021CFA037.

References

[1] Intel 64 and IA-32 Architectures Software Developer Manuals. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html, 2025.

[2] Amanda Baran, Jacob Nelson-Slivon, Lewis Tseng, and Roberto Palmieri. Alock: Asymmetric lock primitive for RDMA systems. In *Proceedings of the 36th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'24)*, pages 15–26, 2024.

[3] SmallBank Benchmark. https://hstore.cs.brown.edu/documentation/deployment/benchmarks/smallbank/, 2025.

[4] Telecom Application Transaction Processing Benchmark. https://tatpbenchmark.sourceforge.net/, 2025.

[5] TPC-C Benchmark. https://www.tpc.org/tpcc/, 2025.

[6] Eric A. Brewer. Pushing the CAP: strategies for consistency and availability. *Computer*, 45(2):23–29, 2012.

[7] Yanzhe Chen, Xingda Wei, Jiaxin Shi, Rong Chen, and Haibo Chen. Fast and general distributed transactions using RDMA and HTM. In *Proceedings of the Eleventh European Conference on Computer Systems (EuroSys'16)*, pages 26:1–26:17, 2016.

[8] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC'10)*, pages 143–154, 2010.

[9] Introducing Intel Rack Scale Design. https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/rack-scale-design-architecture-white-paper.pdf, 2018.

[10] Ananth Devulapalli and Pete Wyckoff. Distributed Queue-Based Locking Using Advanced Network Features. In *Proceedings of the 34th International Conference on Parallel Processing (ICPP'05)*, pages 408–415, 2005.

[11] Bailu Ding, Lucja Kot, and Johannes Gehrke. Improving optimistic concurrency control through transaction batching and operation reordering. *Proc. VLDB Endow.*, 12(2):169–182, 2018.

[12] Aleksandar Dragojevi, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. FaRM: Fast Remote Memory. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI'14)*, pages 401–414, 2014.

[13] Aleksandar Dragojević, Dushyanth Narayanan, Edmund B. Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. No Compromises: Distributed Transactions with Consistency, Availability, and Performance. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP'15)*, pages 54–70, 2015.

[14] Zhuohui Duan, Haodi Lu, Haikun Liu, Xiaofei Liao, Hai Jin, Yu Zhang, and Song Wu. Hardware-supported remote persistence for distributed persistent memory. In *Proceedings of the 2021 International Conference for High Performance Computing, Networking, Storage, and Analysis (SC'21)*, pages 91:1–91:14, 2021.

[15] Idan Burstein. RDMA Memory Placement Extensions for PMEM. https://www.flashmemorysummit.com/English/Collaterals/Proceedings/2018/20180808_PMEM-202-1_Burstein.pdf, 2018.

[16] RDMA Extensions for Remote Persistent Memory Access. https://openfabrics.org/images/eventpresos/2016presentations/215RDMAforRemPerMem.pdf, 2016.

[17] PolarDB for Xscale. https://www.alibabacloud.com/en/product/polardb-for-xscale?_p_lc=1&spm=a3c0i.7911826.2564562790.3.1ce82129Zefh2t#J_7387178670, 2024.

[18] FORD. https://github.com/minghust/ford, 2022.

[19] Jian Gao, Youyou Lu, Minhui Xie, Qing Wang, and Jiwu Shu. Citron: Distributed range lock management with one-sided RDMA. In *Proceedings of the 21st USENIX Conference on File and Storage Technologies (FAST'23)*, pages 297–314, 2023.

[20] Seth Gilbert and Nancy A. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, 2002.

[21] Cary G. Gray and David R. Cheriton. Leases: An Efficient Fault-Tolerant Mechanism for Distributed File Cache Consistency. In *Proceedings of the Twelfth ACM Symposium on Operating System Principles (SOSP'89)*, pages 202–210, 1989.

[22] Andrew B. Hastings. Distributed Lock Management in a Transaction Processing Environment. In *Proceedings of the Ninth Symposium on Reliable Distributed Systems (SRDS'90)*, pages 22–31, 1990.

[23] HDTX. https://github.com/CGCL-codes/HDTX, 2025.

[24] Jiamin Huang, Barzan Mozafari, Grant Schoenebeck, and Thomas F. Wenisch. A Top-Down Approach to Achieving Performance Predictability in Database Systems. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD'17)*, pages 745–758, 2017.

[25] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *Proceedings of the 2010 USENIX Annual Technical Conference (USENIX ATC'10)*, 2010.

[26] What is Persistent Memory Over Fabric? https://www.intel.cn/content/www/cn/zh/developer/videos/what-is-persistent-memory-over-fabric.html, 2025.

[27] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Design Guidelines for High Performance RDMA Systems. In *Proceedings of the 2016 USENIX Annual Technical Conference (USENIX ATC'16)*, pages 437–450, 2016.

[28] Anuj Kalia, Michael Kaminsky, and David G. Andersen. FaSST: Fast, Scalable and Simple Distributed Transactions with Two-Sided (RDMA) Datagram RPCs. In *Proceedings of the 13th USENIX Symposium on Networked Systems Design and Implementation (OSDI'16)*, pages 185–201, 2016.

[29] Antonios Katsarakis, Vasilis Gavrielatos, M. R. Siavash Katebzadeh, Arpit Joshi, Aleksandar Dragojevic, Boris Grot, and Vijay Nagarajan. Hermes: A Fast, Fault-Tolerant and Linearizable Replication Protocol. In *Proceedings of the 2020 International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'20)*, pages 201–217, 2020.

[30] Antonios Katsarakis, Yijun Ma, Zhaowei Tan, Andrew Bainbridge, Matthew Balkwill, Aleksandar Dragojevic, Boris Grot, Bozidar Radunovic, and Yongguang Zhang. Zeus: locality-aware distributed transactions. In *Proceedings of the Sixteenth European Conference on Computer Systems (EuroSys'21)*, pages 145–161, 2021.

[31] Daehyeok Kim, Amirsaman Memaripour, Anirudh Badam, Yibo Zhu, Hongqiang Harry Liu, Jitu Padhye, Shachar Raindel, Steven Swanson, Vyas Sekar, and Srinivasan Seshan. Hyperloop: Group-based NIC-offloading to Accelerate Replicated Transactions in Multi-tenant Storage Systems. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM'18)*, pages 297–312, 2018.

[32] Leslie Lamport. A New Solution of Dijkstra's Concurrent Programming Problem. *Communications of the ACM*, 17(8):453–455, 1974.

[33] Yuliang Li, Gautam Kumar, Hema Hariharan, Hassan M. G. Wassel, Peter Hochschild, Dave Platt, Simon L. Sabato, Minlan Yu, Nandita Dukkipati, Prashant Chandra, and Amin Vahdat. Sundial: Fault-tolerant clock synchronization for datacenters. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI'20)*, pages 1171–1186, 2020.

[34] Haodi Lu, Haikun Liu, Chencheng Ye, Xiaofei Liao, Fubing Mao, Yu Zhang, and Hai Jin. Software-Defined, Fast and Strongly-Consistent Data Replication for RDMA-Based PM Datastores. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS'23)*, pages 90–101, 2023.

[35] David T. McWherter, Bianca Schroeder, Anastassia Ailamaki, and Mor Harchol-Balter. Priority Mechanisms for OLTP and Transactional Web Applications. In *Proceedings of the 20th International Conference on Data Engineering (ICDE'04)*, pages 535–546, 2004.

[36] Sundeep Narravula, A. Marnidala, Abhinav Vishnu, Karthikeyan Vaidyanathan, and Dhabaleswar K. Panda. High Performance Distributed Lock Management Services using Network-based Remote Atomic Operations. In *Proceedings of the Seventh IEEE International Symposium on Cluster Computing and the Grid (CCGrid'07)*, pages 583–590, 2007.

[37] Thomas Neumann, Tobias Mühlbauer, Kemper, and Alfons. Fast serializable multi-version concurrency control for main-memory database systems. In *Proceedings of the 2015 ACM International Conference on Management of Data (SIGMOD'15)*, page 677–689, 2015.

[38] Waleed Reda, Marco Canini, Dejan Kostic, and Simon Peter. RDMA is Turing complete, we just did not know it yet! In *Proceedings of 19th USENIX Symposium on Networked Systems Design and Implementation (NSDI'22)*, pages 71–85, 2022.

[39] FaRM-Microsoft Research. https://www.microsoft.com/en-us/research/project/farm/, 2025.

[40] Henry N. Schuh, Weihao Liang, Ming Liu, Jacob Nelson, and Arvind Krishnamurthy. Xenic: SmartNIC-Accelerated Distributed Transactions. In *Proceedings of the 28th Symposium on Operating Systems Principles (SOSP'21)*, pages 740–755, 2021.

[41] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. LegoOS: A Disseminated, Distributed OS for Hardware Resource Disaggregation. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI'18)*, pages 69–87, 2018.

[42] Junyi Shu, Ruidong Zhu, Yun Ma, Gang Huang, Hong Mei, Xuanzhe Liu, and Xin Jin. Disaggregated RAID Storage in Modern Datacenters. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'23)*, pages 147–163, 2023.

[43] InfiniBand Architecture Specification. https://www.infinibandta.org/ibta-specification/, 2024.

[44] Shin-Yeh Tsai, Yizhou Shan, and Yiying Zhang. Disaggregating Persistent Memory and Controlling Them Remotely: An Exploration of Passive Disaggregated Key-Value Stores. In *Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC'20)*, pages 33–48, 2020.

[45] Cheng Wang, Jianyu Jiang, Xusheng Chen, Ning Yi, and Heming Cui. APUS: fast and scalable paxos on RDMA. In *Proceedings of the 2017 Symposium on Cloud Computing (SoCC'17)*, pages 94–107, 2017.

[46] Tianzheng Wang and Hideaki Kimura. Mostly-optimistic concurrency control for highly contended dynamic workloads on a thousand cores. *Proc. VLDB Endow.*, 10(2):49–60, 2016.

[47] Yandong Wang, Xiaoqiao Meng, Li Zhang, and Jian Tan. C-hint: An Effective and Reliable Cache Management for RDMA-accelerated Key-value Stores. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC'14)*, pages 1–13, 2014.

[48] Xingda Wei, Zhiyuan Dong, Rong Chen, and Haibo Chen. Deconstructing RDMA-enabled Distributed Transactions: Hybrid is Better! In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI'18)*, pages 233–251, 2018.

[49] Xingda Wei, Jiaxin Shi, Yanzhe Chen, Rong Chen, and Haibo Chen. Fast in-memory transaction processing using RDMA and HTM. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP'15)*, pages 87–104, 2015.

[50] Xingda Wei, Xiating Xie, Rong Chen, Haibo Chen, and Binyu Zang. Characterizing and optimizing remote persistent memory with RDMA and NVM. In *Proceedings of the 2021 USENIX Annual Technical Conference (USENIX ATC'21)*, pages 523–536, 2021.

[51] Dong Young Yoon, Mosharaf Chowdhury, and Barzan Mozafari. Distributed Lock Management with RDMA: Decentralization without Starvation. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD'18)*, pages 1571–1586, 2018.

[52] Ming Zhang, Yu Hua, and Zhijun Yang. Motor: Enabling multi-versioning for distributed transactions on disaggregated memory. In *Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI'24)*, pages 801–819, 2024.

[53] Ming Zhang, Yu Hua, Pengfei Zuo, and Lurong Liu. FORD: Fast One-sided RDMA-based Distributed Transactions for Disaggregated Persistent Memory. In *Proceedings of the 20th USENIX Conference on File and Storage Technologies (FAST'22)*, pages 51–68, 2022.

[54] Mingxing Zhang, Teng Ma, Jinqi Hua, Zheng Liu, Kang Chen, Ning Ding, Fan Du, Jinlei Jiang, Tao Ma, and Yongwei Wu. Partial Failure Resilient Memory Management System for (CXL-based) Distributed Shared Memory. In *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP'23)*, pages 658–674, 2023.

[55] Yang Zhou, Xingyu Xiang, Matthew Kiley, Sowmya Dharanipragada, and Minlan Yu. DINT: fast in-kernel distributed transactions with ebpf. In *Proceedings of the 21st USENIX Symposium on Networked Systems Design and Implementation (NSDI'24)*, pages 401–417, 2024.

Cuckoo for Clients: Disaggregated Cuckoo Hashing

Stewart Grant and Alex C. Snoeren
UC San Diego

Abstract

RCuckoo is a fully disaggregated lock-based key/value store in which clients cooperatively access a passive memory server using exclusively one-sided RDMA operations. RCuckoo employs cuckoo hashing to enable single round-trip reads of small values while updates and deletes require only two. We introduce locality-enhanced dependent hashing that allows us to adjust the expected distance between a key's potential table locations, dramatically improving insert performance compared to prior cuckoo-hashing approaches while limiting I/O amplification and maintaining practical maximum fill factors. We show that not only does RCuckoo outperform all existing state-of-the-art RDMA-based key/value stores when reading small values, but under severe contention RCuckoo delivers up to 7× the throughput of comparison systems across the standard set of YCSB workloads. Moreover, RCuckoo's lease-based locking mechanism enables it to gracefully recover from 100s of client failures per second.

1 Introduction

Disaggregated architectures aim to improve scalability and utilization by pooling network-attached resources [3, 17, 21, 35]. In theory, pooling reduces per-machine fragmentation and stranding by exposing resources over the network. Collectively, disaggregated resources can be provisioned for the sum-of-peak workloads rather than the peak-of-sums [2, 36]. Stagnating DRAM densities are driving interest in primary storage pooling, but performant, general-purpose far-memory tiers remain elusive [1, 4, 5, 11, 18, 24, 35, 42] as even the fastest rack-scale networks have access latencies that are an order-of-magnitude slower than local memory. (e.g., RDMA latency is approximately 1 µs versus DRAM's 50 ns) [40].

As a result, most existing far-memory systems [5, 12, 34, 39, 44] choose to statically partition remote memory to avoid the multiple round trips required to synchronize shared updates. While challenging to achieve, memory sharing is critical to making remote memory pooling practical as it can improve utilization—the entire point of memory disaggregation. To date, the most promising approach to delivering shared access to pooled remote memory is to expose a key/value store (KVS) interface that provides coherent read/write semantics [19, 22, 37, 38, 40, 41, 43]. Yet, most performant KVS systems employ two-sided RDMA operations which rely upon memory-side CPUs to manage locks and execute critical sections [9, 20, 25, 26], reducing their potential savings.

To fully unlock the potential of memory disaggregation, we propose a design where compute and memory are entirely separate. In our fully disaggregated model, a KVS must implement a client-side serialization protocol implemented entirely in one-sided RDMA operations (i.e., read, write, and atomic updates). Maintaining a coherent memory model is the core challenge in designing such systems. Existing KVS's with ordered keys (e.g., BTrees & Radix Trees) rely upon locking schemes [22, 41], while those that provide unordered access favor lock-free techniques [19, 37, 38, 40] to decrease read latency while avoiding the RDMA hardware bottlenecks and complex failure scenarios that can arise with locks. The trade-off, however, is that lock-free approaches perform worse as write intensity increases.

In this work we present RCuckoo, a system that uses lock-based synchronization on commodity RDMA hardware to out-perform existing key/value stores across a wide range of workload mixes. Critically, locks enable the use of cuckoo hashing [32], a well-established, high-performance datastructure for key/value stores. Locks also support in-lining small values in the hash table, delivering significantly higher throughput on common mixed read/write workloads. Further, our fine-grained locks and careful protocol design reduce lock hold times during updates to two round trips in the common case, and we avoid known hardware bottlenecks with RDMA atomic operations by crafting a lock table small enough to be stored entirely on NIC-based memory. Finally, lease-based lock acquisition delivers correct, high-throughput performance even in the face of client failures with held locks.

RCuckoo's efficient locking is made possible by a *dependent hashing* algorithm that makes spatial locality a

tunable parameter. Cuckoo hashing works by deterministically computing two potential hash locations—primary and secondary—for any key; collisions are resolved by relocating the existing entry in a key's primary location to the existing key's secondary location, and so on down what is known as a cuckoo path. Traditionally, a key's two potential locations are drawn independently at random, defeating any locality-based optimizations. By controlling the distance between a key's locations—and therefore the span of a potential cuckoo path—we probabilistically bound the range of memory that must be locked, and, relatedly, the number of locks that a client must acquire.

Combined with aggressive batching of RDMA operations, RCuckoo's spatial locality limits the number of round trips required for all table operations. In the common case, reads execute in one (for small values) or two round trips, uncontested updates and deletes require two round trips, and the median insert operation involves only two round trips—although the expected number increases as the table fills and cuckoo paths grow. On our testbed, RCuckoo delivers comparable or higher performance on small values across the standard set of YCSB benchmarks than all of the existing disaggregated key/value stores we consider. Concretely, with 320 clients RCuckoo delivers up to a 2.5× throughput improvement on read-intensive (YCSB-B) workloads and up to 7.1× their throughput on write-intensive (YCSB-A) workloads. Moreover, RCuckoo's performance remains high despite 100s of clients failing per second.

2 Background

Existing disaggregated key/value stores build upon the extensive literature of RDMA-based key/value stores. In this section we overview that lineage and discuss the algorithmic distinctions between the two. Our hash table design draws on both cuckoo and hopscotch hashing. We provide a brief overview of both approaches and related literature.

2.1 Disaggregated Key/Value Stores

The challenge to adapting existing RDMA-based KVS designs to a fully disaggregated setting is the need to rely exclusively on one-sided operations due to the lack of a server-side CPU. Precursor RDMA KVS systems achieve performance by striking a delicate balance between efficient one-sided RDMA operations and CPU-serialized two-sided operations [8, 15, 25, 26, 28], where the later are used sparingly, but are critical for correctness.

In general, the performance of any KVS on a mixed read/write workload hinges on its serialization performance. Fast consistent writes with one-sided RDMA are hard because RDMA atomic operations operate on small (8-byte) regions of memory and the cost of round trips is high. This constraint has lead to a divide in the design of disaggregated KVS systems between those that support range queries over ordered keys and those that do not. Systems with ordered keys use locks to guard their complex updates, while those with unordered keys employ lock-free optimistic approaches by constraining updates to atomic 8-byte writes. These latter designs typically use hash indexes with 8-byte entries that point to uncontested extent regions where the corresponding values are stored. Commonly the index entries use 48 of the 64 bits as a pointer and the remaining 16 bits for a digest of the key and the size of the value. This approach requires additional round trips to check for a key's presence and limits the size of values that can be stored [38, 45].

On the other hand, KVS systems that support ordered keys use locks to guard the complex operations required to update B-Trees or Radix Trees. In exchange for expensive update operations, locks allow these systems to support inlined values and, thus, faster small-value reads. Performance is gated, however, by lock granularity and hold times. At the time of writing all lock-based KVS systems assume that clients are grouped into co-located servers which can locally coordinate lock accesses and batch their writes [22, 41], an assumption that optimistic KVS systems do not make [37, 38, 40, 45]. We show that it is possible to realize the performance gain of lock-guarded inlined operations without requiring client co-location if locks are sufficiently fine grained and hold times limited.

2.2 RDMA and Network Performance

Historically, network bandwidth has limited KVS throughput more than operation rate. For instance, while a 40-Gbps ConnectX-3 NIC can process 75 million packets per second, line rate restricts it to 5 MOPS when reading 1-KB objects. As commodity datacenter link rates increase from 100 Gbps to 400 Gbps and beyond, however, latency and contention—rather than raw network bandwidth—become the primary scalability bottlenecks. To this end our design choices seek to reduce round trips at the cost of slight bandwidth amplification and we focus our evaluation on workloads with small key/value pairs whose performance is not limited by line rate on our 100-Gbps ConnectX-5 testbed hardware.

Concretely, Figure 1(a) shows the growth in round trip time for RDMA reads of increasing size. On our hardware, the round trip time for a single 1-KB read is lower than the total latency of issuing two dependent reads of

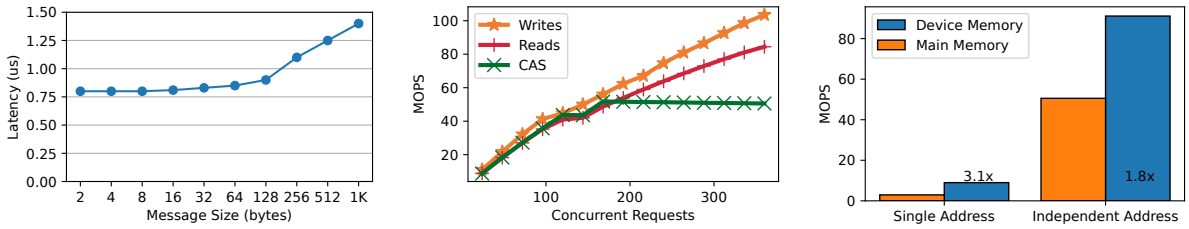


Figure 1: Basic RDMA performance on our testbed. **(a)** Read latency as a function of message size [31]. **(b)** Operation throughput as a function of offered load. **(c)** Atomic compare-and-swap performance on device and main memory.

just a few bytes each. This suggests that performance can be gained if a single large packet can complete the work of two smaller, but dependent messages. Indeed, our evaluation (see Figure 8(d)) shows that even on our 100-Gbps testbed inlining values and servicing lookups with a single large read provides a 4–37% throughput increase compared to an extent-based approach for the same value size.

Prior systems have avoided locks due to the stark performance penalties of RDMA atomics [16]. Figure 1(b) illustrates this bottleneck on ConnectX-5 NICs when atomic operations are issued on remote server memory: reads and writes scale almost linearly while atomics reach a hard limit around 50 MOPS. Using RDMA atomic operations to spin on locks quickly induces these bottlenecks as the cap is even lower (3 MOPS as shown in Figure 1(c)) for atomics that contend with each other (i.e., access the same remote address). Recent ConnectX series NICs sport a small amount (256 KB) of on-NIC memory which avoids a PCIe round trip and raises the cap on atomic operation throughput [30, 41]. Figure 1(c) illustrates the performance improvement for single—i.e., contended—and independent addresses. If a system can arrange to store its locks exclusively in NIC memory it can gain up to a 3× performance improvement on contended workloads.

Others have explored ways to improve RDMA-based lock performance. Citron implements a general purpose, range-based lock table with fairness guarantees [10]. While the current design is ill-suited for our use case (each request solves a knapsack problem, maintains a tree structure, and can introduce false contention), we hope to adapt its fair bakery algorithm to RCuckoo in future work. At present, RCuckoo focuses on raw, common-case performance, supporting concurrent operations across disjoint ranges as described in the following section.

2.3 Cuckoo and Hopscotch Hashing

Both cuckoo and hopscotch hashing have been highly successful as indexes for RDMA KVS [8, 9, 14, 20, 25]. Both algorithms enable clients to locally calculate a key's location (or neighborhood) in the index without consulting the server. This fact enables a critical division of labor for mixed read and write workloads. Reads can be performed asynchronously via one-sided RDMA, while writes require serialization—typically implemented using locks managed by the CPU at the memory server using two-sided RDMA. Both cuckoo and hopscotch hashing algorithms have complex mechanisms for resolving hash collisions which make them difficult to implement entirely with one-sided RDMA. This section describes both algorithms and their associated challenges.

Cuckoo hashing uses independent hash functions to compute a primary and secondary location for each key. Keys are always inserted into their primary location and evict an existing key to their secondary on collision which in turn can cause a chain of evictions (a cuckoo path) shown in Figure 2. Associative locations and BFS search are used to minimize path length [9, 20] which is the largest factor in the cost of insertions. In a fully disaggregated implementation each path step incurs a read to a random index location which requires an RDMA round trip (as the lack of locality makes client-side caching infeasible). The lack of locality similarly frustrates attempts to proactively acquire locks. That said, cuckoo hashing does not require any metadata to be maintained in the index, simplifying collision and error recovery.

Hopscotch hashing, on the other hand, delivers a tunable level of locality for insertions, but at the cost of maintaining index metadata. In hopscotch hashing, all insertions occur in a neighborhood of their original hash location [8, 13]. Evictions are performed by "hopscotching" entries through their neighborhood until an open location is found. Each entry has a bitmask which tracks its collisions. When hopscotching these bitmasks must be updated which requires additional locking. Farm [8] and Reno [14] bound this expense by setting hard limits on chain length but still require a server-side CPU to fix bitmasks when concurrent inserts execute. Hopscotch hashing trades locality for additional metadata, while cuckoo hashing eschews metadata at the cost of random accesses. Our system RCuckoo combines the advantages of both ap-

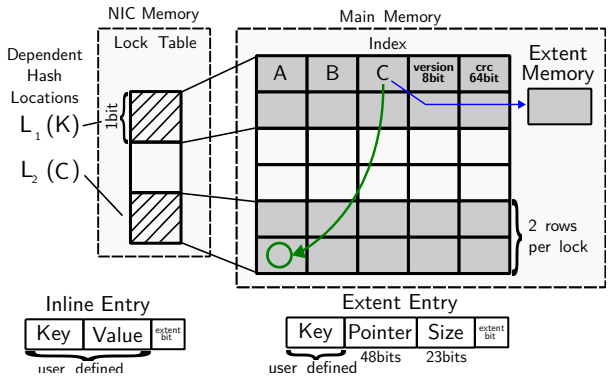


Figure 2: RCuckoo's datastructures showing insertion of key K as it displaces C, whose value is stored in an extent.

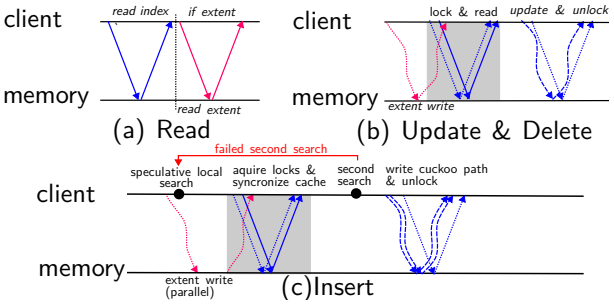


Figure 3: RCuckoo's operation protocol. Blue lines are index accesses, and red lines are extent accesses. Solid lines are reads, dotted lines are CAS, and curved dashed lines are writes. Overlapping messages indicate batching.

proaches with a cuckoo hash function with hopscotch-like locality properties in the common case.

3 Design

In this section we describe the design of RCuckoo, a fully disaggregated key/value store implemented as a lock-based cuckoo hash table. RCuckoo's performance stems largely from its extremely compact lock table which is enabled by our dependent hashing algorithm. Before describing the hashing algorithm or lock table, however, we first introduce the datastructures and protocols to read and modify the contents of the key/value store. For simplicity, we describe our design in the context of a single server, but it is straightforward to shard a large table across multiple servers (with a minor tweak, see Section 3.3).

3.1 Datastructures

Figure 2 shows RCuckoo's index and lock table (both maintained at the remote memory server) during an insertion of key K (which requires the cuckooing of key C). The index table (right) is a single region of RDMA-registered main memory divided into rows of fixed-width entries. Each row contains n associative entries (3 in this figure; we use 8 in practice) and terminates with an 8-bit version number and 64-bit CRC (computed over the entire row including version number). Clients access the index table using 1-sided RDMA reads and writes. Clients can safely read one or more rows in a single lock-free RDMA operation as CRCs allow each row's contents to be verified while version numbers enable clients to detect if a row has been modified (used to detect failures; see Section 4.1).

Table entry and key sizes are configuration parameters, but are the same for all rows and must be fixed at table creation. Individual entries, on the other hand, can contain either inlined key/value pairs (bottom left) or a key and 48-bit pointer to an extent to store larger values (bottom right). The least-significant bit of an entry signifies its type; values are inlined by default. Extent entries use 23 bits to encode the value size (which can range from 2^3 to 2^{26} bytes). Extents are located in separate, pre-allocated, per-client, RDMA-registered regions of server memory to avoid contention on inserts. Given the fully-disaggregated context we assume the index table will be initially provisioned at maximum size; we defer resizing to future work.

Locks (stored in a bit vector in NIC memory, shown on the left) each protect a tunable number (here, two; 16 in our experiments) of index rows. Clients perform lock acquisition and release with RDMA compare-and-swap (CAS) operations. Specifically, RCuckoo leverages masked CAS (MCAS) operations [29, 41] to obtain up to 64 locks simultaneously while avoiding false sharing.

3.2 Operations

We detail the operations supported by RCuckoo below; Figure 3 visualizes the corresponding message exchanges.

3.2.1 Reads

RCuckoo is designed to facilitate lock-free, single round-trip reads for small values as they are the dominant operation for key/value stores in many data centers [6, 27]. To read the value associated with a given key clients calculate the potential table locations for the key's entry (using the hash functions described in the next subsection) and issue RDMA reads for both rows simultaneously. Because all operations between a client and a given server travel over a reliable connection, they are intrinsically ordered, but in our description we will only call out when a particular ordering among a batch of messages is required.

Moreover, as suggested by Figure 1(a), if the rows are located sufficiently close together, it can be beneficial for the client to issue a single *covering read* that returns the contents of both rows—as well as any intervening ones—in a single request. In our experiments RCuckoo clients issue a single, large read rather than two small reads if the locations are in adjacent rows. Reads are successful if either row contains an entry with the desired key and a valid CRC. An invalid CRC indicates a torn write or rare failure case, in which case the operation is retried (see Section 4.1). As shown in Figure 3(a) successful inlined reads complete in one round trip, while reads for large values require a second round trip to retrieve the extent.

3.2.2 Updates and deletes

Updates and deletes, like reads, access only two locations in the index table, but require a client to acquire the associated locks. Due to RCuckoo's dependent hashing, it is usually possible to attempt to acquire both locks in a single MCAS operation (Section 3.4.1). If so, the client issues read(s) for the corresponding rows of the index table immediately afterwards but in the same batch of operations.[1] In the rare case that the locks must be acquired independently—necessitating an additional round trip—the reads are batched with the second lock request.

Assuming successful lock acquisition and valid reads, the operation can proceed if the key is present in either location. In a single (ordered) batch of operations, the client first writes the updated/freed entry and recomputed row version and CRC before releasing the locks. When updating values stored in extents, clients store the value to a new extent via an RDMA write that is sent in parallel with lock requests. On lock release clients write the first bit of the old extent to free it. Deletes operate identically save writing a new extent. Clients garbage collect their own extents by occasionally scanning their allocated region for freed extents. Figure 3(b) shows that most uncontested operations complete in two round trips; clients retry acquisitions until they succeed or detect a failed client.

3.2.3 Inserts

Inserts are challenging because concurrent operations might result in cuckoo paths that collide. To avoid the complexity of rolling back partially completed inserts in the event of a collision, RCuckoo clients compute a complete cuckoo path ahead of time and then acquire locks on all the relevant rows to ensure its success. Moreover, to facilitate recovery from client failures, an insert is performed by cuckooing elements one at a time, starting by moving the last entry in the path to the empty location, and then replacing it with the previous entry in the path, and so on until the new entry is inserted in its primary location.

To speed up cuckoo-path searches, RCuckoo clients keep a local, RDMA-registered cache of (relevant portions[2] of) the index table. The cache is populated with the results of any operation, but is used only for inserts. Clients validate—and, if necessary, update—their cache at each step of the insert operation as explained below, so stale entries do not impact correctness, only performance.

At a high level an insert operation proceeds in three (or four) phases. For extent entries, clients first write the value to a free extent—in parallel with the remaining three phases. Clients maintain a local slab allocator that manages their private extent region, so there is no contention. Regardless of whether the entry contains an inlined value or a pointer to an extent, RCuckoo clients start by identifying a potential cuckoo path using only the contents of their local table cache. Clients then simultaneously attempt to acquire the locks for and update their cache of the rows that comprise the candidate path. Using only the contents of their newly updated local cache, clients conduct a second search to confirm that a candidate path—either the initial guess or an alternative that similarly consists only of currently locked rows—exists. If so, the insert is performed; if not, the client releases its locks and retries.

Speculative local search: Each insertion attempt uses a heuristic search to find a *speculative* cuckoo path. An attempt starts by identifying a potential path using the (potentially stale) contents of the client's local table cache. We use breadth-first search (BFS) [20] to identify short paths in an attempt to minimize bandwidth and locking overhead. If the client cache is empty the degenerate path is presumed, i.e., that the key will be inserted into its primary location without the need for any cuckooing. Speculative cuckoo paths are especially useful when client caches are fresh—often due to a failed prior attempt to insert the same key.

Cache synchronization: Armed with a speculative cuckoo path, clients identify the set of locks necessary to protect the relevant rows. Approximately 99% of paths can be locked with a single MCAS operation (Figure 4(c)); longer paths acquire locks in groups (Section 3.4.1). Immediately after, but in the same batch of RDMA operations as each attempt to acquire (a subset of) the locks, clients synchronize their local cache by issuing

[1] Because the lock table is located in NIC memory, RCuckoo clients can employ `SEND_FENCE` on reads batched with lock acquisitions to ensure consistency without incurring a performance penalty.

[2] A small cache suffices; we use 64 KB in our experiments. Caching the entire index yields negligible additional benefit.

reads for all of the rows covered by that set of locks. In general, locks cover multiple rows, so this will be a superset of the rows necessary for the identified path. Note that if lock acquisition is successful, the values returned by the read of the corresponding rows will remain unchanged until the lock is released.

Figure 2 shows an example insert operation where the primary row for K is full and the entry for key C is being evicted to its secondary location. Hence, the client has acquired locks corresponding to the row where it hopes to insert the entry for key K as well as the row into which it plans to cuckoo the existing entry for key C. Rows shaded in gray are synchronized because they are covered by the locks, while the contents of any other rows that happen to be in the client's cache cannot be depended upon without additional validation. Rather, they may increase the likelihood that a subsequent speculative search succeeds.

Second search: If all the lock acquisitions are successful, the client confirms that the speculative path remains valid. Under an insert-heavy workload, however, speculative cuckoo paths are frequently stale. Yet, a valid cuckoo path may still exist within the locked rows. Hence, if the speculative path is no longer viable, clients perform a second search within their cache restricted to only the rows for which they currently hold the lock in hopes of identifing an alternative path. If the speculative path works—or a valid alternative is located—the appropriate series of swaps and version/CRC updates are calculated and issued as a batch of RDMA writes, one row at a time, followed by (an ordered set of) lock releases. If no valid path exists within the currently locked rows the client releases its locks and tries again, conducting another speculative search on the updated cache contents.

This entire process repeats until success, a client determines there is no viable cuckoo path within a maximum search depth (we use a depth of five in our experiments, at which point the insert operation returns an error indicating the table is full), or a failed client is detected. If cuckoo paths were to randomly span the table it is unlikely that an alternate valid path would exist within the locked rows when speculation fails. In the next subsection, we describe how RCuckoo uses dependent hashing to dramatically increase the likelihood that an alternate path exists.

3.3 Locality

In a traditional cuckoo hash, the two locations for a given key are deliberately independent which allows the table to be filled quite full before inserts begin to fail. In RCuckoo the distance between keys' two cuckoo hash locations is a tunable parameter. Increased locality has two direct benefits: it decreases the number of MCAS operations necessary to acquire the relevant locks and increases the probability that both of a keys' locations can be read with a single covering read. It also reduces the region of the index table likely to be spanned by cuckoo paths, which speeds up inserts, but leads to hot spots that limit the table's expected maximum fill factor. Experiments show that an optimal locality setting can dramatically decrease the number of round trips required to perform inserts in RCuckoo while maintaining high (90%+) maximum fill factors.

In RCuckoo, the primary location for a key is chosen uniformly at random, while the second is offset from the first by uniformly random value drawn from a probabilistically bounded range, where the range is likely to be relatively small. We start with a base hash[3], $h()$, and use it to implement three independent hash functions $h_1()$, $h_2()$, and $h_3()$. (In our implementation we use a different salt for each of the three functions.) We compute the two locations L_1 and L_2 for a key K as

$$L_1(K) = h_1(K) \mod T,$$

$$L_2(K) = L_1 + (h_2(K) \mod \lfloor f^{f+\mathcal{Z}(h_3(K))} \rfloor) \mod T,$$

where T is the size of the index table in rows, $\mathcal{Z}(x)$ is the number of trailing zeros in x, and f is a parameter that controls the expected distance between the two hash locations. (In a sharded deployment, the second location is restricted to the same shard by "wrapping around" the offset accordingly.) The particular formulation is not important, but the upshot is exponentially fewer keys have secondary locations at increasing distances from their primary. The probabilistic aspect is crucial, as any fixed bound on the distance between hash locations leads to low maximum fill factors (on the order of 10–15% in our experiments).

Figure 4(a) shows the distance between hash locations as a CDF for different values of f, while Figure 4(b) shows that larger f enables higher fill prior to the first insertion failure. In our evaluation we set $f = 2.3$ based on this empirical data. As shown in the figure, for index tables with eight entries per row, RCuckoo delivers an expected max fill of greater than 95% with a 68% probability that a key's locations are located five or fewer rows apart.

Decreased distances between hash locations naturally lead to shorter cuckoo paths when combined with our breadth-first search approach. Using $f = 2.3$ and a table size of 100-K rows, Figure 4(c) shows that when filling the table to 95% full, slightly more than half of insertions do not require any cuckooing, and 95% of insertions require cuckoo paths that span 32 or fewer rows while

[3] We use xxHash [7] in our implementation.

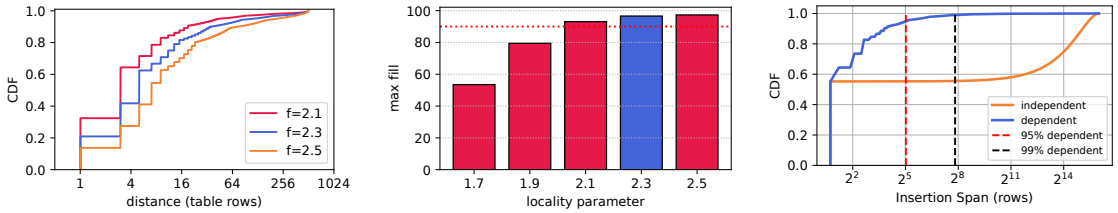


Figure 4: **(a)** CDF of distances between cuckoo locations for different locality settings using RCuckoo's dependent hashing. **(b)** Fill percentage before insertion failure for different locality settings (average of 10 runs, negligible variance). 90% fill indicated by dashed red line. **(c)** CDF of cuckoo spans for dependent and independent hashing. A cuckoo span is the distance between the smallest and largest index in a cuckoo path.

nearly 99% span 256 or fewer. Conversely, with independent hashing, insertions that require any cuckooing at all almost always result in spans of 2 K rows or more.

3.4 Locking

While RCuckoo reads are lock free, updates and especially inserts depend critically on locking performance.

3.4.1 Lock granularity

Increased locality decreases the number of round trips required for lock acquisition. Recall the lock table is a linear array of lock bits and each bit locks one or more table rows. As mentioned previously, RCuckoo implements lock acquire and release using RDMA masked compare-and-swap (MCAS) operations that can update 64 bits at a time. To avoid deadlock RCuckoo acquires locks in increasing order. For any given operation, clients group the necessary locks into the smallest number of sets as possible (where each set is an attempt to acquire one or more locks within a single 64-bit span) and issue MCAS operations one at a time in order of their target address. Clients continuously spin on lock acquisitions and only move to the next MCAS operation after the current one succeeds. Due to this one-MCAS-per-round-trip acquisition procedure, lock granularity is critical to performance.

If, as in Figure 4(c), almost 99% of cuckoo spans are 256 rows or less, and each lock protects four rows, almost all insertions can have their locks acquired with a single MCAS (lock-table accesses do need to be byte aligned). Of course, increasing the number of rows covered by a single lock can lead to false sharing, forcing additional retries to acquire the necessary locks. Figure 5 shows the results of a representative experiment where 8 clients are concurrently filling a 512-row table to 95% full. We report the 99th-percentile (i.e., the most expensive inserts when the table is nearly full) number of round trips required to acquire the locks necessary to perform an insertion as a function of both lock granularity (i.e., number of rows per lock, on the x-axis) and lock size (i.e., the number of locks that can be accessed with a single MCAS operation, on the y-axis—RCuckoo's single-bit locks correspond to 64 locks per message shown on the bottom row).

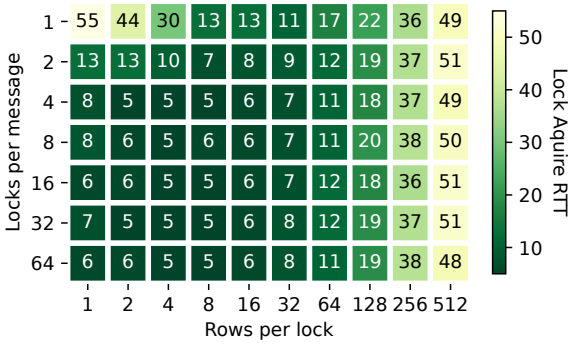


Figure 5: 99th-percentile round trips required per insert in a 512-row table when filling to 95%. 512 buckets per lock corresponds to a single global lock.

While there is some noise due to experimental variance, the far-right column shows that a single global lock results in high contention (as there is only one lock in the system, it does not matter how many locks can be acquired per message). Conversely, the top left corner shows that, despite the lack of false sharing when a lock corresponds to exactly one row, the inability to acquire more than one lock at a time leads to a large number of round trips. Under these conditions, the sweet spot falls in the range of 2–16 rows per lock. In our experiments we use 16 rows per lock as, when combined with our choice of f, a single MCAS suffices for the vast majority of insertions.

3.4.2 Virtual locks

While Figure 5 suggests that RCuckoo could employ larger locks (e.g., 8–16-bits per) without increasing the number of round trip times required to acquire them, there is an additional design constraint that drives our choice of single-bit locks. Specifically, to improve locking per-

formance, RCuckoo locates the lock table in NIC device memory which delivers $3\times$ higher throughout on contented addresses than host memory (Figure 1(c)). It is also lower latency as operations to device memory avoid a PCIe round trip. Unfortunately, NIC memory is limited (to 256 KB on our ConnectX-5s), so this choice bounds the size of the lock table and drives our single-bit design.

To allow RCuckoo to support tables with more than 64 M rows, we implement a *virtual* lock table where multiple logical locks map to a single physical lock. Concretely, we map a logical lock l drawn from a table of size L to a physical location p in a bit-array of size P by computing $p = l \mod P$. Mapping multiple virtual locks to a single physical lock introduces yet another source of false sharing, but allows us to support arbitrarily large tables. When employing virtual locking, clients performing an insert first map rows to virtual locks, then to physical locks, then sort them into groups.

4 Fault tolerance

In keeping with the rest of its design, RCuckoo handles failures in a fully disaggregated manner as well. We depend upon the RDMA hardware to handle network failures and focus exclusively on clients which can *fail-stop* mid-operation. Server failure can be addressed by employing client-driven replication on top of RCuckoo. While there may be opportunities to integrate replication into RCuckoo itself, we defer such an exploration to future work. In RCuckoo, client failure is only of concern if the failure occurs while the client was holding one or more locks, i.e., in the middle of a mutating operation (i.e., update, delete, or insert); hence, RCuckoo detects client failures by noticing that a pending mutation does not complete in a timely fashion. Any client that encounters such a situation endeavors to recover the stranded lock and repair the impacted portion of the index table. The remainder of this section describes how RCuckoo clients detect faults, reclaim stranded locks, and, if necessary, repair the index table. Finally, we discuss additional measures that can be employed to prevent stale writes if desired.

4.1 Failure detection

Clients detect failures by setting a timeout when attempting repeated lock acquisition or read requests. Because RCuckoo operations are designed to require only a few round trip times, a client performing a successful mutating operation will complete and release its locks extremely rapidly. Conversely, a client that is unable to acquire all the locks required for an insert operation releases those they do hold before trying again. Hence, it is extremely unlikely that repeated attempts to acquire a lock or perform an untorn read will fail continuously.

Of course, there is a possibility that a given row is highly popular, leading to high lock contention and/or repeatedly torn reads. Clients distinguish this case by consulting the CRC for the row they are unable to successfully read or lock. Because each mutation increases the version number, even updates that replace an entry with the same value will result in a different CRC. Clients declare a false positive and restart their failure timer if a CRC changes between attempts.

We expect client failures to be relatively rare, so set our fault timeout conservatively. Failure timers must allow for worst-case locking time, second-search time, and RDMA message transmission time. We bound locking time; search and message propagation are both measured in single-digit microseconds on our testbed. To guard against the possibility that network conditions lead to high rates of RDMA retries we set the maximum RDMA operation retry number to three. In this context, we set the failure timeout to 100 ms in our experiments, orders of magnitude above the 99th-percentile insert time of 50 μs.

4.2 Repair leases

RCuckoo recovers stranded locks one lock at a time; if a client fails holding multiple locks recovery may be conducted by multiple clients at different times depending on their access patterns. RCuckoo's lock table does not maintain records of ownership, so there is no way to "transfer" lock ownership from the failed node to a recovery node in the table itself. Instead, clients acquire a *repair lease* that grants exclusive permission to reclaim locks on a region of the index table. The index table is broken into n regions so repairs can be executed in parallel.

RCuckoo maintains a lease table in RDMA-registered main memory on the server. Lease entries contain a $(\log n)$-bit lease ID, a set flag, the lease holder's 48-bit queue pair ID (which RDMA ensures is unique for a given server), and an 8-bit counter (incremented on each acquisition). A lease is considered free if the set bit of the current entry in the lease table is zero. Clients attempt to acquire the lease using RDMA CAS operations to ensure mutual exclusion. Upon successful acquisition, a client completes the repair (described below) and then relinquishes the lease by clearing the set bit. Leases are revoked (to handle the case of a failed recovery node) using a timeout mechanism similar to normal locks. If a client times out while attempting lease acquisition it claims it for itself (again, using CAS to resolve any races) and marks

the lease holder as failed (see Section 4.4).

4.3 Table repair

All modification operations write new entries as a cuckoo path; updates and deletes have a path of length one. As described in Section 2.3 cuckoo paths are executed by first claiming an open entry at the end of the path and proceeding backward along the path, cuckooing entries forward one-by-one until the new entry is written at the beginning of the path. Client failures can occur at any point along an insertion path; a failed client can leave the table in one of four distinct states based on how far along it was:

1. A duplicate entry exists and one has a bad CRC,
2. A duplicate entry exists and both have correct CRCs,
3. No duplicate exists but one row has a bad CRC, or
4. No duplicate exists and no rows have a bad CRC.

The last case can occur if a client fails prior to issuing any updates to the table or if it fails after updating all the rows but before releasing the locks. In either case recovery is trivial: a client with the repair lease can simply unlock the stranded lock. Recovery from the other three cases requires modification to the index table.

To repair the table a client first detects in which state the table is and then transitions the table forward through the states with a deterministic sequence of operations so that failures during recovery can be repaired by a subsequent client. A client determines the state by issuing reads to all rows protected by the stranded lock. It then proceeds one-by-one through each entry within the rows, checking both hash locations for the corresponding key (one of which may not be in the locked rows) for duplicates or a bad CRC. Because RCuckoo updates one row per RDMA write, there can be at most one duplicate or bad CRC.

Clients repair the table by transitioning though the states one step at a time. To move from state 1 to 2, the client writes a new CRC for the bad duplicate. The table can be transitioned from state 2 to 4 by clearing the duplicate entry in the second (i.e., pointed to by $L_2(K)$) location. Finally, the table can be transitioned from state 3 to 4 by recalculating and writing a new CRC to the impacted row. After a client has issued its repair sequence it unlocks the reclaimed lock and returns its lease.

From a correctness perspective, once all rows have a valid CRC and there are no duplicates, the table is usable again. Clearly the new value being inserted into the table by the failed client is lost, but this is indistinguishable from the case that the client failed before attempting the insert. If the client was in the middle of cuckooing values up the path, a subset of the values were moved from their primary cuckoo location to their secondary location, but reads check both locations in any case, so the entry will still be located. Finally, because duplicate entries are freed, no space in the table is lost.

4.4 Preventing stale writes

The one remaining concern is that a supposed-failed client could just be slow, and may yet attempt to complete its cuckoo path despite the fact that its locks were reclaimed. Our failure timeout is deliberately set many orders of magnitude larger than the expected operation completion time, but we cannot completely rule out the possibility. Large-scale deployments can implement a separate liveness protocol to identify stalled clients and prevent such black swan events.

Specifically, a portion of remote memory could be used to store a datastructure that each client must update with some frequency so failed (or unreasonably slow) clients can be identified by their failure to update their entry in a timely manner. Such a protocol is straightforward to implement with one-sided RDMA operations, but we have not found the need to implement it in our testbed—we have never seen a stale write that was delivered with a delay anywhere close to approaching our timeout value.

Once a failed client is identified, real-world deployments have many ways to ensure the client ceases operation, but it is interesting to consider providing such functionality within RCuckoo itself. Unfortunately, at the time of writing the Infiniband specification does not allow clients to modify each other's RDMA permissions.[4] To the best of our knowledge, the only current alternative is to reset a failed client's queue pair by crafting an invalid packet and sending it to their queue pair at the server [33, Attack 2]. For this attack to work the packet sequence number of the invalid packet must match expected sequence number at the receiver so 2^{24} packets must be sent to ensure the connection is corrupted successfully.

5 Evaluation

We evaluate RCuckoo by directly comparing its performance in terms of throughput and latency against representative state-of-the-art (partially) disaggregated key/-value stores. When accessing sufficiently large values, all systems can max-out our testbed's 100-Gbps link rate, so we focus on workloads with small key/value pairs that remove the bandwidth limitation, exposing the

[4] Type-II memory windows enable clients to remove their own permissions using `SEND_WITH_INV`, but not another client's.

(in)efficiency of each system's management and synchronization techniques.

When values are stored inline, RCuckoo outperforms existing systems while delivering competitive insert latencies. Using fault injection, we show that our distributed approach to client failure detection and recovery enables RCuckoo to sustain high throughput even though 100s of clients are failing per second. Finally, we justify our design decisions through a series of micro-benchmarks. Specifically, we quantify the benefit RCuckoo extracts from its locality enhancement before measuring the impact of index-table entry size.

5.1 Implementation

We have built two separate versions of RCuckoo, an 8.7 K-line C++ implementation tuned for high performance and a 12 K-line Python implementation that simulates RDMA operations to facilitate correctness testing. Both implementations will be made available on GitHub at publication. We report results using our C++ implementation which requires OFED-4.9 to support masked CAS operations and device-mapped memory on ConnectX-5 NICs. All experiments use 64-KB client index-table caches and a virtual lock table at the server.

5.2 Testbed

We conduct our evaluation on an 9-node cluster of dual-socket Intel machines. Each CPU is an Intel Xeon E5-2650 clocked at 2.20 GHz. Each machine has 256 GB of RAM with 128 GB per NUMA node. All machines have a single dual-port ConnectX-5 attached to a 100-Gbps Mellanox Onyx switch. In our RCuckoo experiments we use one sever as the memory server and the rest a client machines spreading threads evenly across machines.

We compare RCuckoo against three recent RDMA key/value stores with different designs, FUSEE [38], Clover [40], and Sherman [41]. While none have the exact same assumptions or feature set as RCuckoo, each represents an apt comparison point for different aspects. To avoid biasing our evaluation, we consider the same workloads (YCSB) as the authors of the previous systems.

FUSEE is a fully disaggregated key/value store that represents the closest available comparison point to RCuckoo. While both employ only 1-sided RDMA operations, FUSEE eschews locking in favor of optimistic insertions. FUSEE clients use CAS operations to manage fixed, 64-bit index table entries that contain pointers to values stored in extents. Due to its reliance on CAS operations, FUSEE is unable to support inlined storage of small values like RCuckoo, forcing all reads to require two round trips. Unlike RCuckoo, FUSEE is designed to support replication. To remove the overhead of replication, we deploy FUSEE with a single memory node.

Clover is only partially disaggregated—it requires a metadata server to manage its index structure—but can deliver higher read performance than FUSEE on read-only workloads. Clover is designed to leverage remote persistent memory and implements both reads and updates using one-sided RDMA operations. Moreover, unlike FUSEE—and similar to RCuckoo—Clover reads are self verifying. In contrast to prior comparisons [38] that force clients to consult the metadata server on each read, we allow Clover to take advantage of its client caching to achieve maximum performance on read-heavy workloads.

Sherman is the highest-throughput distributed key/-value storage system of which we are aware that employs locks. Sherman maintains a B-tree that spans multiple servers and supports range queries, a feature none of the other systems—RCuckoo included—provide. On the other hand, Sherman clusters are not fully disaggregated: each node in a cluster is a peer with many CPU cores and a single memory core that is responsible for servicing allocation RPC calls from clients. As such, Sherman does not encounter the same bandwidth bottlenecks as the other systems because requests are partitioned across machines.

5.3 Performance

We start by considering throughput and latency on the classical YCSB workloads which employ varying mixes of read and update operations before turning to the more complex insert operation. RCuckoo delivers the highest performance on reads and updates across all settings, while insert performance varies as a function of table fill factor. Even in the worst case, however, RCuckoo limits insert I/O amplification to around $2\times$.

Throughput. Figure 6 shows YCSB throughput for RCuckoo, FUSEE, Clover, and Sherman on three different YCSB workloads. For each system, we allocate a 100-M-entry table and pre-populate it with 90 M entries that each consist of a 32-bit key and 32-bit value (we consider larger sizes in Section 5.5). We plot the aggregate throughput of a variable number of clients concurrently accessing entries according to a Zipf(0.99) distribution.

In a read-only (YCSB-C) workload, FUSEE suffers from its extent-based value storage. RCuckoo, Clover, and Sherman perform similarly at low-to-moderate levels of concurrency, but they separate at scale. Sherman's read algorithm is more complex than RCuckoo's leading to lower top-end performance. Clover's client-side caching shines under this skewed workload, where almost all reads

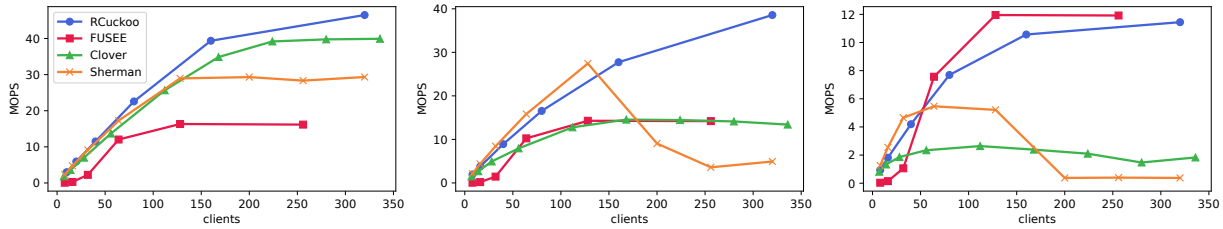


Figure 6: Throughput as a function of the number of clients for three different workloads (Zipf θ=0.99): **(a)** Read only (YCSB-C), **(b)** YCSB-B 95% read, 5% update and **(c)** 50% read 50% update (YCSB-A).

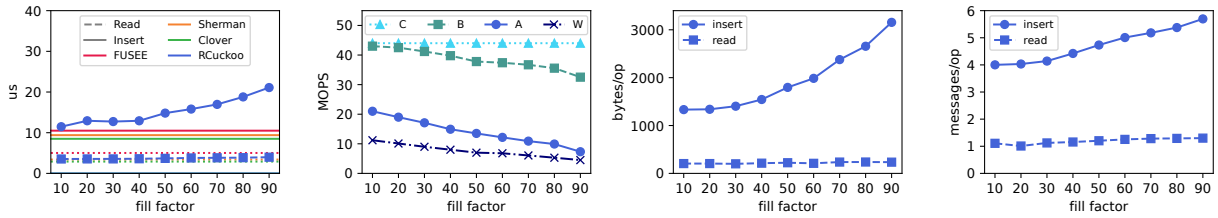


Figure 7: Insert performance as a function of fill factor. **(a)** Throughput for four different insert workloads, **(b)** median operation latency, **(c)** mean operation size, and **(d)** mean per-operation RDMA message count under workload A.

hit in a client's index cache, requiring only a single read for the value; its performance degrades under a more uniform workload (not shown). RCuckoo, on the other hand, reads inlined values in a single round trip regardless of the distribution, leading to the highest performance.

Increased update rate slows all systems. Even with only 5% updates (YCSB-B), the picture changes dramatically. Sherman performs well at low levels of concurrency due to its single-round-trip reads, but hits a severe bottleneck due to lock contention on the skewed access pattern. (Sherman improves—but does not surpass RCuckoo—for uniform workloads, not shown, where lock contention is less of an issue.) Caching is less effective with updates, bringing Clover's throughput in line with FUSEE.

On the 50/50-mixed YCSB-A workload RCuckoo and FUSEE perform similarly, although we are unable to scale FUSEE past 250 clients in our testbed while RCuckoo continues to scale. Sherman begins to suffer from lock contention even earlier, topping out around 5 MOPS before collapsing. (Absolute performance improves as Zipf skew decreases—not shown—but the trend remains.) Clover performs worst under write-heavy workloads due to its inability to effectively leverage caching with a constantly changing index structure.

Latency. Latency varies dramatically depending on operation for all systems, each of which optimizes for reads. For RCuckoo, insert is by far the most involved operation, but it remains highly performant. To evaluate insert performance we run workloads with a mix of reads and inserts. Figure 7 considers performance on workloads that exclusively use inserts (rather than updates); as with YCSB nomenclature A is 50/50 read/insert; B is 95/5; C is read-only; and W is insert only. Inserts become more expensive as the table fills, so we pre-populate the table with a varying number of entries and report insert performance as a function of the table's initial fill factor.

As a baseline we collect the read and insert latencies for all systems under light load (Figure 7(a)). Read latency is nearly identical for all systems save FUSEE as it makes a read to both the index and extent. Insert times vary: Clover and Sherman use two-sided RDMA operations for insert and both need to perform allocations and set up metadata for the requesting client. FUSEE is slightly slower, roughly the same as RCuckoo's best case. As the table fills, however, cuckoo paths grow in length causing RCuckoo insert operations to require additional round trips to find valid cuckoo paths. Insert operations at maximum fill take roughly twice as long as at empty.

Insert I/O amplification. Figure 7(b) shows impact of table fill on insert throughput at heavy load (320 clients). As the index table fills, cuckoo paths become longer leading to increased contention and additional bandwidth consumption. In each case (except read-only C) RCuckoo's performance declines with fill factor. In the insert-only W case RCuckoo's performance drops from a high of 11.5 MOPS in a nearly empty table to 4.5 MOPS at a 90% fill factor. As a point of comparison, FUSEE's maximum insert-only performance is 9.1 MOPS on our testbed, although it is independent of fill factor. While FUSEE out-performs RCuckoo at high fill factors, we observe that insert-only workloads are rare in practice [27]. Figures 7(c) and (d) show the impact of fill factor on the band-

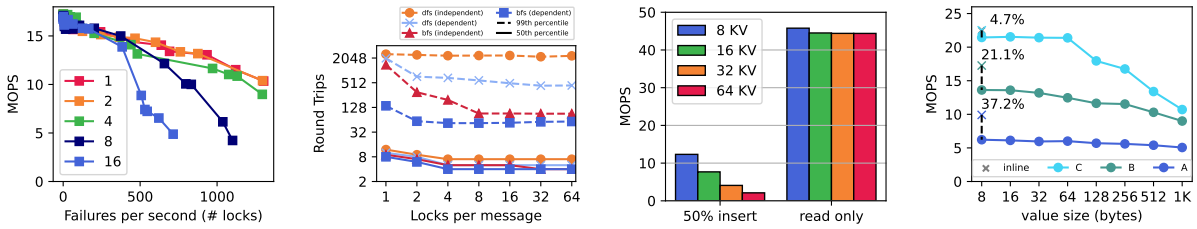


Figure 8: RCuckoo microbenchmarks: **(a)** YCSB-A throughput vs. client failure rate, **(b)** round trip times required to acquire locks on insert, **(c)** throughput vs. key/value-entry size for 50% insert 50% read, and YCSB-C (read-only) workloads, and **(d)** extent performance for value sizes up to 1 KB (with 8-byte inline performance plotted with $\times$s; relative performance improvement depicted with dashed black lines and labeled accordingly).

width cost (at most 2$\times$) of each operation and the number of RDMA messages ($\leq$1.5$\times$) they require to complete.

5.4 Fault tolerance

RCuckoo runs at nearly full throughput during realistic failure scenarios and remains functional in the face of hundreds of failures per second. We emulate client failures by performing a partial insert operation that randomly truncates the batch of RDMA operations (including lock releases), leaving the table in one of the states listed in Section 4.3. Figure 8(a) shows that throughput remains high until about 500 client failures per second, at which point lock granularity begins to play a significant role; finer-grained locks are easier to recover leading to less throughput degradation. As a point of reference, we observe that RDMA itself struggles to handle churn of this magnitude: a server can only establish approximately 1.4 K RDMA connections per second [23].

5.5 Microbenchmarks

Having established RCuckoo's superiority over prior systems and demonstrated its robustness to client failure, we now evaluate the impact of particular design choices.

Locality enhancement. Figure 8(b) illustrates the dramatic benefit RCuckoo extracts from its dependent hashing combined with a BFS cuckoo-path search strategy. To focus on longer cuckoo paths, we pre-populate a 100-M entry table to 85% and then report both the median and 99th percentile round trips per insert key/value pairs until the table is 95% full as a function of lock granularity. While median performance is on the same order, the 99th-percentile insert takes an order of magnitude fewer round trips with dependent hashing and BFS as opposed to independent hashing and DFS as used in prior cuckoo hash systems [20, 25, 32]. As before (c.f. Figure 5), performance is similar with four or more locks per message.

Entry/value sizes. Inlined key/value entries enable single-round-trip reads. However large entries increase bandwidth consumption for inserts. Figure 8(c) shows the effect of entry size on throughput under 50% insert and read-only (YCSB-C) workloads. Insert is a bandwidth-limited operation on our 100-Gbps testbed, while read performance (and update/delete, not shown) is largely unaffected by entry size. Extent entries are slightly slower: Figure 8(d) shows YCSB throughput as a function of value size from 8 to 1024 bytes on 6 client machines with 120 cores. Read performance becomes link-rate limited after about 64 bytes. For comparison, we show the performance of the 8-byte inlined values used elsewhere in the evaluation on this configuration and compute the difference. Inlined entries have two sources of performance gain: they avoid the overhead of reading and writing to extents which increases with value size, but, more importantly they avoid additional rounds trips on cache misses. YCSB-B sees a 21% performance improvement from inlining while YCSB-A gains 37% (YCSB-C has misses).

6 Conclusion

In this work we design and implement a lock-based cuckoo hash table for passive remote memory. Using dependent hashing, a carefully designed locking protocol, and the latest RDMA NIC features we achieve single-round-trip reads, update operations that require two round trips when uncontested, and insert operations that require only two round trips in the median case. Even in the face of high contention and nearly full tables, client caching limits I/O amplification. Moreover, our step-by-step insert protocol enables fully disaggregated failure detection and recovery.

References

[1] Amaro, E., Branner-Augmon, C., Luo, Z., Ousterhout, A., Aguilera, M. K., Panda, A., Ratnasamy, S., and

SHENKER, S. Can far memory improve job throughput? In *Proceedings of the Fifteenth European Conference on Computer Systems* (New York, NY, USA, 2020), EuroSys '20, Association for Computing Machinery.

[2] BANSAL, D., DEGRACE, G., TEWARI, R., ZYGMUNT, M., GRANTHAM, J., GAI, S., BALDI, M., DODDAPANENI, K., SELVARAJAN, A., ARUMUGAM, A., RAMAN, B., GUPTA, A., JAIN, S., JAGASIA, D., LANGLAIS, E., SRIVASTAVA, P., HAZARIKA, R., MOTWANI, N., TIWARI, S., GRANT, S., CHANDRA, R., AND KANDULA, S. Disaggregating stateful network functions. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)* (Boston, MA, Apr. 2023), USENIX Association, pp. 1469–1487.

[3] BIELSKI, M., SYRIGOS, I., KATRINIS, K., SYRIVELIS, D., REALE, A., THEODOROPOULOS, D., ALACHIOTIS, N., PNEVMATIKATOS, D., PAP, E., ZERVAS, G., MISHRA, V., SALJOGHEI, A., RIGO, A., ZAZO, J. F., LOPEZ-BUEDO, S., TORRENTS, M., ZYULKYAROV, F., ENRICO, M., AND DE DIOS, O. G. dredbox: Materializing a full-stack rack-scale system prototype of a next-generation disaggregated datacenter. In *2018 Design, Automation Test in Europe Conference Exhibition (DATE)* (2018), pp. 1093–1098.

[4] BRANNER-AUGMON, C., GALSTYAN, N., KUMAR, S., AMARO, E., OUSTERHOUT, A., PANDA, A., RATNASAMY, S., AND SHENKER, S. 3po: Programmed far-memory prefetching for oblivious applications, 2022.

[5] CALCIU, I., IMRAN, M. T., PUDDU, I., KASHYAP, S., MARUF, H. A., MUTLU, O., AND KOLLI, A. Rethinking software runtimes for disaggregated memory. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2021), ASPLOS '21, Association for Computing Machinery, p. 79–92.

[6] CAO, Z., DONG, S., VEMURI, S., AND DU, D. H. Characterizing, modeling, and benchmarking RocksDB Key-Value workloads at facebook. In *18th USENIX Conference on File and Storage Technologies (FAST 20)* (Santa Clara, CA, Feb. 2020), USENIX Association, pp. 209–223.

[7] COLLET, Y. xxhash extremely fast hash algorithm. https://xxhash.com/, 2023.

[8] DRAGOJEVIĆ, A., NARAYANAN, D., CASTRO, M., AND HODSON, O. FaRM: Fast remote memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)* (Seattle, WA, Apr. 2014), USENIX Association, pp. 401–414.

[9] FAN, B., ANDERSEN, D. G., AND KAMINSKY, M. MemC3: Compact and concurrent memcache with dumber caching and smarter hashing. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)* (Lombard, IL, Apr. 2013), USENIX Association, pp. 371–384.

[10] GAO, J., LU, Y., XIE, M., WANG, Q., AND SHU, J. Citron: Distributed range lock management with one-sided RDMA. In *21st USENIX Conference on File and Storage Technologies (FAST 23)* (Santa Clara, CA, Feb. 2023), USENIX Association, pp. 297–314.

[11] GU, J., LEE, Y., ZHANG, Y., CHOWDHURY, M., AND SHIN, K. G. Efficient memory disaggregation with infiniswap. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)* (Boston, MA, Mar. 2017), USENIX Association, pp. 649–667.

[12] GUO, Z., HE, Z., AND ZHANG, Y. Mira: A program-behavior-guided far memory system. In *Proceedings of the 29th Symposium on Operating Systems Principles* (New York, NY, USA, 2023), SOSP '23, Association for Computing Machinery, p. 692–708.

[13] HERLIHY, M., SHAVIT, N., AND TZAFRIR, M. Hopscotch hashing. In *Proceedings of the 22nd International Symposium on Distributed Computing* (Berlin, Heidelberg, 2008), DISC '08, Springer-Verlag, p. 350–364.

[14] HUANG, R., HUANG, K., WANG, J., AND CHEN, Y. Reno: An rdma-enabled, non-volatile memory-optimized key-value store. In *2021 IEEE 27th International Conference on Parallel and Distributed Systems (ICPADS)* (2021), pp. 466–473.

[15] KALIA, A., KAMINSKY, M., AND ANDERSEN, D. G. Using RDMA efficiently for key-value services. *SIGCOMM Comput. Commun. Rev. 44*, 4 (Aug. 2014), 295–306.

[16] KALIA, A., KAMINSKY, M., AND ANDERSEN, D. G. Design guidelines for high performance RDMA systems. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)* (Denver, CO, June 2016), USENIX Association, pp. 437–450.

[17] KARANDIKAR, S., OU, A., AMID, A., MAO, H., KATZ, R., NIKOLIĆ, B., AND ASANOVIĆ, K. FirePerf: FPGA-accelerated full-system hardware/software performance profiling and co-design. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2020), ASPLOS '20, Association for Computing Machinery, p. 715–731.

[18] LEE, Y., MARUF, H. A., CHOWDHURY, M., CIDON, A., AND SHIN, K. G. Hydra : Resilient and highly available remote memory. In *20th USENIX Conference on File and Storage Technologies (FAST 22)* (Santa Clara, CA, Feb. 2022), USENIX Association, pp. 181–198.

[19] LI, P., HUA, Y., ZUO, P., CHEN, Z., AND SHENG, J. ROLEX: A scalable RDMA-oriented learned Key-Value store for disaggregated memory systems. In *21st USENIX Conference on File and Storage Technologies (FAST 23)* (Santa Clara, CA, Feb. 2023), USENIX Association, pp. 99–114.

[20] LI, X., ANDERSEN, D. G., KAMINSKY, M., AND FREEDMAN, M. J. Algorithmic improvements for fast concurrent cuckoo hashing. In *Proceedings of the Ninth European Conference on Computer Systems* (New York, NY, USA, 2014), EuroSys '14, Association for Computing Machinery.

[21] LIM, K., CHANG, J., MUDGE, T., RANGANATHAN, P., REINHARDT, S. K., AND WENISCH, T. F. Disaggregated memory for expansion and sharing in blade servers. In *Proceedings of the 36th Annual International Symposium on Computer Architecture* (New York, NY, USA, 2009), ISCA '09, Association for Computing Machinery, p. 267–278.

[22] LUO, X., ZUO, P., SHEN, J., GU, J., WANG, X., LYU, M. R., AND ZHOU, Y. SMART: A High-Performance adaptive radix tree for disaggregated memory. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)* (Boston, MA, July 2023), USENIX Association, pp. 553–571.

[23] MA, T., MA, T., SONG, Z., LI, J., CHANG, H., CHEN, K., JIANG, H., AND WU, Y. X-rdma: Effective rdma middleware in large-scale production environments. In *2019 IEEE International Conference on Cluster Computing (CLUSTER)* (2019), pp. 1–12.

[24] MARUF, H. A., AND CHOWDHURY, M. Effectively prefetching remote memory with Leap. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)* (July 2020), USENIX Association, pp. 843–857.

[25] MITCHELL, C., GENG, Y., AND LI, J. Using one-sided RDMA reads to build a fast, CPU-efficient key-value store. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)* (San Jose, CA, June 2013), USENIX Association, pp. 103–114.

[26] MITCHELL, C., MONTGOMERY, K., NELSON, L., SEN, S., AND LI, J. Balancing CPU and network in the cell distributed B-Tree store. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)* (Denver, CO, June 2016), USENIX Association, pp. 451–464.

[27] NISHTALA, R., FUGAL, H., GRIMM, S., KWIATKOWSKI, M., LEE, H., LI, H. C., MCELROY, R., PALECZNY, M., PEEK, D., SAAB, P., STAFFORD, D., TUNG, T., AND VENKATARAMANI, V. Scaling memcache at facebook. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)* (Lombard, IL, Apr. 2013), USENIX Association, pp. 385–398.

[28] NOVAKOVIC, S., SHAN, Y., KOLLI, A., CUI, M., ZHANG, Y., ERAN, H., PISMENNY, B., LISS, L., WEI, M., TSAFRIR, D., AND AGUILERA, M. Storm: A fast transactional dataplane for remote data structures. In *Proceedings of the 12th ACM International Conference on Systems and Storage* (New York, NY, USA, 2019), SYSTOR '19, Association for Computing Machinery, p. 97–108.

[29] NVIDIA. Advanced transport. https://docs.nvidia.com/networking/display/MLNXOFEDv494170/Advanced+Transport.

[30] NVIDIA. Device memory programming. https://docs.nvidia.com/networking/display/OFEDv502180/Programming#Programming-DeviceMemoryProgramming.

[31] NVIDIA. Roce vs. iwarp competitive analysis. https://network.nvidia.com/pdf/whitepapers/WP_RoCE_vs_iWARP.pdf.

[32] PAGH, R., AND RODLER, F. F. Cuckoo hashing. *Journal of Algorithms 51*, 2 (2004), 122–144.

[33] ROTHENBERGER, B., TARANOV, K., PERRIG, A., AND HOEFLER, T. ReDMArk: Bypassing RDMA security mechanisms. In *30th USENIX Security Symposium (USENIX Security 21)* (Aug. 2021), USENIX Association, pp. 4277–4292.

[34] RUAN, Z., SCHWARZKOPF, M., AGUILERA, M. K., AND BELAY, A. AIFM: High-performance, application-integrated far memory. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)* (Nov. 2020), USENIX Association, pp. 315–332.

[35] SHAN, Y., HUANG, Y., CHEN, Y., AND ZHANG, Y. LegoOS: A disseminated, distributed OS for hardware resource disaggregation. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)* (Carlsbad, CA, 2018), USENIX Association, pp. 69–87.

[36] SHAN, Y., LIN, W., KOSTA, R., KRISHNAMURTHY, A., AND ZHANG, Y. SuperNIC: A hardware-based, programmabe, and multi-tenant SmartNIC. In *Proceedings of the 32nd ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Monterey, CA, Mar. 2024).

[37] SHEN, J., ZUO, P., LUO, X., SU, Y., GU, J., FENG, H., ZHOU, Y., AND LYU, M. R. Ditto: An elastic and adaptive memory-disaggregated caching system. In *Proceedings of the 29th Symposium on Operating Systems Principles* (New York, NY, USA, 2023), SOSP '23, Association for Computing Machinery, p. 675–691.

[38] SHEN, J., ZUO, P., LUO, X., YANG, T., SU, Y., ZHOU, Y., AND LYU, M. R. FUSEE: A fully Memory-Disaggregated Key-Value store. In *21st USENIX Conference on File and Storage Technologies (FAST 23)* (Santa Clara, CA, Feb. 2023), USENIX Association, pp. 81–98.

[39] TAURO, B. R., SUCHY, B., CAMPANONI, S., DINDA, P., AND HALE, K. C. Trackfm: Far-out compiler support for a far memory world. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1* (New York, NY, USA, 2024), ASPLOS '24, Association for Computing Machinery, p. 401–419.

[40] TSAI, S.-Y., SHAN, Y., AND ZHANG, Y. Disaggregating persistent memory and controlling them remotely: An exploration of passive disaggregated key-value stores. In *USENIX Annual Technical Conference* (July 2020), pp. 33–48.

[41] WANG, Q., LU, Y., AND SHU, J. Sherman: A write-optimized distributed B+Tree index on disaggregated memory. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, 2022), Association for Computing Machinery, p. 1033–1048.

[42] YOON, W., OH, J., OK, J., MOON, S., AND KWON, Y. Dilos: adding performance to paging-based memory disaggregation. In *Proceedings of the 12th ACM SIGOPS Asia-Pacific Workshop on Systems* (New York, NY, USA, 2021), APSys '21, Association for Computing Machinery, p. 70–78.

[43] ZHANG, M., HUA, Y., ZUO, P., AND LIU, L. FORD: Fast one-sided RDMA-based distributed transactions for disaggregated persistent memory. In *20th USENIX Conference on File and Storage Technologies (FAST 22)* (Santa Clara, CA, Feb. 2022), USENIX Association, pp. 51–68.

[44] ZHOU, Y., WASSEL, H. M. G., LIU, S., GAO, J., MICKENS, J., YU, M., KENNELLY, C., TURNER, P., CULLER, D. E., LEVY, H. M., AND VAHDAT, A. Carbink: Fault-Tolerant far memory. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)* (Carlsbad, CA, July 2022), USENIX Association, pp. 55–71.

[45] ZUO, P., SUN, J., YANG, L., ZHANG, S., AND HUA, Y. One-sided RDMA-Conscious extendible hashing for disaggregated memory. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)* (July 2021), USENIX Association, pp. 15–29.

ARTIFACT EVALUATED
usenix ASSOCIATION
FUNCTIONAL

LITESHIELD: Secure Containers via Lightweight, Composable Userspace μKernel Services

Kaesi Manakkal, Nathan Daughety[†], Marcus Pendleton[†], Hui Lu
The University of Texas at Arlington, [†]Air Force Research Laboratory (AFRL)

Abstract

This paper introduces LITESHIELD, a new userspace isolation architecture for secure containers that reexamines the boundary between user applications and system services. LITESHIELD decouples traditional guest kernel functionality into modular userspace microkernel (μkernel) services that interact with guest applications via low-latency, shared-memory-based inter-process communication (IPC). By serving most Linux syscalls in userspace, LITESHIELD enforces a significantly reduced user-to-host interface, with just 22 syscalls, achieving strong isolation comparable to virtual machines (VMs) while avoiding the complexity of hypervisors and hardware virtualization. LITESHIELD further provides a POSIX-compatible runtime with fine-grained syscall interception to support legacy applications and enables composable μkernel services that can integrate specialized userspace components (e.g., networking and filesystems). Our implementation demonstrates that LITESHIELD delivers strong isolation with performance comparable to traditional containers.

1 Introduction

Due to high portability, high density, and low operational cost, containers have been widely used for packaging, isolating, and multiplexing cloud applications. In contrast to virtual machines (VMs) (i.e., hypervisor-based virtualization), containers execute applications directly on the native host OS [8, 53, 61] and leverage kernel-level features, such as namespaces [23], cgroups [16], and seccomp [20], to enforce *isolation* between containerized applications. While the lack of guest OSes and virtual hardware abstraction makes containers *lightweight*, they cannot be directly adopted as the isolation mechanism in multi-tenancy clouds due to *weak isolation* – sharing the same host results in a large attack surface, e.g., 300+ system calls, or syscalls, in Linux. That is precisely why, in today's production systems, containers are deployed within VMs for strong isolation [13, 37].

To address the tension between isolation and performance, recent efforts [28, 56, 62] adopt the technique of *minimization*, including tailoring a VM's kernel with minimal components [28], linking a hosted application into a tiny unikernel image under a single address space [56], and attaching a userspace guest kernel (with a substantial portion of the Linux surface) to a container for VM-like isolation [62]. While these efforts have blurred the isolation boundaries of VMs and containers, they share some common limitations. *First*, maintaining a full guest kernel, even a minimized one, for each hosted entity remains inefficient. *Second*, different applications require access to specific functionalities of the guest kernel, rendering a "one-size-fits-all" guest kernel impractical, if not impossible. *Third*, since some guest kernels have been minimized or even degraded to provide system functions at the same level as userspace applications [56], these approaches rely solely on hypervisors for isolation. However, hypervisors have their share of vulnerabilities [27].

In this paper, we reexamine the isolation boundaries between user/kernel space for applications and kernel/system services and explore a new isolation architecture for secure containers, called LITESHIELD, to achieve *lightweight yet strong isolation*. Inspired by the microservices architecture, LITESHIELD decouples the closely-coupled guest kernel and its hosted applications, or *guest applications*, as loosely-coupled entities. Such decoupling allows the guest kernel to operate as a collection of userspace microkernel (μkernel) services, each running as regular userspace processes. Communication between μkernel services and guest applications is facilitated through efficient userspace inter-process communication (IPC), eliminating costly syscalls.

The isolation architecture in LITESHIELD achieves *strong isolation with minimal overhead*. First, LITESHIELD achieves strong isolation by blocking direct host kernel access for guest applications. Instead, they are served by userspace μkernel services: By serving system services (e.g., networking and filesystems) in the userspace, the *user-to-kernel* interface – i.e., between μkernel services and the host – is significantly

DISTRIBUTION STATEMENT A. Approved for public release: Distribution unlimited. Case Number AFRL-2025-0650. Dated 06 Feb 2025.

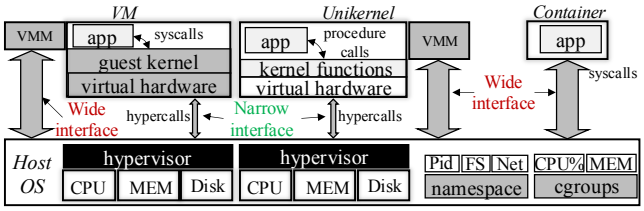


Figure 1: Comparisons of three representative isolation architectures: VMs, unikernels, and containers.

reduced (e.g., from 300+ syscalls to 20+) and comparable to VMs (e.g., 20+ hypercalls and 60+ VMExits). Even if a malicious guest application exploits a μkernel service (via userspace IPC), its access is limited to a restricted userspace process (i.e., defense in depth like VMs). As communication between guest applications and μkernel services is facilitated through userspace IPC, LITESHIELD removes the hypervisor and further reduces the attack surface. Further, by hosting system services in userspace, LITESHIELD replaces expensive cross-boundary invocation overheads (e.g., syscalls or VMExits/VMEntries) with fast (cache-to-cache) IPC. Last, LITESHIELD supports composable μkernel services, enabling seamless integration of specialized userspace approaches [41,44,47,48] to deliver highly efficient userspace system services, instead of relying solely on general-purpose, monolithic, and hard-to-optimize in-kernel services.

To ensure compatibility with existing commodity kernels (e.g., Linux) and legacy applications, LITESHIELD addresses several challenges: First, the Linux kernel's implementation restricts certain syscalls (e.g., process and memory management) from being executed in a separate process, complicating the operation of μkernel services as independent userspace processes. To address this, LITESHIELD categorizes syscalls into *delegable syscalls*, which are redirected to LITESHIELD's μkernel services for processing, and *non-delegable syscalls*, which are trapped, monitored, and validated before being executed within the same process via an arbitration mechanism. To further support legacy applications, LITESHIELD provides a POSIX-compatible library that supports runtime injection and fine-grained syscall interception, enabling seamless redirection of *delegable syscalls* from legacy applications to userspace μkernel services, without any binary modifications. Last, to achieve high performance, LITESHIELD employs a userspace IPC mechanism with a shared memory region and polling-based threads to facilitate low-latency communication between guest applications and μkernel services.

Our current implementation of LITESHIELD supports most of the Linux kernel syscalls in userspace required by regular guest applications (i.e., those not running with root privileges) and achieves a thin *user-to-host* interface with only 22 syscalls that need the support of the host kernel (compared to 20+ hypercalls and 64 VMExits for KVM-based VMs), while significantly reducing both the software codebase (eliminating the need for a hypervisor and QEMU-based emulator) and hardware complexity (requiring no hardware virtualization). We have ported an existing userspace network stack, `f-stack` [9], and implemented an `ext2`-like userspace filesystem as userspace networking and filesystem μkernel services. Porting `f-stack` to LITESHIELD only required 400+ lines of code. By leveraging lightweight userspace μkernel services and fast shared memory-based IPC between guest applications and μkernel services, LITESHIELD delivers performance comparable to traditional containers.

2 Motivation

2.1 Cloud Native and Isolation

IT companies are under constant pressure to simplify their product development with shortened production cycles to adapt to changing markets and diverse demands. Cloud-native technologies are poised to tackle this pressing challenge. First, developers decompose a traditional monolithic application into graphs of *single-purpose, loosely-coupled* microservices. As each microservice focuses on a small subset of the monolithic application's functionality, this microservices-based architecture reduces development complexity and increases code velocity [1, 3, 4, 6, 12]. Further, cloud-native platforms deploy, manage, and scale microservices completely for cloud tenants, further liberating them from the management of virtual servers (i.e., server*less* for tenants). The use of cloud-native technologies is becoming pervasive: Companies like Amazon, Netflix, Twitter, Uber, and eBay have adopted the microservices architecture [5, 7, 10, 17, 26]. In addition, a proliferation of serverless platforms enables a simple way, i.e., via Function-as-a-Service (FaaS), to build and execute cloud-native applications [2, 24, 25, 35, 36].

Isolation ensures safe resource sharing by preventing cloud tenants from accessing each other's shared resources. Without enforcing isolation, a malicious user could steal sensitive information from victims [54, 60, 63, 64], or an aggressive user might degrade the performance of others [49].

State-of-the-art virtualization techniques provide isolation via full-blown VMs [14, 58], micro-VMs [28, 62], containers [8, 53, 61], microkernels [30, 43, 52], and unikernels [31,40,42,57]. As illustrated in Figure 1, VM-based virtualization achieves isolation through a *virtual hardware interface*, allowing each VM to operate with a fully functional guest operating system (OS). Since the virtual hardware interface differs from real hardware, it is further supervised by another software layer, the *hypervisor*. The virtualization architecture provides a *strong* security boundary between sandboxed applications in VMs due to: 1) a minimal attack surface between VMs and the native host (e.g., tens of hypercalls or VMExits); and 2) defense-in-depth, where both the guest OS and hypervisor contribute to security. However, this defense-in-depth approach introduces non-trivial performance overhead by layering guest kernels and hypervisors, resulting in costly inter-

DISTRIBUTION STATEMENT A. Approved for public release:
Distribution unlimited. Case Number AFRL-2025-0650. Dated 06 Feb 2025.

actions across multiple layers of the virtualization stack for CPU, memory, and I/O virtualization. Microkernels improve isolation by minimizing kernel functionality and shifting OS services, such as file systems and drivers, to userspace, reducing the attack surface and improving fault isolation. Notably, seL4 [43] offers formal verification guarantees, while systems like Barrelfish [30] and Arrakis [52] extend this model to multicore and I/O-optimized architectures. Though well-suited for specialized, high-assurance systems, microkernels often face challenges with performance due to IPC overhead and supporting legacy applications. These factors limit usability in general-purpose cloud-native environments.

In contrast, containers execute applications directly on the native host OS [8, 53, 61]. Kernel-level features, e.g., namespaces [23], cgroups [16], and seccomp [20], enforce isolation between containerized applications. While the lack of virtual hardware abstraction makes containers lightweight, they cannot be directly adopted as the isolation mechanism in multi-tenant clouds due to *weak isolation* – i.e., sharing the same host results in a large attack surface (e.g., 400+ system calls, or syscalls, in Linux). While modern OSes have a "whitelisting" mechanism (e.g., `seccomp`) allowing containers to transition into a restricted mode with a narrowed syscall interface, the common problem is that it is difficult to determine what syscalls an application may use. Default whitelisting policies still tend to be large, e.g., 250+ syscalls [21].

Recent efforts [28, 50, 62] aim to balance isolation and performance through *minimization*. For example, AWS's Firecracker [28] optimizes a VM's kernel by including only essential components and adopting a simplified I/O model. NEC's LightVM [50] integrates a hosted application into a minimal unikernel image within a single address space for greater efficiency. Google's gVisor [62] employs a userspace guest kernel, incorporating a significant portion of the Linux interface, to provide VM-like isolation for containers.

2.2 Limitations

Unfortunately, cloud native presents new and pressing challenges to existing virtualization techniques. *1) High performance overhead:* Unlike monolithic applications, a cloud-native application consists of numerous distributed microservices, each requiring sandboxing, which significantly amplifies the performance overhead and memory footprint. Moreover, virtualization overhead becomes more pronounced compared to monolithic applications, as each hosted microservice is typically smaller and more ephemeral (e.g., high startup overhead). *2) Generality vs. specialization:* The increasing diversity of microservices necessitates specialized system services, such as customized network stacks [51] or specialized file systems [47]. However, existing virtualization techniques for isolation continue to rely on general-purpose, one-size-fits-all guest kernels. While these kernels are often highly optimized [28], they struggle to meet the varied and specific requirements of diverse microservices. Furthermore, despite recent advancements in high-performance networking and file systems [41, 47, 51], there is no practical way to seamlessly integrate these technologies with general applications or microservices. *3) Large attack surface.* Although the user-to-host interface for VM-based virtualization is thin, the code responsible for virtualization and isolation remains large. This includes both *the hypervisor*, which resides in the host kernel (i.e., type-2 hypervisor) and interacts with hardware mainly for CPU and memory management, and *the QEMU-based Virtual Machine Monitor* (VMM), which operates in userspace primarily for device emulation. According to the Common Vulnerabilities and Exposures (CVE) database, 184 vulnerabilities have been reported in major hypervisors (e.g., Xen, KVM, Hyper-V) since 2007, with 33% of these occurring in the past 1.5 years. Additionally, the VMM is huge (e.g., QEMU-based VMM has more than 1.4 million lines of code) and can access the host via the whole syscall interface.

2.3 Threat Model and Assumptions

In this paper, we focus on security vulnerabilities and isolation mechanisms in general-purpose monolithic kernels (e.g., Linux). We share the common isolation assumptions as VMs/unikernels [56, 62]: We trust fundamental hardware-based protections – such as page tables and CPU execution modes – that ensure strong isolation between different processes and between user and kernel execution within the same process. We focus on software deficiencies in host kernels, guest kernels, and hypervisors that can be exploited through their exposed *user-to-host* interfaces, namely, system calls and hypercalls, which constitute the main attack surface. Therefore, our threat model is that one malicious user could break out of the isolation by compromising the *user-to-host* interface. Two metrics are used to evaluate the attack surface: 1) the size of the user-to-host interface and 2) the amount of code accessible through the interface. Other attacks, such as covert channels and side channels, pose security risks by enabling unauthorized communication and information leakage through unconventional pathways. Existing software-based isolation approaches, including VMs, microVMs, unikernels, and others, are susceptible to these attacks. Although eliminating all side and covert channels is challenging due to the complexity of hardware and software interactions, existing measures, such as secure hardware designs, strict resource isolation, and the introduction of system randomness, can be orthogonally applied to software-based isolation approaches.

3 Design of LITESHIELD

We present LITESHIELD, a novel sandboxing architecture providing *strong yet lightweight* isolation for secure containers. Drawing inspiration from the microservice architecture, LITESHIELD offers *on-demand, composable* guest kernel/system services to cloud-native applications, functioning

DISTRIBUTION STATEMENT A. Approved for public release:
Distribution unlimited. Case Number AFRL-2025-0650. Dated 06 Feb 2025.

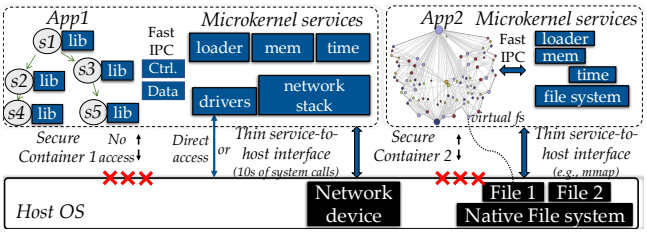


Figure 2: Architecture of LITESHIELD.

as userspace microkernel (μkernel) services. (Figure 2).

LITESHIELD's core innovation lies in decoupling guest applications from their hosting guest kernels, running them as separate userspace entities. Each application process acts as a client, while all required μkernel services are combined to provide system services. Clients interact with userspace μkernel services through fast inter-process communication (IPC) channels (§3.2). System calls made by a guest application are intercepted and redirected to the appropriate userspace μkernel service, which handles the request mostly within the userspace while only contacting the host kernel for (a limited number of) privileged operations, resulting in a thin *user-to-host* interface (comparable to VMs). To prevent malicious applications from bypassing the interception library and making direct system calls to the host kernel, LITESHIELD employs `seccomp` [20], effectively blocking unauthorized "direct" syscalls.

LITESHIELD inherits advantages from the microservices architecture, such as modular development, flexible deployment, and rapid iteration. Userspace μkernel services can be gradually and individually developed, extended, replaced, customized, and integrated with other existing/ongoing userspace systems [41, 47, 51]. In addition, while hardware specialization is accelerating in cloud [18], LITESHIELD can quickly support specialized hardware (e.g., GPU, smart NICs, and persistent memory) with userspace support, making new, advanced hardware accessible by specific cloud-native applications (e.g., machine learning tasks).

3.1 Strong Isolation via Thin Interface

As shown in Figure 2, LITESHIELD achieves a level of isolation comparable to VMs and unikernels, maintaining a similarly *thin user-to-host interface* (i.e., requiring tens of syscalls to the host, compared to 300+ for containers).

First, guest applications have no direct access to the host kernel. LITESHIELD enforces this restriction by using `seccomp` to block all direct syscalls from guest applications, i.e., by applying a `seccomp` profile that denies all syscalls by default. Second, userspace μkernel services are permitted to access the host kernel when necessary. To ensure that the *user-to-host* interface remains *thin*, `seccomp` is also applied to these services, allowing only a minimal set of explicitly defined syscalls through a restrictive profile. The rationale is simple: "the more done in userspace, the less needed in the kernel". For example, the unikernel approach [56] reduces

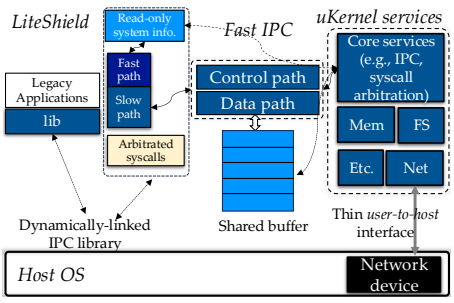


Figure 3: Fast userspace inter-process communication (IPC).

syscall or hypercall usage to fewer than ten by shifting most system functions into userspace. Similarly, LITESHIELD performs most guest kernel functions, such as file and filesystem management, networking, and IPC, entirely within userspace, thereby minimizing interactions with the host kernel and achieving a thin *user-to-host* interface.

Unlike the unikernel approach, which embeds guest kernel functions in the same address space as guest applications, LITESHIELD executes these functions within more flexible and independent μkernel services. However, certain syscalls, namely *non-delegable* ones, must be executed within the context of the same process, such as process management (e.g., `fork`, `clone`, `wait`, and `exit`) and memory management (e.g., `mmap`, `munmap`, `mprotect`, `msync`, and `madvise`). A straightforward solution would make the kernel LITESHIELD-aware by introducing kernel support to convert *non-delegable* syscalls to *delegable* ones, allowing them to execute in the context of another process. LITESHIELD, instead, focuses on compatibility with the *unmodified* Linux kernel and realizes an *arbitration* mechanism to handle the execution of the non-delegable syscalls. Specifically, LITESHIELD permits guest applications to execute non-delegable syscalls by explicitly allowing them in the `seccomp` profile. However, it leverages Linux's `ptrace` mechanism to trap and monitor these syscalls through LITESHIELD's core μkernel service (§3.3). When a guest application is launched, it is registered as a tracee of the core μkernel service before execution begins. Thus, any subsequent invocation of a non-delegable syscall is intercepted by the core service, allowing LITESHIELD to perform sanity checks or other forms of inspection [34, 38] before permitting the syscall to proceed. This mechanism enables fine-grained monitoring of non-delegable syscalls while maintaining compatibility with the existing Linux kernel.

Since guest applications can only access the host kernel via LITESHIELD, even if a malicious guest application exploits a bug in a μkernel service of LITESHIELD (e.g., through IPC channels), it can only gain access to a restricted userspace process (i.e., defense in depth). As the communication between guest applications and μkernel services is through userspace IPCs or arbitrated syscalls, LITESHIELD eliminates the hypervisor, further minimizing the attack surface for isolation.

DISTRIBUTION STATEMENT A. Approved for public release:
Distribution unlimited. Case Number AFRL-2025-0650. Dated 06 Feb 2025.

3.2 Lightweight Isolation via Fast IPCs

One key goal of LITESHIELD is to provide strong isolation with much *less* overhead than VM/unikernel-based isolation to *legacy* applications, better, with performance comparable to that of "lightweight" containers.

Containerized applications request system services via *syscalls* from the kernel space. This mechanism is actually costly, as it involves same-core user/kernel mode transitions, cache pollution, and data movement across the kernel/user space, consuming *microseconds* [55]. In addition, the "one-size-fits-all" monolithic kernel services could be suboptimal for many new application scenarios [45, 46]. The generality of monolithic kernels makes optimization difficult, and vertical integration efforts are often broken by upstream changes [51]. On the other hand, recent userspace and hybrid approaches [32, 39, 41, 44, 47, 48, 59] have been proposed to mitigate such overhead by redistributing functions between user and kernel space, demonstrating superior performance compared to in-kernel services. Following this direction, LITESHIELD runs most kernel services in userspace, achieving a thin user-to-host interface (and strong isolation) while also enabling the potential for high performance through various specialized userspace approaches.

Guest applications request system services from *userspace* μkernel services through IPC channels, instead of syscalls. As shown in Figure 3, LITESHIELD develops a *high-performance shared-memory-based IPC mechanism* for fast system service delivery. *First*, each application is assigned a shared memory buffer. When an application makes a request, it places the syscall number and arguments in the shared buffer and toggles a flag to let the μkernel services know that a request has been made. Once the request has been completed, the μkernel service puts the response in this buffer and toggles the flag to let the guest application know it is done. This approach draws inspiration from existing userspace communication techniques, e.g., RDMA, and incorporates a *polling-based* mechanism to reduce communication latency: A core *μkernel service for IPCs* employs a dedicated polling thread to continuously monitor for incoming IPC requests from guest application processes. Upon detection of a request, this thread promptly handles it – by forwarding it to one of the composable userspace μkernel services (e.g., networking or file systems). Similarly, application processes use another polling thread to actively wait for and immediately process responses from the IPC μkernel service. *Second*, LITESHIELD leverages a multi-core system (common today) to place application processes and userspace μkernel services on separate cores – i.e., avoiding same-core context switches. This separation precludes context switching on the same core and further minimizes communication latency between guest applications and μkernel services. Since the context switch overhead for userspace processes is typically lower than that of virtual CPUs, LITESHIELD's userspace solution is expected to outperform VMs even with core multiplexing. *Finally*, the communication latency under the proposed shared-memory-based IPC mechanism mainly hinges on the memory access latency. If the cores for the application and the userspace μkernel services are situated on the same CPU, this configuration can capitalize on the last-level cache (LLC) to expedite IPC (i.e., cache-to-cache transfers that typically require only tens of CPU cycles [55]).

Moreover, LITESHIELD provides a POSIX-compatible library that combines `LD_PRELOAD` and the binary translation capabilities of the `libsyscall_intercept` library [15]. The `LD_PRELOAD` mechanism allows the library to be injected into the address space of legacy applications at runtime, enabling it to override standard library functions (i.e., `glibc`) and intercept system calls. Meanwhile, `libsyscall_intercept` provides fine-grained control over syscall interception by hooking directly into the syscall execution path using inline hooking and binary rewriting techniques. This combined approach enables LITESHIELD to dynamically link the library to legacy applications – without any modifications to the binaries – and intercept delegable syscalls. The intercepted syscalls are redirected to the IPC shared buffer, where LITESHIELD facilitates communication with its μkernel services.

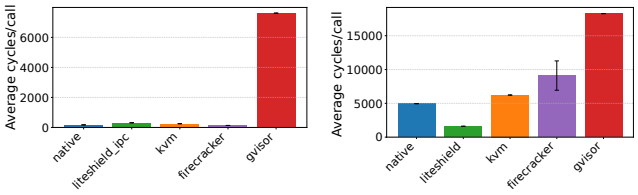
Like FlexSC [55], LITESHIELD aims to improve syscall performance by rethinking the traditional syscall interface. However, they take different architectural approaches. FlexSC introduces exception-less system calls that batch syscall execution and decouple it from the application thread, reducing kernel traps and improving throughput on multicore systems. In contrast, LITESHIELD decomposes traditional kernel functionality into userspace μkernel services and uses fast userspace IPC and selective syscall trapping (via ptrace) to intercept and redirect syscalls, maintaining compatibility with unmodified Linux and legacy applications.

3.3 Userspace μKernel Services

As illustrated in Figure 3, LITESHIELD's μkernel services are divided into two categories: core services and composable services. Core services, such as IPC, syscall arbitration, time management, and memory management, provide essential μkernel functionality required by every guest application. In contrast, composable services, including file, device, and networking management, are provided on demand. Integrating existing and new userspace approaches into LITESHIELD as composable μkernel services is straightforward.

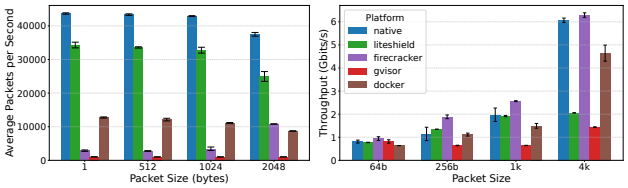
We have integrated the `DPDK`-based userspace network stack, `f-stack` [9], into LITESHIELD. First, we extended LITESHIELD to fetch syscall requests from the IPC shared buffer and enqueue them in a separate operation queue for `f-stack` to process. Further modifications were made to `f-stack`'s main network processing loop to monitor the operation queue and handle any received network requests. Once processed, the results are placed back into the IPC shared buffer, where the guest application can retrieve them.

DISTRIBUTION STATEMENT A. Approved for public release:
Distribution unlimited. Case Number AFRL-2025-0650. Dated 06 Feb 2025.



(a) `getpid`. (b) `read`.

Figure 4: Syscall latency comparisons.



(a) UDP. (b) TCP.

Figure 5: Network performance comparisons, on UDP and TCP protocols, with different packet sizes.

`f-stack` primarily processes packets in userspace, interacting with the underlying network device only when necessary (e.g., for sending packets) through a limited syscall interface (e.g., `read` and `write`). Each instance of `f-stack` is bound to a software `tap` device, enabling the sharing of a physical network card across multiple containers. Last, `f-stack` supports direct use of a physical NIC for scenarios requiring dedicated access. We have also implemented and integrated a userspace filesystem, compatible with `ext2` [22], into LITESHIELD.

Table 1: Ptrace overhead for non-delegable syscalls.

Syscall	Syscall lat (μs)	Ptrace lat (μs)	Ptrace overhead
mmap	0.220	25.485	99.1%
fork	40.553	35.656	46.8%
clock_nanosleep	56.896	23.256	29.0%
futex	0.254	15.476	98.4%

4 Evaluation

We have implemented LITESHIELD in approximately 7,000 lines of C/C++ code [1]. Currently, LITESHIELD supports the redirection or arbitration of around 170 Linux kernel syscalls, including 142 delegable and 28 non-delegable ones. Most of the remaining unsupported syscalls (approximately 132) require root privileges from guest applications. LITESHIELD's modular design allows for the incremental addition of support for these syscalls. The *user-to-host* interface in LITESHIELD is thin, requiring only 22 syscalls – compared to 60+ VMExits for KVM-based VMs and 250+ syscalls in the default `seccomp` whitelist for containers. Please refer to Appendix A for a more detailed breakdown. We have evaluated the effectiveness of LITESHIELD by comparing it with state-of-the-art isolation mechanisms, including Docker containers [8], KVM-based VMs [14], Firecracker [28], and gVisor [62].

Testbed. We conducted our experiments on a platform with an Intel Xeon Gold 6430 CPU, 96GB DDR5 RAM, and a Micron 7450 NVMe SSD (ext4), running Ubuntu 22.04 with Linux kernel 5.15. Hyperthreading was disabled, and resources were configured to avoid bottlenecks. KVM tests used 16 vCPUs, 32GB RAM, and a virtio disk in direct sync cache mode with the same OS as the host. gVisor [62] (v1.10.1) was tested in systrap mode with Docker support, while Firecracker [28] (v1.10.1) was tested with an Ubuntu 24.04 root filesystem on Linux kernel 5.10, configured with 16 vCPUs and 32GB RAM. Docker containers and LITESHIELD were constrained to 16 cores and 32GB RAM using `cgroup` [16].

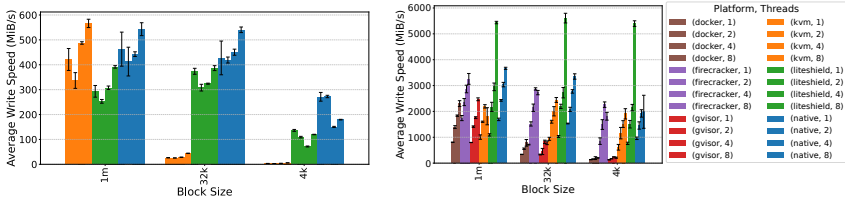
Syscall latency. We evaluated syscall performance using two representative syscalls: a simple syscall, `getpid`, and a complex syscall, `read`. Figure 4a shows the average latency of invoking `getpid` one million times – LITESHIELD achieves significantly lower latency than the user-level isolation mechanism gVisor due to its fast IPC mechanism, while maintaining comparable latency to other VM-based approaches for this simple syscall. Figure 4b illustrates the average time to read 1 byte from each block of a 4GB file. For this complex syscall, LITESHIELD outperforms VM-based approaches, as `read` triggers VMExits in VMs, incurring high context-switch overhead. Further, LITESHIELD surpasses native performance due to its *specialized, lightweight* userspace filesystem.

Ptrace overhead. We evaluated the performance impact of LITESHIELD's `ptrace`-based arbitration mechanism for non-delegable syscalls. As shown in Table ??, we selected one representative syscall from each of the four main classes of non-delegable syscalls: process management, memory management, timing, and locking. On average, `ptrace` introduces 15–35μs of overhead, with the effect being more pronounced for lightweight syscalls (e.g., `mmap` and `futex`) and less significant for heavier ones (e.g., `fork` and `clock`). We note that these syscalls are generally invoked infrequently. In future work (§5), we plan to convert these non-delegable syscalls into delegable ones to eliminate this overhead.

Userspace networking. We evaluated the performance of LITESHIELD's userspace network stack, ported from `f-stack`. Figure 5a illustrates UDP network performance (packets/second) with a client sending UDP packets of various sizes to a server hosted under different isolation approaches. We used a pair of connected virtual interfaces (i.e., `veth`) to connect the virtual NICs of the client and server. LITESHIELD outperforms all other isolation mechanisms, with performance slightly below native, due to the highly optimized userspace network stack provided by `f-stack`. Figure 5b shows TCP performance using `iperf` [11], with one instance as the client and another as the server hosted under those same isolation

[1] Project website: https://github.com/kmanakk1/liteshield

DISTRIBUTION STATEMENT A. Approved for public release:
Distribution unlimited. Case Number AFRL-2025-0650. Dated 06 Feb 2025.



(a) Direct IO. (b) Cached IO.

Figure 6: Performance for writing a 4GB file with various threads and block sizes.

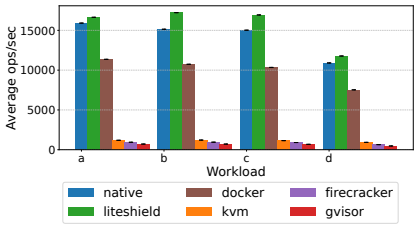


Figure 7: `YCSB` [33] performance on Redis [19] with different workloads.

approaches. While LITESHIELD delivers comparable performance for small packets, it falls behind the microVM approach (e.g., Firecracker) for larger packets because `f-stack` lacks the `GRO` feature. `GRO` reduces packet processing overhead for large packets by merging multiple small packets before processing, but it may introduce latency for small packets, highlighting the lack of a "one-size-fits-all" solution. With LITESHIELD, however, guest applications can select μkernel services tailored to their specific requirements.

Userspace filesystem. We evaluated the performance of LITESHIELD 's userspace filesystem, developed from scratch to emulate the functionality of an in-kernel `ext2` filesystem. Using `fio` [29] (version 3.28), we conducted two types of write I/O tests: 1) cached I/O, where data is written to the page cache and asynchronously flushed to disk, and 2) direct I/O (with `O_DIRECT`), which bypasses the page cache and reaches the disk directly. The `fio` benchmarks were run with multiple threads accessing a single 4GB file using various block sizes.

For VM-based approaches (e.g., gVisor and Firecracker), direct I/O (`O_DIRECT`) bypasses the VM's page cache but can still be buffered by the host's page cache, causing double caching and hindering direct persistence to disk. In contrast, with userspace isolation, LITESHIELD completely eliminates double caching. As shown in Figure 6a, LITESHIELD achieves higher write performance than KVM (we explicitly configured QEMU to enforce direct writes for its disk) and native setups for smaller block sizes, with slightly lower performance for very large block sizes (e.g., 1MB). For cached I/O, Figure 6b shows that LITESHIELD demonstrates better scalability than other approaches as the thread number increases. This is because LITESHIELD's userspace filesystem has a greatly simplified and efficient page cache mechanism.

Real-world applications. We evaluated the performance of a real-world application, Redis [19] v6.0.16, on LITESHIELD (with both networking and filesystem μkernel services) using YCSB v0.18.0 [33] by executing four distinct workloads: Workloads A (50% reads, 50% updates), B (95% reads, 5% updates), C (95% reads, 5% inserts), and D (50% reads, 50% read-modify-write). Figure 7 shows that LITESHIELD achieves higher performance compared to native execution. This improvement is primarily due to the reduced overhead of IPC versus traditional syscalls, particularly for complex operations. In contrast, this overhead in alternative isolation solutions, such as Firecracker and gVisor, is more pronounced, resulting in inferior performance.

5 Conclusions and Future Work

We present LITESHIELD, an effort to explore to what extent guest applications can be isolated in userspace without requiring kernel or application modifications. LITESHIELD achieves this by decoupling guest applications from their guest kernels and offering guest kernel services as a collection of userspace μkernel services. It ensures strong isolation through a thin user-to-host interface and delivers high performance with specialized userspace μkernel services.

Despite its effectiveness, LITESHIELD has several limitations that suggest directions for future work. First, the `ptrace`-based arbitration mechanism introduces overhead for non-delegable system calls. While we consider this a reasonable trade-off to achieve performance, compatibility, and isolation, future kernel support could eliminate this overhead by converting non-delegable syscalls into delegable ones. Rather than modifying the host kernel to introduce new context-aware variants of these syscalls, we are exploring a more transparent solution: leveraging a kernel module to detect whether non-delegable syscalls originate from LITESHIELD's μkernel services. When such calls are identified, they could be dynamically converted into context-aware syscalls. Furthermore, statically linked applications, including those with custom `libc` implementations or inline assembly system call instructions, may bypass LITESHIELD's interception library, resulting in failures due to `seccomp` blocking. To address this issue, we are exploring a "hotpatching" technique that disassembles system call instructions in statically linked processes and replaces them with hooks that redirect the calls to LITESHIELD's IPC mechanisms.

6 Acknowledgments

We thank our shepherd, Kevin Pedretti, and the anonymous reviewers for their valuable feedback. We also thank Fotis Antonatos for his early contributions to this project. This work was supported by the Air Force Research Laboratory (AFRL) under Awards FA8750-24-2-0001 and FA8750-25-C-B038, and by the National Science Foundation (NSF) under Awards CCF-2415473 and CNS-2415774.

DISTRIBUTION STATEMENT A. Approved for public release:
Distribution unlimited. Case Number AFRL-2025-0650. Dated 06 Feb 2025.

References

[1] 5 principles for cloud-native architecture—what it is and how to master it. https://cloud.google.com/blog/products/application-development/5-principles-for-cloud-native-architecture-what-it-is-and-how-to-master-it.

[2] AWS Lambda:Run code without thinking about servers. Pay only for the compute time you consume. https://aws.amazon.com/lambda/.

[3] Cloud native applications: Ship faster, reduce risk, and grow your business. https://tanzu.vmware.com/cloud-native.

[4] Cncf cloud native definition v1.0. https://github.com/cncf/toc/blob/main/DEFINITION.md.

[5] Decomposing twitter: Adventures in service-oriented architecture. https://www.infoq.com/presentations/twitter-soa/.

[6] Defining cloud native. https://docs.microsoft.com/en-us/dotnet/architecture/cloud-native/definition.

[7] A design analysis of cloud-based microservices architecture at netflix. https://medium.com/swlh/a-design-analysis-of-cloud-based-microservices-architecture-at-netflix-98836b2da45f.

[8] The docker container. https://www.docker.com/.

[9] f-stack. https://www.f-stack.org/.

[10] Introducing domain-oriented microservice architecture. https://eng.uber.com/microservice-architecture/.

[11] iperf - the ultimate speed test tool for tcp, udp and sctp. https://iperf.fr/.

[12] Journey to being cloud-native – how and where should you start? https://aws.amazon.com/blogs/apn/journey-to-being-cloud-native-how-and-where-should-you-start/.

[13] Kata containers. https://github.com/kata-containers.

[14] Kernel based virtual machine. http://www.linux-kvm.org/.

[15] libsyscall_intercept. https://github.com/pmem/syscall_intercept.

[16] Linux control groups. https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt.

[17] Microservices at ebay-what it looks like today. https://www.sayonetech.com/blog/microservices-ebay/.

[18] A new golden age for computer architecture. https://cacm.acm.org/magazines/2019/2/234352-a-new-golden-age-for-computer-architecture/fulltext.

[19] Redis. https://redis.io/.

[20] Seccomp bpf (secure computing with filters). https://www.kernel.org/doc/html/v5.1/userspace-api/seccomp_filter.html.

[21] Seccomp security profiles for docker. https://docs.docker.com/engine/security/seccomp/.

[22] The second extended filesystem. https://docs.kernel.org/filesystems/ext2.html.

[23] Separation Anxiety: A Tutorial for Isolating Your System with Linux Namespaces. https://www.toptal.com/linux/separation-anxiety-isolating-your-system-with-linux-namespaces.

[24] Serverless. https://cloud.google.com/serverless/.

[25] Serverless computing. https://azure.microsoft.com/en-us/overview/serverless-computing/.

[26] What led amazon to its own microservices architecture. https://thenewstack.io/led-amazon-microservices-architecture/.

[27] Common Vulnerabilities and Exposures: Hypervisors. http://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=hypervisor.

[28] AGACHE, A., BROOKER, M., IORDACHE, A., LIGUORI, A., NEUGEBAUER, R., PIWONKA, P., AND POPA, D.-M. Firecracker: Lightweight virtualization for serverless applications. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)* (Santa Clara, CA, Feb. 2020), USENIX Association, pp. 419–434.

[29] AXBOE, J. Flexible i/o tester. https://github.com/axboe/fio.

[30] BAUMANN, A., BARHAM, P., DAGAND, P.-E., HARRIS, T., ISAACS, R., PETER, S., ROSCOE, T., SCHÜPBACH, A., AND SINGHANIA, A. The multikernel: a new os architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles* (New York, NY, USA, 2009), SOSP '09, Association for Computing Machinery, p. 29–44.

DISTRIBUTION STATEMENT A. Approved for public release: Distribution unlimited. Case Number AFRL-2025-0650. Dated 06 Feb 2025.

[31] BRATTERUD, A., WALLA, A.-A., HAUGERUD, H., ENGELSTAD, P. E., AND BEGNUM, K. Includeos: A minimal, resource efficient unikernel for cloud services. CLOUDCOM '15, IEEE Computer Society, p. 250–257.

[32] CHEN, Y., SHU, J., OU, J., AND LU, Y. Hinfs: A persistent memory file system with both buffering and direct-access. *ACM Trans. Storage 14*, 1 (apr 2018).

[33] COOPER, B. Yahoo! cloud serving benchmark. https://github.com/brianfrankcooper/YCSB.

[34] DEMARINIS, N., WILLIAMS-KING, K., JIN, D., FONSECA, R., AND KEMERLIS, V. P. sysfilter: Automated system call filtering for commodity software. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)* (San Sebastian, Oct. 2020), USENIX Association, pp. 459–474.

[35] ELLIS, A. Introducing Functions as a Service (OpenFaaS). https://blog.alexellis.io/introducing-functions-as-a-service/.

[36] HENDRICKSON, S., STURDEVANT, S., HARTER, T., VENKATARAMANI, V., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Serverless computation with openlambda. *Elastic 60*, 80.

[37] HUANG, H., LAI, J., RAO, J., LU, H., HOU, W., SU, H., XU, Q., ZHONG, J., ZENG, J., WANG, X., HE, Z., HAN, W., LIU, J., MA, T., AND WU, S. Pvm: Efficient shadow paging for deploying secure containers in cloud-native environment. In *Proceedings of the 29th Symposium on Operating Systems Principles* (New York, NY, USA, 2023), SOSP '23, Association for Computing Machinery, p. 515–530.

[38] JACOBS, A., GÜLMEZ, M., ANDRIES, A., VOLCKAERT, S., AND VOULIMENEAS, A. System call interposition without compromise. In *2024 54th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)* (2024), pp. 183–194.

[39] KADEKODI, R., LEE, S. K., KASHYAP, S., KIM, T., KOLLI, A., AND CHIDAMBARAM, V. Splitfs: Reducing software overhead in file systems for persistent memory. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2019), SOSP '19, Association for Computing Machinery, p. 494–508.

[40] KANTEE, A. The Design and Implementation of the Anykernel and Rump Kernels, 2nd Edition. http://book.rumpkernel.org.

[41] KAUFMANN, A., STAMLER, T., PETER, S., SHARMA, N. K., KRISHNAMURTHY, A., AND ANDERSON, T. Tas: Tcp acceleration as an os service. In *Proceedings of the Fourteenth EuroSys Conference 2019* (New York, NY, USA, 2019), EuroSys '19, Association for Computing Machinery.

[42] KIVITY, A., LAOR, D., COSTA, G., ENBERG, P., HAR'EL, N., MARTI, D., AND ZOLOTAROV, V. Osv—optimizing the operating system for virtual machines. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)* (Philadelphia, PA, June 2014), USENIX Association, pp. 61–72.

[43] KLEIN, G., ELPHINSTONE, K., HEISER, G., ANDRONICK, J., COCK, D., DERRIN, P., ELKADUWE, D., ENGELHARDT, K., HUUCK, R., MURRAY, T. C., SEWELL, T., TUCH, H., AND WINWOOD, S. seL4: Formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP)* (Big Sky, Montana, USA, 2009), ACM, pp. 207–220.

[44] KWON, Y., FINGLER, H., HUNT, T., PETER, S., WITCHEL, E., AND ANDERSON, T. Strata: A cross media file system. In *Proceedings of the 26th Symposium on Operating Systems Principles* (New York, NY, USA, 2017), SOSP '17, Association for Computing Machinery, p. 460–477.

[45] LEI, J., MUNIKAR, M., SUO, K., LU, H., AND RAO, J. Parallelizing packet processing in container overlay networks. In *Proceedings of the Sixteenth European Conference on Computer Systems* (New York, NY, USA, 2021), EuroSys '21, Association for Computing Machinery, p. 261–276.

[46] LEI, J., SUO, K., LU, H., AND RAO, J. Tackling parallelization challenges of kernel network stack for container overlay networks. In *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)* (Renton, WA, July 2019), USENIX Association.

[47] LIU, J., REBELLO, A., DAI, Y., YE, C., KANNAN, S., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Scale and performance in a filesystem semi-microkernel. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (New York, NY, USA, 2021), SOSP '21, Association for Computing Machinery, p. 819–835.

[48] LIU, J., REBELLO, A., DAI, Y., YE, C., KANNAN, S., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Scale and performance in a filesystem semi-microkernel. SOSP '21, Association for Computing Machinery, p. 819–835.

DISTRIBUTION STATEMENT A. Approved for public release: Distribution unlimited. Case Number AFRL-2025-0650. Dated 06 Feb 2025.

[49] LU, H., SALTAFORMAGGIO, B., KOMPELLA, R., AND XU, D. vFair: Latency-aware fair storage scheduling via per-io cost-based differentiation. In *Proceedings of the 6th ACM Symposium on Cloud Computing* (2015).

[50] MANCO, F., LUPU, C., SCHMIDT, F., MENDES, J., KUENZER, S., SATI, S., YASUKATA, K., RAICIU, C., AND HUICI, F. My vm is lighter (and safer) than your container. In *Proceedings of the 26th Symposium on Operating Systems Principles* (2017), pp. 218–233.

[51] MARTY, M., DE KRUIJF, M., ADRIAENS, J., ALFELD, C., BAUER, S., CONTAVALLI, C., DALTON, M., DUKKIPATI, N., EVANS, W. C., GRIBBLE, S., KIDD, N., KONONOV, R., KUMAR, G., MAUER, C., MUSICK, E., OLSON, L., RUBOW, E., RYAN, M., SPRINGBORN, K., TURNER, P., VALANCIUS, V., WANG, X., AND VAHDAT, A. Snap: a microkernel approach to host networking. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2019), SOSP '19, Association for Computing Machinery, p. 399–413.

[52] PETER, S., LI, J., ZHANG, I., PORTS, D. R. K., WOOS, D., KRISHNAMURTHY, A., ANDERSON, T., AND ROSCOE, T. Arrakis: The operating system is the control plane. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)* (Broomfield, CO, 2014), USENIX Association, pp. 1–16.

[53] RAHO, M., SPYRIDAKIS, A., PAOLINO, M., AND RAHO, D. Kvm, xen and docker: A performance analysis for arm based nfv and cloud computing. In *Information, Electronic and Electrical Engineering (AIEEE), 2015 IEEE 3rd Workshop on Advances in* (2015), IEEE, pp. 1–8.

[54] RISTENPART, T., TROMER, E., SHACHAM, H., AND SAVAGE, S. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *Proceedings of the 16th ACM conference on Computer and communications security* (2009), pp. 199–212.

[55] SOARES, L., AND STUMM, M. FlexSC: Flexible system call scheduling with Exception-Less system calls. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)* (Vancouver, BC, Oct. 2010), USENIX Association.

[56] WILLIAMS, D., KOLLER, R., LUCINA, M., AND PRAKASH, N. Unikernels as processes. In *Proceedings of the ACM Symposium on Cloud Computing* (New York, NY, USA, 2018), SoCC '18, Association for Computing Machinery, p. 199–211.

[57] WILLIAMS, D., KOLLER, R., LUCINA, M., AND PRAKASH, N. Unikernels as processes. In *Proceedings of the ACM Symposium on Cloud Computing* (New York, NY, USA, 2018), SoCC '18, Association for Computing Machinery, p. 199–211.

[58] XEN. http://www.xen.org/.

[59] YANG, J., KIM, J., HOSEINZADEH, M., IZRAELEVITZ, J., AND SWANSON, S. An empirical guide to the behavior and use of scalable persistent memory. In *18th USENIX Conference on File and Storage Technologies (FAST 20)* (Santa Clara, CA, Feb. 2020), USENIX Association, pp. 169–182.

[60] YAROM, Y., AND FALKNER, K. Flush+ reload: a high resolution, low noise, l3 cache side-channel attack. In *23rd {USENIX} Security Symposium ({USENIX} Security 14)* (2014), pp. 719–732.

[61] YOUNG, E. G., ZHU, P., CARAZA-HARTER, T., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. The true cost of containing: A gvisor case study. In *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)* (Renton, WA, July 2019), USENIX Association.

[62] YOUNG, E. G., ZHU, P., CARAZA-HARTER, T., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. The true cost of containing: A gvisor case study. In *11th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 19)* (2019).

[63] ZHANG, Y., JUELS, A., REITER, M. K., AND RISTENPART, T. Cross-vm side channels and their use to extract private keys. In *Proceedings of the 2012 ACM conference on Computer and communications security* (2012), pp. 305–316.

[64] ZHANG, Y., JUELS, A., REITER, M. K., AND RISTENPART, T. Cross-tenant side-channel attacks in paas clouds. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security* (2014), pp. 990–1003.

DISTRIBUTION STATEMENT A. Approved for public release: Distribution unlimited. Case Number AFRL-2025-0650. Dated 06 Feb 2025.

Appendix A

Table 2: System call classification.

User-host interface (supported)	Delegable (supported)	Non-delegable (supported)	Blocked (not supported)	Blocked (root privilege) (not supported)
timerfd_create	read	mmap	msgget	sched_getaffinity
timerfd_settime	write	munmap	msgsnd	mincore
dup	open	mremap	msgrcv	pause
poll	close	brk	msgctl	vfork
ioctl	stat	mprotect	gettimeofday	times
read	fstat	madvise	setsid	rt_sigpending
write	lstat	rt_sigprocmask	getsid	rt_sigtimedwait
pread64	poll	rt_sigsuspend	capget	rt_sigqueueinfo
msync	lseek	rt_sigreturn	capset	sigaltstack
brk	ioctl	rt_sigaction	acct	utime
futex	pread64	prlimit64	setdomainname	uselib
clock_nanosleep	pwrite64	prctl	iopl	personality
tgkill	writev	clock_nanosleep	ioperm	sysfs
epoll_create	pipe	set_robust_list	io_setup	getpriority
epoll_ctl	dup	rseq	io_destroy	setpriority
epoll_wait	dup2	fork	io_getevents	sched_setparam
readv	getpid	clone3	io_submit	sched_getparam
writev	sendfile	execve	io_cancel	sched_setscheduler
wait4	socket	exit_group	get_thread_area	sched_getscheduler
listen	connect	wait4	lookup_dcookie	sched_get_priority_max
fcntl	accept	futex	remap_file_pages	sched_get_priority_min
rt_sigreturn	sendto	tgkill	semtimedop	sched_rr_get_interval
	recvfrom	msync	settimeofday	mlock
	sendmsg	arch_prctl	fanotify_init	munlock
	recvmsg	alarm	fanotify_mark	mlockall
	shutdown	exit	shmget	munlockall
	bind	setitimer	shmat	vhangup
	listen	getitimer	shmctl	modify_ldt
	getsockname		getrandom	pivot_root
	getpeername		semget	sysctl
	socketpair		semop	adjtimex
	setsockopt		semctl	setrlimit
	getsockopt		shmdt	chroot
	uname		getrlimit	mount
	fcntl		getrusage	umount2
	flock			swapon
	fsync			swapoff
	fdatasync			reboot
	ftruncate			create_module
	getcwd			init_module
	chdir			delete_module
	rename			get_kernel_syms
	mkdir			query_module
	rmdir			quotactl

Continued on next page...

DISTRIBUTION STATEMENT A. Approved for public release: Distribution unlimited. Case Number AFRL-2025-0650. Dated 06 Feb 2025.

User-host interface	Delegable	Non-delegable	Implementable	Blocked
	unlink			nfsservctl
	getppid			getpmsg
	epoll_create			putpmsg
	getdents64			afs_syscall
	fadvise64			tuxcall
	epoll_wait			security
	epoll_ctl			readahead
	openat			time
	newfstatat			sched_setaffinity
	unlinkat			epoll_ctl_old
	pselect6			epoll_wait_old
	sync_file_range			restart_syscall
	timerfd_create			clock_getres
	fallocate			utimes
	timerfd_settime			vserver
	timerfd_gettime			mbind
	accept4			set_mempolicy
	eventfd2			get_mempolicy
	epoll_create1			mq_open
	pipe2			mq_unlink
	statx			mq_timedsend
	access			mq_timedreceive
	gettid			mq_notify
	getdents			mq_getsetattr
	fchdir			kexec_load
	creat			waitid
	link			add_key
	symlink			request_key
	readlink			keyctl
	chmod			ioprio_set
	fchmod			ioprio_get
	chown			inotify_init
	fchown			inotify_add_watch
	lchown			inotify_rm_watch
	umask			migrate_pages
	getuid			unshare
	getgid			get_robust_list
	setuid			splice
	setgid			tee
	geteuid			sync_file_range
	getegid			vmsplice
	setpgid			move_pages
	getpgrp			utimensat
	truncate			inotify_init1
	setreuid			rt_tgsigqueueinfo
	setregid			perf_event_open
	getgroups			name_to_handle_at
	setgroups			open_by_handle_at
	setresuid			clock_adjtime

Continued on next page...

DISTRIBUTION STATEMENT A. Approved for public release: Distribution unlimited. Case Number AFRL-2025-0650. Dated 06 Feb 2025.

User-host interface	Delegable	Non-delegable	Implementable	Blocked
	getresuid			syncfs
	setresgid			setns
	getresgid			getcpu
	getpgid			process_vm_readv
	setfsuid			process_vm_writev
	setfsgid			kcmp
	ustat			finit_module
	statfs			sched_setattr
	fstatfs			sched_getattr
	sync			memfd_create
	sethostname			kexec_file_load
	mkdirat			bpf
	mknodat			userfaultfd
	fchownat			membarrier
	futimesat			mlock2
	renameat			pkey_mprotect
	linkat			pkey_alloc
	symlinkat			pkey_free
	readlinkat			io_pgetevents
	fchmodat			seccomp
	faccessat			sched_yield
	ppoll			nanosleep
	epoll_pwait			ptrace
	dup3			tkill
	preadv			syslog
	pwritev			set_tid_address
	recvmmsg			set_thread_area
	sendmmsg			execveat
	renameat2			kill
	copy_file_range			clock_settime
	preadv2			clock_gettime
	pwritev2			timer_create
	sysinfo			timer_settime
	select			timer_gettime
	mknod			timer_getoverrun
	readv			timer_delete
	setxattr			signalfd
	lsetxattr			eventfd
	fsetxattr			signalfd4
	getxattr			
	lgetxattr			
	fgetxattr			
	listxattr			
	llistxattr			
	flistxattr			
	removexattr			
	lremovexattr			
	fremovexattr			

DISTRIBUTION STATEMENT A. Approved for public release: Distribution unlimited. Case Number AFRL-2025-0650. Dated 06 Feb 2025.

Accelerating Nested Virtualization with HyperTurtle

Ori Ben Zur[1], Jakob Krebs[1], Shai Aviram Bergman[2], and Mark Silberstein[1]

[1]Technion - Israel Institute of Technology
[2]Huawei Zurich Research Center

Abstract

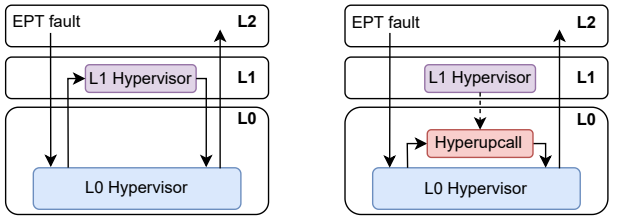
Nested virtualization provides strong isolation but incurs non-trivial performance costs. Prior works alleviate some overheads but suffer from limitations such as intrusive code changes or reduced control over nested virtual environments. We present HyperTurtle, a general approach to accelerate nested virtualization. It reduces the number of costly world switches between the virtualization layers, the primary source of performance overheads. HyperTurtle *offloads* the execution of certain parts on the critical path of the virtualized hypervisor, encapsulating them as *eBPF* programs and executing them safely in the context of the bare-metal hypervisor. Thus, HyperTurtle reduces the performance cost of world switches whilst retaining control over nested VMs. We show that HyperTurtle can be used to optimize a variety of OS subsystems and apply it to memory management, networking, and application profiling. HyperTurtle achieves significant performance improvements in micro and macro-benchmarks, for example, 5× faster EPT fault handling, which translates to up to 27% faster boot-time of Kata containers, without requiring intrusive code changes to the virtualization infrastructure.

1 Introduction

Virtual machine-based containers that rely on *CPU virtualization capabilities* to achieve isolation offer a more secure alternative to process-level containers. For example, Kata containers, used by several cloud vendors [1, 42, 64], are lightweight VMs that expose the Kubernetes management interface to facilitate adoption. Kata containers offer strong isolation to allow tenants to execute untrusted third-party code, which is a central requirement of serverless runtimes [1] in cloud computing systems.

In practice, VM-based containers are often deployed as *nested* VMs [31, 50, 57, 58], which is convenient for consolidating workloads and provides more flexibility. In such a nested virtualization scenario, a guest VM (L1) runs its own hypervisor to manage container VMs (L2).

Unfortunately, nested virtualization incurs substantial performance overheads [11], mainly due to the excessive number of transitions between the virtualization layers, commonly called *world switches* [11]. The transitions are necessary since the hypervisor running in a virtual machine needs to execute privileged instructions to manage the nested VMs, but it cannot do so without the bare-metal hypervisor [11].



(a) Vanilla nested virtualization

(b) Nested virtualization with HyperTurtle

Figure 1: The world switch problem in nested virtualization. When a nested virtual machine (L2) performs a `vmexit`, the virtualized hypervisor (L1) must handle it. However, since it is virtualized itself, the event handling is forwarded to L0, forcing an additional world switch. HyperTurtle reduces the number of the world switches by offloading the handling of such events as eBPF programs to L0.

Fig. 1a illustrates the world switch problem. Every time the execution of the nested VM (L2) requires the hypervisor's intervention (e.g., EPT faults, emulation of hardware features), L2 transitions to the virtualized hypervisor (L1). This transition requires forwarding of the `vm-exit` to the bare-metal hypervisor (L0). Hence, there are *at least four* world switches overall. In contrast, non-nested virtualization requires only two world switches. Moreover, in some common performance-critical cases, the number of world switches is higher. For example, EPT fault handling requires *six* world switches if the physical memory of L2 is not present (see §4.3), vs. only two switches in the non-nested case.

These additional world switches cause substantial performance degradations. For example, prior work has shown that each world switch takes about 1μs, and comprises about 33% of the total `vm-exit` handling time [60].

Moreover, as we explain in §3, some of L0's `vm-exit` handlers are much more expensive in the nested scenario vs. the non-nested one, e.g., 3.3× (0.78μs vs. 2.77μs) and 2.9× (5.7μs vs. 16.7μs) for `cpuid` hypercall and non-maskable interrupt handling respectively (see Tab. 1), thus their negative impact on the nested VM performance is even higher.

Prior work has explored several techniques for reducing nested virtualization overheads [11, 31, 45, 58, 60]. Direct Virtual Hardware (DVH) [45] bypasses L1's I/O stack by exposing virtual devices from L0 directly to L2, similar to passthrough in non-nested setups. DVH improves I/O performance, but retains other nesting overheads and prevents L1 from enforcing policies on I/O paths such as traffic shaping [17]. Other systems that use hardware-assisted virtualization, such as SVT [60], reduce switching overheads by co-scheduling L1 and L2 on sibling hyperthreads, but rely on simultaneous multithreading (SMT), which is being phased out and becoming unavailable [2,36]. Software-based systems like PVM [31], X-Containers [57], and CKI [58] decouple L0 from nested execution through syscall redirection, library OSes, or hardware-isolated compartments within L1, but introduce compatibility and performance issues or require hardware changes. Finally, Peer-Pods [5], Free-The-Turtles [63], and Dichotomy [62] sidestep nesting entirely, at the cost of hypervisor-level control and orchestration flexibility.

We present **HyperTurtle**, a general approach to accelerating nested virtualization with minimal code modifications to both L0 and L1. HyperTurtle reduces the number and cost of world switches by offloading certain parts of the L2 `vm-exit` handling logic from L1 to L0 (see Fig. 1b). Notably, it encapsulates relevant functionality as an *eBPF* program loaded into and executed by L0 when handling L2's `vm-exits`. To achieve this, HyperTurtle enables L1 to hook into L0's `vm-exit` handling logic, which executes the eBPF that implements the relevant functionality of the L1 hypervisor, such as EPT fault handling. Not only does this mechanism eliminate the respective world switches, but it also obviates the need for emulating CPU virtualization features necessary for handling the event in L1. The eBPF bytecode undergoes the standard kernel verification when loaded by L0 to ensure safe execution. Thus, HyperTurtle improves the performance of nested virtualization with minor changes to L0 and L1 hypervisors, it is hardware agnostic, and keeps L1 in full control of L2.

The idea of offloading guest functionality to a hypervisor via eBPF was first introduced in Hyperupcalls [4]. However, adopting it to accelerate nested virtualization poses a few non-trivial challenges. First, eBPF programs cannot use locks or unbounded loops and cannot perform arbitrary memory accesses due to safety constraints, complicating the implementation of complex functionality such as EPT fault handling. Second, L1's address space and data structures are not accessible to eBPFs running in L0. Last, a naive approach where L0 may host eBPFs that belong to different L1 hypervisors may compromise isolation. For example, attaching a L1's eBPF program to L0's physical network interface enables it to receive the network traffic of L0 as well as other L1's.

HyperTurtle solves these challenges, demonstrating how eBPFs can be used to accelerate three specific subsystems of high practical importance where nested virtualization incurs particularly high overheads: EPT fault handling, network I/O, and application performance profiling. In each case, we show how to expose the relevant L1's state to an eBPF program via *eBPF maps*, develop *helper functions* that enable direct and safe access to L1's data structures from the eBPF context, implement the respective eBPF program, and leverage HyperTurtle infrastructure to safely attach it to the appropriate hooks in the L0 kernel.

We show that HyperTurtle offers substantial performance benefits in all three scenarios, and provides a generic way to reduce both `vm-exits` and I/O overheads. Specifically, moving the EPT fault handling into L0 allows L2 physical memory to be paged in without trapping to L1, resulting in a 5.1× reduction in EPT fault handling latency on average. This reduction accelerates Kata container launch times by an average of 27%.

Running L1-controlled networking with eBPF in L0 enables the usage of DVH without compromising L1's control over L2 networking. This enables 33% average latency reduction compared to the two-level network virtualization previously required to achieve the same level of control over L2 network stack. Thus, HyperTurtle achieves a 72% throughput increase at 500μs in the memcached latency-throughput benchmark, making the performance of nested Kata containers on par with regular containers, while retaining full compatibility with the Kubernetes Container Network Interface despite the use of DVH. Lastly, HyperTurtle enables profiling nested virtual machines from L1 with a reduction in sampling latency by 7×.

Importantly, HyperTurtle is practical and easy to maintain, as it achieves high performance with modest modifications limited mostly to the KVM module, and its most complex EPT fault handling hyperupcall is only 560 LOC.

The HyperTurtle source code and evaluation scripts are available at https://github.com/acsl-technion/hyperturtle.

2 Background

This section introduces the necessary background in nested virtualization and the previous work we build upon.

2.1 Nested Virtualization

We refer to a bare-metal hypervisor, a virtualized hypervisor running in a virtual machine, and a guest virtual machine managed by the virtual hypervisor as L0, L1, and L2, respectively.

In nested virtualization, L1 cannot directly access the CPU's virtualization hardware extensions, which are only available to L0. Thus, L0 emulates these features for L1 by trapping virtualization-related instructions executed by L1.

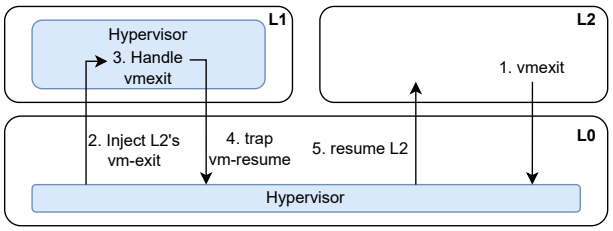


Figure 2: L0 runs the nested VM on L1's behalf. L0 injects all events that cause L2 to exit into L1.

Effectively, this results in L0 running L2-related procedures on L1's behalf, forwarding all virtualization-related events to L1. This process is entirely transparent to L1 and L2. When L2 triggers a `vm-exit`, the CPU exits to L0, which injects the event to L1. This emulation doubles the number of world switches compared to non-nested virtualization. Fig. 2 illustrates the event forwarding. Often, the number of world switches may be higher (as, for example, in the case of EPT faults), and their costs may differ. We discuss this in §3.

Memory management in nested virtualization is also more complex. In non-nested virtualization, L0 maintains L1's physical memory space, mapping guest physical addresses (gPA) memory to host physical (hPA). However, an additional level of indirection is necessary for nested virtualization, as L2 addresses are mapped to L1 addresses, which are then mapped to L0 addresses. The guest's memory can be backed by anonymous, or file-backed memory via a shared-memory file system such as `shmem`. File-backed memory enables the *Virtual Machine Monitor* (VMM) to share the memory of the guest with another process. Kata containers utilize this with virtiofsd [61] for file system virtualization, providing a shared file system between the containers and the host.

Similarly, device virtualization adds another level of indirection. Consider, for example, the virtualization of a Network Interface Card (NIC). VirtIO [56] is a popular choice of the network para-virtualized device in non-nested virtual systems, as it facilitates VM migration and network changes without the guest's cooperation. L0 exposes the NIC backed by VirtIO to L1 VMs. This para-virtual approach requires the translation of I/O events from physical devices in the host to virtual devices. In the nested virtualization scenario, there are two layers of para-virtual devices, one managed by L0 exposed to L1, and another by L1 exposed to L2.

2.2 eBPF

eBPF (extended Berkeley Packet Filter) allows users to dynamically load and safely run code in the kernel. To use eBPF, the user compiles restricted C code into eBPF bytecode. An object file with the bytecode can be loaded into the kernel and linked (or *attached*) to pre-existing function calls, *hooks*, in the kernel. The Linux kernel exposes many hooks. In particular, eXpressDataPath (XDP) and Traffic Control (TC) are popular hooks in the network stack, used, for example, to implement DDoS protection [21], load balancing [49], and observability [14]. eBPF uses key-value data structures, called *shared maps*, to share data between eBPF programs and user-space applications. The eBPF program defines the maps and the kernel allocates them when loading the program.

The kernel has a built-in verifier that checks the safety of eBPF programs. However, the verification restricts the expressiveness of eBPF programs. The verifier checks the bounds of memory accesses, disallows unbounded loops, and restricts the use of locks, e.g., by non-root users.

Helper functions allow the eBPF programs to request services from the kernel or query its data structures [46]. The Linux kernel exposes a limited set of functions to eBPF programs, which can be divided to 5 main types: (i) map access (`bpf_map_lookup_elem()`) (ii) time & perf utilities (`bpf_ktime_get_ns()`, `bpf_perf_event_read_value()`) (iii) network stack integration (`bpf_redirect()`) (iv) syscall–like services (`bpf_get_current_pid_tgid()`) (v) arbitrary data access (`bpf_probe_read()`).

Not all functions are available to every eBPF program. E.g., a networking-related helper function is not available to a tracing-type eBPF program. Moreover, a program's set of available functions can be restricted with the Linux capability system [47], and some helper functions require special permissions (e.g., arbitrary data access helper functions).

2.3 Kata Containers

Tenants of public cloud systems may serve as service providers to third-party users [3]. While containers are common and the most convenient way to manage computational resources for such services, their isolation guarantees are often insufficient to run third-party code. Kata containers solve this issue. They offer the level of isolation of hardware virtualization but retain the same control interface as regular containers. They execute an application in a lightweight virtual machine, yet run a special agent to allow access to that application and maintain the management interface compatible with popular container management systems such as Kubernetes [42].

From the cloud provider's perspective, Kata containers run as nested virtual machines. Unfortunately, the overheads associated with such deployments are too high, up to 3× slower [45, 57] than non-nested setups, forcing users to sacrifice stronger isolation advantages for better performance. Thus, HyperTurtle's primary goal is to improve the performance of nested virtualization and make it an attractive alternative to containers.

3 Analysis

We first analyze the latency of world switches in isolation, and then show their performance impact in three important practical cases: EPT fault handling, networking and profiling. We use the same evaluation setup as in §5.

Exit Reason	L1 → L0 [μs]	L2 → L0 [μs]
nop hypercall	0.85	2.82
`vm-write`	1.1	—
`vm-enter`	8.0	—
`cpuid`	0.78	2.77
EPT Fault	4.8	3.7 ± 0.8
Non-maskable interrupt	5.7	16.7

Table 1: L0 `vm-exit` handling duration, measured from `vm-exit` of the VM to the next `vm-enter`. All measurements are with CoV $< 1\%$, except for EPT fault from L2.

3.1 Understanding World Switch Costs

World switches between L0, L1 and L2 may occur in a variety of execution scenarios, such as the invocation of privileged CPU instructions by L2 or L1. Notably, L2 always traps into L0, which forwards the event to L1. Some world switches are voluntary, i.e., when para-virtual I/O devices notify the L0 to perform an I/O event. This notification is implemented as a hypercall, which initiates a voluntary world switch.

At the CPU level, world switches require flushing of the CPU execution pipeline, saving the VM's registers into the Virtual Machine Control Structure (*VMCS*) data structure, and loading the hypervisor registers from the VMCS structure. These operations lead to non-negligible overheads. Vilanova et al. [60] show that these overheads amount to about 33% of a 10μs-long emulation of the `cpuid` instruction.

Moreover, the handling latency of the `vm-exit` associated with each world switch may vary significantly depending on the cause of `vm-exit`, due to the execution of different code paths. Tab. 1 shows the latency of handling `vm-exits` in a few different scenarios. For example, handling a non-maskable interrupts is the most expensive. Notably, most transitions from L2 to L0 are more expensive than those from L1. This is caused by the need to inject the `vm-exit` from L0 to L1 and the manipulation of many of L0's own data structures.

Takeaway: World switches are expensive, and their latency varies depending on their cause. They are more expensive when triggered from L2 than from L1.

So far, we have discussed the costs of world switches in isolation. Next, we focus on three specific scenarios to show how world switches affect their end-to-end performance in a nested virtual machine.

3.2 EPT Faults

Memory management is a key culprit of the nested virtualization tax, as the additional virtualization layer triples the number of world switches required to handle Extended Page Table (EPT) faults. To understand the origin of this overhead, we detail the implementation of the EPT fault mechanism for a nested guest in KVM.

Fig. 3 illustrates the hierarchy of page tables in nested virtualization. As with the non-nested virtualization, L0 maintains

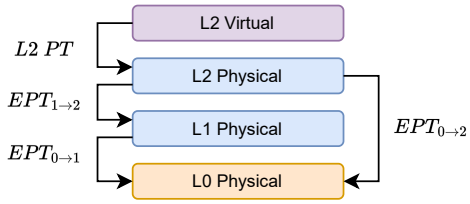
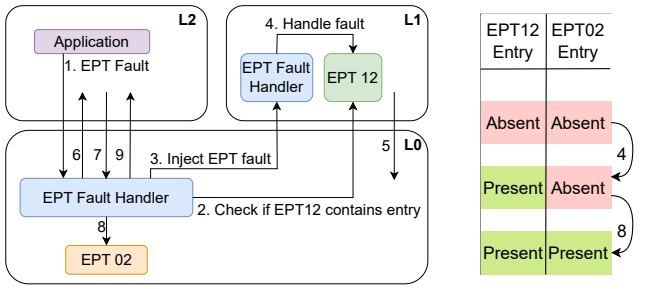


Figure 3: Page table hierarchy in nested virtualization



(a) EPT fault handling in nested virtualization, see the text for details.

(b) L2-PA mapping. Steps refer to (a).

Figure 4: EPT fault handling in nested virtualization

an EPT, called $EPT_{0\to1}$ [1], from L1 physical addresses (L1-PA) to L0 physical addresses (L0-PA). However, unlike non-nested virtualization, L1 maintains an EPT, called $EPT_{1\to2}$, from L2 physical addresses (L2-PA) to L1 physical addresses (L1-PA). As CPUs do not support a 3-dimensional page-walk, L0 *shadows* $EPT_{1\to2}$ and maintains $EPT_{0\to2}$ which translates L2-PAs directly to L0-PAs. L0 lazily creates $EPT_{0\to2}$ while running L2.

Fig. 4a walks through the EPT fault handler, while Fig. 4b illustrates the state of $EPT_{1\to2}$ and $EPT_{0\to2}$ throughout. L2 encounters an EPT fault due to a missing $EPT_{0\to2}$ entry, which triggers a `vm-exit` to L0 ❶. L0 checks if a valid mapping appears in $EPT_{1\to2}$ ❷. If it is missing, L0 injects the EPT fault into L1, and enters L1 immediately thereafter ❸. L1 handles the EPT fault by mapping the address in $EPT_{1\to2}$ ❹. L1 tries to resume L2, trapping again into L0 ❺. L0 enters L2 ❻, which re-executes the instruction that triggered the EPT fault. Since $EPT_{0\to2}$ still contains a non-present entry, another `vm-exit` occurs ❼. L0 once again walks $EPT_{1\to2}$ ❽, finds the entry, and fixes $EPT_{0\to2}$ with the value from $EPT_{1\to2}$. L0 then re-enters L2 ❾, resuming execution. This procedure requires a total of *six* world switches, compared to only *two* in non-nested virtualization.

We measure the EPT fault latencies incurred by a VM when faulting in 1GiB of consecutive memory with and without nesting. Tab. 2 shows the results. We observe that nested EPT faults experience 5.1× and 5.3× latency increase over the non-nested scenario for the average and 99p latencies, respectively, demonstrating the costs of additional world switches.

[1] This notation is adopted from [11]. The first digit is the virtualization layer that manages the table for the layer denoted by the second digit.

	L2 VM	L1 VM	Slowdown
Average Latency [us] ↓	28.39	5.58	5.1×
99p Latency [us] ↓	49.13	9.35	5.3×

Table 2: EPT-fault Latencies in demand paging (lower is better).

Impact on startup time. EPT faults occur most frequently early in the VM's life cycle because memory has not been allocated yet. Therefore, the EPT fault handling performance directly impacts the startup time. In the context of nested virtualization, this metric is particularly important for Kata containers, as their usage scenarios are identical to regular containers, i.e., short lifetime and on-demand dynamic deployment [1], thus requiring low startup latency.

To demonstrate the impact of the EPT fault handling on the boot time of a nested VM, we measure the startup time of a Node.js application with a Linux Ubuntu image in a Kata container. We define *startup time* as the time from issuing the startup command until the application responds to a network request. Starting the Kata container in a non-nested scenario takes 0.7 seconds, compared to 1.5 seconds as a nested VM. The latency breakdown shows that the EPT faults account for 14% (0.1 seconds) and 46% (0.69 seconds) in the non-nested and nested cases respectively, highlighting the importance of reducing the EPT fault handling costs.

Takeaway: World switches cause large overheads in EPT fault handling in nested virtual machines, with a significant end-to-end impact, e.g., on the startup time.

3.3 Networking

Network I/O from L2 to the world outside the physical server must be performed via L0 which manages physical NICs. In a simple approach, called `Nested-VirtIO`, L1 mediates between L2 and L0. L0 exposes a virtual NIC to L1 ($vNIC_{01}$), which in turn creates its own vNIC ($vNIC_{12}$) and exposes it to L2. Under the hood, L1 connects and forwards traffic between $vNIC_{01}$ and $vNIC_{12}$. Both vNICs are virtioNICs.

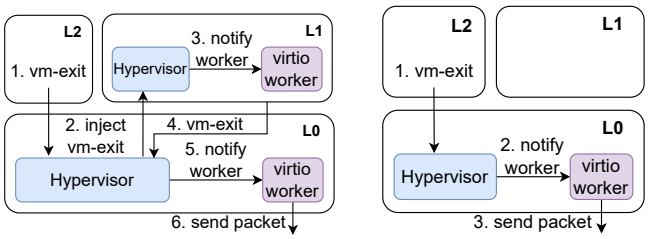
A more sophisticated approach, called *Direct Virtual Hardware*, `Direct-Assignment` [45], utilizes QEMU's IOMMU emulation, where L0 exposes multiple vNICs to L1, allowing it to connect $vNIC_{01}$ directly to L2, effectively removing L1 from the L2 network data path. Indeed, Tab. 3 shows significant performance benefits of `Direct-Assignment` over `Nested-VirtIO` - with latency reduced by 40% and 37% for mean and 99p respectively, and throughput increased by 44%. Both setups utilize the vhost-net [59] driver in the hypervisors.

To understand this performance gap, we review the chain of events when L2 sends a single packet via `Nested-VirtIO` and via `Direct-Assignment` in Fig. 5a and Fig. 5b respectively.

In `Nested-VirtIO`, ❶ L2 executes an MMIO write to notify $vNIC_{12}$ of the pending packet, which triggers `vm-exit` to L0. ❷ L0 injects `vm-exit` to L1's hypervisor, which forwards L2's request to ($vNIC_{12}$)'s virtio worker thread ❸. The latter

	Nested-Virtio	Direct-Assignment	Relative
Avg Latency [us] ↓	90.21	54.52	0.60×
99p Latency [us] ↓	106	67	0.63×
Throughput [Gb/s] ↑	12.1	17.4	1.44×
CPU Usage [%] ↓	100	196	1.96×

Table 3: UDP round-trip latency of packets with 1-Byte payload using netperf; TCP throughput with iperf3.



(a) First packet transmission with `Nested-VirtIO`

(b) First packet transmission with `Direct-Assignment`

Figure 5: Comparison of single packet transmission in `Nested-VirtIO` and `Direct-Assignment`

tries to transmit the packet ❹, triggering a `vm-exit` from L1 to L0 due to a similar MMIO write. L0's hypervisor forwards the request to $vNIC_{01}$'s virtio worker ❺ which sends it via the physical NIC ❻.

This process is much shorter in `Direct-Assignment`: ❶ L2 executes an MMIO write to notify ($vNIC_{02}$), which triggers a `vm-exit` to L0. L0's virtio worker sends the packet ❷ via the physical NIC.

The overheads of `Nested-VirtIO` stem from the higher number of world switches: we observe a total of *three* world switches to transmit a packet using `Nested-VirtIO` compared to *one* world switch with `Direct-Assignment`.

Unfortunately, `Direct-Assignment` effectively disables the control of L1 over L2 networking, by removing L1 from the L2 network control path. Thus, L1 cannot affect the policies of network management operations for its L2 guests, such as routing, monitoring, and traffic shaping, which are essential for the proper operation of containers.

Takeaway: World switches in L2 network stack cause significant overheads. `Direct-Assignment` *reduces the overheads, but it does not allow L1 to control L2's network policies.*

3.4 Application Profiling

Profiling is broadly used to find bottlenecks in development and production systems, with some vendors, such as Netflix [30] and Google [55] continuously profiling their production applications. Moreover, all large cloud providers, such as AWS, Google Cloud, and Azure [8, 20, 27, 28] offer profiling utilities to their tenants.

Setup	L0 → L1	L1 → L1	L1 → L2
Latency [μs] ↓	5.66	25.63	60.68

Table 4: Profiler sampling latency in different settings.

Sampling profilers periodically interrupt a program and sample the programs state, as well as hardware performance monitoring counters (PMCs). Notably, sampling-based profilers are popular because they introduce minimal execution overhead, causing between $1\% - 3\%$ slowdown for most applications [20, 38].

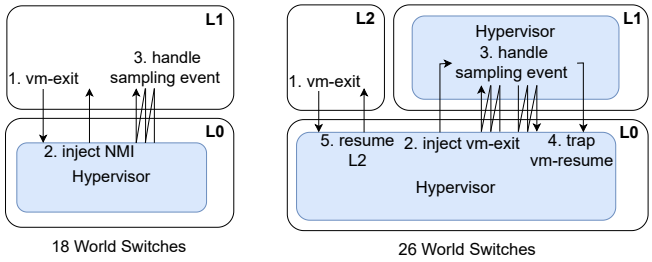
Each *sampling event* begins with an interrupt, caused by the overflow of a PMC, which counts cycles since the last event and ends when the profiled program resumes execution. During the event, the state of the running program is sampled and the PMC is reprogrammed. From a performance perspective, profiling an application inside a container and running on bare-metal hardware is the same as profiling a regular application. Profiling an L1 from L0 requires a single `vm-exit` for each sampling event.

However, profiling inside a VM causes a dramatic slowdown. Tab. 4 compares the average latency of handling a sampling event when profiling L1 from L0 ($L0 \rightarrow L1$), a container from L1 ($L1 \rightarrow L1$) and a nested VM from L1 ($L1 \rightarrow L2$). We observe that the latency of ($L1 \rightarrow L2$) is about twice as large as the latency of ($L1 \rightarrow L1$), and almost $8\times$ higher than profiling a VM from L0.

We now analyze the sequence of events that occur when profiling a container (Fig. 6a) and a nested VM (Fig. 6b) from L1. In the former case, KVM emulates the HW counters by exposing virtual registers. L0 traps the L1 when the latter tries to access them. The flow is as follows: ❶ The HW counter triggers an interrupt, which triggers a `vm-exit` to the host (L0). ❷ The host injects a similar interrupt to the L1, and enters L1. ❸ As part of the interrupt handling routine, L1 reprograms the HW counter and performs multiple accesses to MSRs, bringing the total world switch count to *18*.

Profiling an application in a nested virtual machine incurs higher overheads. ❶ The HW counter triggers an interrupt, which triggers an exit from L2 to L0. ❷ L0 injects a similar interrupt to L1, and enters L1. ❸ As part of the interrupt handling routine, L1 reprograms the HW counter, accesses MSRs and performs the `vmread` and `vmwrite`, which access the VMCS, overall triggering 22 world switches. ❹ L1 executes the `vm-enter` instruction, which triggers an exit to L1. ❺ L0 emulates the `vm-enter` instruction and enters L2, which brings the total number of world switches to *26*.

Interestingly, the performance discrepancy between profiling a nested VM or a container stems not only from the number of world switches but also from their higher costs for the nested case. Our measurements show that 7.8 μs are spent by L0 injecting the NMI into L1, additional 8μs on the emulation of L1's execution of the `vm-enter` instruction.



(a) Profiling a container in L1 (b) Profiling a Kata container in L1

Figure 6: Comparison of world switches when profiling a standard container versus a Kata container from L1.

Takeaway: Excessive world switches and their increased cost cause high overhead when profiling an L2 VM from L1.

These diverse examples motivate us to build a generic infrastructure to reduce the aforementioned overheads of nested virtualization, as we describe next.

4 Design & Implementation

HyperTurtle pursues the following design goals:

1. **Performance:** Eliminate excessive world switches,
2. **Control:** Retain L1's ability to enforce policies on L2,
3. **Non-intrusiveness:** Minimize L0 modifications,
4. **Generality:** Provide a hardware-agnostic approach generalizable to optimize different OS subsystems.

HyperTurtle offloads a subset of L1 hypervisor functionality into L0 thus minimizing excessive world switches (Goal 1). HyperTurtle leverages *hyperupcalls*, i.e., eBPF programs, to safely execute L1's code in L0. Thus, L1 fully controls the offloaded logic executed in L0, implying that it retains full control over L2 (Goal 2). As the eBPF subsystem already exists in L0's kernel, the only L0 modifications required are the export of nested virtualization-related information to be accessible to the hyperupcall eBPF function via helper functions, and the addition of an installation mechanism for eBPF programs from L1, achieving Goal 3. Last, HyperTurtle's implementation is modular, making it applicable to accelerate different subsystems according to the user's choice, as we empirically demonstrate in the rest of the paper (Goal 4).

We focus our efforts on KVM, as the da-facto standard for virtualization in Linux, however, we believe that the design principles are applicable to Xen-based VMs. Additionally, we limit our design to a single level of nesting, leaving further design choices, such as the choice of the layer to which a hyperupcall should be installed for future work.

4.1 Design Overview

HyperTurtle comprises the eBPF offloading infrastructure, new hooks in L0, new eBPF helper functions and the mechanisms for sharing data between L1 and eBPFs.

The offloading mechanism allows L1 to install the relevant eBPF in L0 via a hypercall. It exposes two main features: *load* a program into L0 and *link* the program to the hook.

A program is first loaded into L0 and verified by its eBPF verifier. L1 then attaches the hyperupcall to a hook in L0, which is invoked as part of L0's logic for handling L2 events.

Design recipe. Each subsystem requires its own design to reduce the associated world switch overheads via eBPFs. However, each such design follows a general recipe which comprises three key steps:

1. **Sharing information:** Exposing necessary information between L1 and the eBPF program for its operation. L1 might need to be modified and L0 might need to expose new helper functions to assist the hyperupcall.
2. **Hook into L0:** Implementing a hook in the L0 hypervisor to intercept and handle specific events.
3. **Develop eBPF program:** Writing the desired functionality as an eBPF program that will run in its own isolated context from L0.

We now explain the infrastructure shared by all hyperupcalls and then delve into the eBPFs for accelerating EPT fault handling, network control of L1, and application profiling.

4.2 Hyperupcalls Infrastructure

eBPF maps and L1 memory access. HyperTurtle employs shared eBPF maps for L0-L1 data sharing, instead of the packet-based communications in Hyperupcalls, to support concurrent access to shared data. To achieve this, HyperTurtle adds a new hypercall to create a shared-memory PCI device which exposes array eBPF maps from L0 to L1.

In addition, we introduce a new helper function in L0, `bpf_probe_read_hyperupcall`, to enable eBPFs to access L1's physical memory. This function is safe, in that it verifies that the address is valid and mapped to L1. In addition, it guarantees isolation between the eBPFs installed by different L1 VMs, ensuring that each eBPF can only access the memory regions of the L1 to which it belongs.

Hyperupcall registration. An L1 guest registers a hyperupcall to the L0 hypervisor via a hypercall interface. L1 provides the compiled eBPF program and the hook to attach it. L0 uses the existing eBPF registration mechanism in the Linux kernel to verify, load and link the hyperupcall. For improved security, L1 might also request to install an eBPF from a set of eBPFs vetted by L0 in advance.

In contrast to the original hyperupcall infrastructure where the registration is implemented in a single hypercall, we separate the loading and linking into two hypercalls. This allows linking a single hyperupcall to multiple hooks and avoids duplicate loading, e.g., for networking-related hyperupcalls.

Security. HyperTurtle may be used with two security models: 1. L1-provided eBPFs secured by L0's eBPF verifier or stricter alternatives such as Prevail [25]. 2. Cloud vendor-vetted eBPFs, loaded only if signed and verified (e.g., via Hornet [13]). The latter model provides more strict security where necessary. With both models, HyperTurtle ensures eBPFs can access only their own L2s and memory; and subsystem-specific isolation mechanisms are detailed in the following sections.

4.3 EPT Fault Handling

To combat high EPT fault latencies, we implement a hyperupcall which can map a non-present $EPT_{1\rightarrow2}$ entry during a nested guest's EPT fault. Our implementation targets the *file-backed* memory virtualization backend, which is commonly used in QEMU and Cloud Hypervisor [33], and adopted by Kata Containers [42]. It relies on the `shmem` virtual file system, and enables shared memory between the host and guest. This backend typically works in conjunction with a user-space daemon, such as virtiofsd [61], to support shared file systems outside the VMM (see §2.1).

4.3.1 Sharing Information

The key design challenge of the EPT fault hyperupcall is managing the shared state between L1 and the hyperupcall running in L0. The design must account for L1 physical memory allocation, race conditions, as well as the consistency of L2's physical memory mappings that may arise as the VMM and other processes access L2's memory (e.g., virtiofsd [61]).

Memory allocation. Since allocating memory from the hyperupcall's eBPF context is too complex, we use a pre-allocated *memory pool*. Specifically, L1 maintains a pool of allocated frames in an eBPF map which is accessible from the hyperupcall. The hyperupcall falls back to the traditional EPT fault mechanism if the pool is empty. L1 replenishes the memory pool either asynchronously using a background kernel thread or synchronously in batches when L1 is called to serve a `vm-exit`. Our experiments show that using a pool size of 4096 frames suffices to avoid synchronous replenishment. We therefore use a 4096 frame pool size throughout the paper.

Consistency with L1. The hyperupcall must prevent potential inconsistencies between $EPT_{1\rightarrow2}$ and other data structures in L1, such as the VMM's page table and L1's KVM internal data structures. For example, due to races on L1 virtual addresses (L1-VA) which correspond to an L2-PAs.

To notify L1 of changes the hyperupcall makes to $EPT_{1\rightarrow2}$, it exposes a cyclic fault log called the *hyperupcall fault log* to L1 via a shared map, to which it logs the L2-PA it maps. On the next `vm-exit` entering L1, L1's KVM faults the L1-VAs corresponding to the entries in the fault log into the VMM's address space. If the log is full, the hyperupcall falls back to the original EPT fault mechanism. To make the hyperupcall aware of mappings performed by the VMM, the VMM's `cr3` register is copied to a shared map. Access to the `cr3` enables the hyperupcall to walk the VMM's page table using the `bpf_probe_read_hyperupcall` helper function and check whether the faulted address has a preexisting mapping.

To prevent race conditions, we wrap the hyperupcall with a lock. A failure to acquire the lock results in a fallback to the traditional EPT fault mechanism due to the limitations of eBPF lock implementations (see §2.2).

The lock is exposed to L1 via a shared map, allowing L1's KVM to acquire the lock when handling an EPT fault in L1.

Finally, to create a coherent memory mapping between the hyperupcall and users of the L2's file backed memory (e.g., L1's hypervisor and the virtiofsd daemon), we *wrap* the `shmem` page fault handler with the following logic: (1) Acquire the hyperupcall's lock (spinning if necessary). (2) Walk $EPT_{1\to2}$ to check if the hyperupcall already mapped this page. If it has, map it to L1. (3) Call the original page fault handler. (4) Log the L2-PA which corresponds to the fault to a cyclic buffer, called *page fault log*, which is also a shared map. Step (2) is necessary because the hyperupcall might have created a mapping that is not yet visible from L1 (because an exit from L2 to L1 hasn't occurred yet). To minimize its size, the VMM purges the log on the next `vm-exit` from L2. The maximal log size is equal to the number of L2 frames to prevent overflows. Future work may optimize this design.

4.3.2 L0 Hook

We place the hyperupcall hook in L0's handling of L2's EPT fault, during step ❷ from Fig. 4a, which allows L0 to call the hyperupcall upon the discovery of a non-present entry in $EPT_{1\to2}$. The hyperupcall then maps a new L1 frame to L2, and allows L0 to finish the fault handling and resume the execution of L2, avoiding excessive world switches.

4.3.3 eBPF Program

The hyperupcall receives the faulting L2-PA as well as the fault type (read or write), and then performs the following steps, wrapped within the hyperupcall lock:

1. Walk L1's VMM's page table to check if a mapping exists for the L1-VA that corresponds to the L2-PA that triggered the fault. If a mapping exists, return an EPT entry pointing to the appropriate L1-PA.
2. If the L2-PA is in the page fault log (faulted in by an external process in L1), fall back to the traditional EPT fault mechanism.
3. Create a new EPT entry using a frame from the hyperupcall's memory pool.
4. Write the new entry into $EPT_{1\to2}$ and the fault log.

After the hyperupcall returns, L0 continues to handle the EPT fault as $EPT_{1\to2}$ is now present.

4.4 Fast L2 Networking with L1-Controlled Policies

The `Direct-Assignment` method, introduced by DVH [45], improves performance, but takes away L1's control over L2's network traffic. HyperTurtle allows L1 to install eBPF programs on L0's network interfaces, enabling it to regain control by attaching hyperupcalls to the virtual devices used by L2. While this represents a relatively simple use case for hyperupcalls, it demonstrates how HyperTurtle can restore control without sacrificing performance.

Fig. 7 explains the operation of the networking subsystem using `Direct-Assignment` with eBPFs in the L2's network data path. As per the general recipe (see §4.1) — we *share*

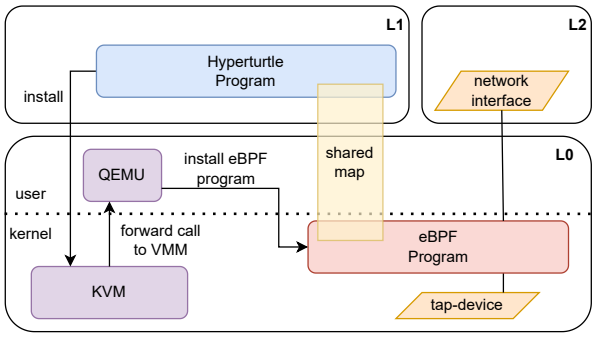


Figure 7: Networking eBPF in the L2 data path

information via shared maps and *hooks* are supplied by L0's eBPF infrastructure [15]. We *implement* the following proof-of-concept eBPF programs to showcase L1's ability to control L2 network policies.

- *Pass* simple eBPF program that does not perform any processing,
- *Stateless Firewall* filters packets based on their source IP address,
- *Rate Limiter* caps the rate of L2's incoming traffic based on data throughput and packet count. The limit can be dynamically configured at runtime by L1,
- *TCP-Top* allows L1 to monitor all TCP connections of L2. It is a modified version of inspector-gadget's `tcptop` program [34].

Isolation & Multiple L2s. HyperTurtle ensures that the eBPF programs can run on the data path of virtual interfaces owned by the L1 VM that installed them. Thus, it guarantees that L1 cannot interpose on the traffic on the interfaces it does not own. Moreover, HyperTurtle utilizes the available per-vNIC hooks to allow L1 to load a different eBPF program to each L2 it manages.

4.4.1 Compatibility Layer with Container Management

Kubernetes (k8s) is a prime candidate as a cluster management tool for nested VMs [23, 42]. In particular, Kata containers support management by k8s, but using `Direct-Assignment` breaks the compatibility with k8s and prevents L1's control over the k8s network required for monitoring and policing with tools such as Clilium [14].

To deploy HyperTurtle in k8s clusters, we integrate dynamic `Direct-Assignment` interface management via a custom *Container Network Interface (CNI)* plugin [7]. The plugin implements configuration of a k8s network by creating and configuring network interfaces as requested by k8s in the initialization process of a k8s pod (management unit of k8s). The L1 administrator may then install the eBPFs to implement the network functions explained before.

Fig. 8 presents the dynamic interface creation. In step ❶, the user requests k8s to create a new pod. To create the pod, K8s requests new NIC from the CNI plugin ❷ which internally requests it from L0 via a hypercall interface ❸.

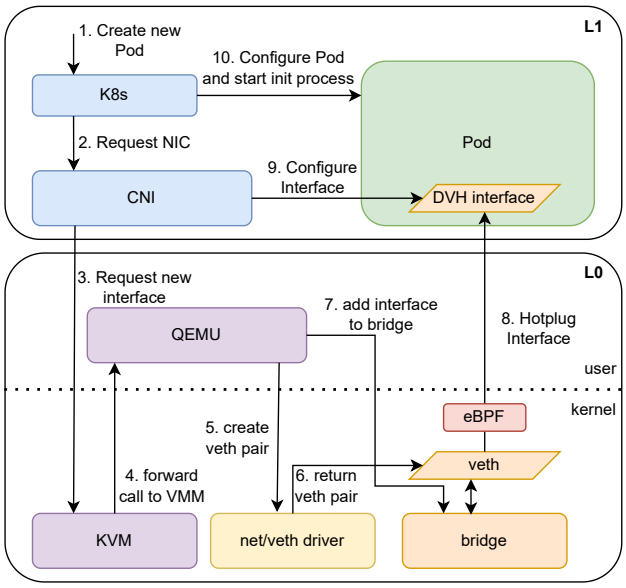


Figure 8: Dynamic virtual interface creation with k8s for spawning a new pod with a DVH network interface. The pod in L1 may be a regular container or a Kata container.

KVM receives the hypercall and forwards it to QEMU ❹, which creates the new vNIC ❺. The L0 kernel creates the interface ❻, such that QEMU attaches it to a predefined bridge ❼, and dynamically hotplugs the interface to L1 via the standard QEMU mechanisms ❽. Meanwhile, the CNI plugin waits for the new interface to appear, and then assigns the correct network namespace and configures the addresses according to k8s' request ❾. Finally, after CNI plugin returns, and k8s starts the pod's init process ❿.

Due to the modular design of the CNI plugin, it also allows the management of VM interfaces and is compatible with other projects without additional implementation efforts.

4.5 Profiling

The hyperupcall to accelerate profiling does not require any information from L1, but it needs to share the samples with the application running in L1. Following the recipe from §4.1, we adopt an approach similar to perf [59] and *share information* via a shared map as a ring buffer which holds the samples. The buffer must be emptied periodically by L1 to allow the profiler to continuously log samples. Additionally, we added a new eBPF helper function, `vcpu_probe` which reads L2's registers, allowing L1 to easily view it's L2 state from within the hyperupcall.

We utilize the preexisting profiling *hook* in Linux to define the hyperupcall, using perf's system call interface to register the hyperupcall's eBPF program as a sampling event. When L1 registers the hyperupcall, is specifies the frequency at which it wants the profiler to run.

After registration, the eBPF program is the periodically called and can sample the guest. Our proof-of-concept profiler *implementation* samples the guest's instruction pointer via new helper function, and logs them onto the ring buffer.

Isolation. We inherit the isolation guarantees from the perf's system call, as it allows us to control the sampler only when the process currently running is in the guest mode.

4.6 Summary of Code Modifications

Tab. 5 highlights the modest code modifications in the respective subsystems required to implement HyperTurtle infrastructure and the hyperupcalls to accelerate the described subsystems. Specifically, the L0 VMM, e.g., QEMU, implements the registration process, requiring the addition of the registration hypercalls. HyperTurtle's EPT fault handling mechanism requires additional eBPF hooks in the L0 kernel that expose the EPT handler routine to the eBPF subsystem. The L0 kernel modifications are limited to exporting additional functions in the memory subsystem and addition of hooks. Lastly, the implementation requires minor additions to the L1 kernel, almost exclusively to the KVM kernel module and the eBPF framework. The implementations of all three hyperupcalls in eBPF are minimal.

In summary, HyperTurtle requires modest changes to the kernel and relatively small amount of eBPF code to achieve its goal of significantly reducing the number of world switches.

4.7 Other usecases

We briefly outline a number of other scenarios where HyperTurtle's approach can be beneficial.

Faster Networking. It is possible to utilize XDP's AF-XDP socket to route traffic directly from the network driver network functions from L2 VMs to L0 to further improve performance.

Storage Devices. Similarly to networking, storage devices may be directly assigned from L0 to L2. By allowing eBPF function that run on L0 to access the file mapping information on L1, files could be read into L2 directly from L0.

SmartNICs. Offloading eBPF programs from L1 to smart NICs can significantly improve performance, thereby completely removing network virtualization logic from the CPU and boosting the performance of nested virtual machines.

Direct Execution and Scheduling. DVH [45] implemented direct execution of nested guests from an idle state of both L1's and L2's vCPU. This can be expanded upon and generalized with hyperupcalls, as L1 may ask L0 to run the L2 vCPU without entering and exiting L1. This is most relevant on over-commited systems running latency sensitive applications such as memcached, as well as when L2 is running co-routines on different vCPUs.

4.8 Limitations

Complex functions, e.g., access to L1's file system, are difficult to implement using eBPFs due to the complexity of synchronizing concurrent access with L1. For the same reason it is unclear how to implement full-offloading without a fallback to L1.

Level	Codebase	Size [LoC]	Description
L0	Kernel/KVM	151	Hyperupcall Infrastructure (see §4.2) and EPT fault hook (see §4.3)
L0	Kernel/eBPF	119	Addition of new helper functions (see §4.5 and §4.2)
L0	QEMU	616	Installer for Hyperupcall programs (see §4.2)
L0	QEMU	159	Dynamic `Direct-Assignment` interface creation (see §4.4.1)
L1	Kernel/mm	32	Support `shmem` backed memory (see §4.3)
L1	Kernel/KVM	653	Exposing data to hyperupcalls (see §4.3)
L0	eBPF	566	EPT fault hyperupcall (see §4.3)
L0	eBPF	263	Networking hyperupcalls (see §4.4)
L0	eBPF	59	Profiler hyperupcall (see §4.5)

Table 5: Code changes necessary to implement HyperTurtle

Our EPT fault hyperupcall implementation does not support allocating huge pages. EPT faults of huge pages are scarce, and therefore accelerating the faults with a hyperupcall has no big performance impact. Additionally, the hyperupcall cannot map pages from L1's page cache unless they are already mapped to L1's VMM page table. Last, it does not yet support multiple EPT-fault hyperupcalls simultaneously. Thus, only one L1 may use the hyperupcall for a single L2. This limitation stems from using a global L0 eBPF hook, but can be eliminated by storing a per-L2 handler in L0's KVM. We leave the implementation for future work.

5 Evaluation

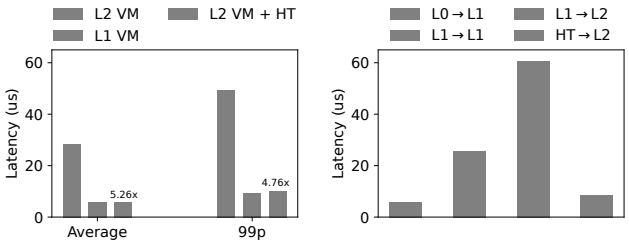
Setup. Tab. 6 describes the setup in all the experiments unless stated otherwise. We pin vCPUs and vhost threads to different physical cores in a single NUMA node, and only one vhost thread is used in each hypervisor. We report average results of ten benchmark iterations. For some benchmarks, we describe the underlying distribution; however, for some microbenchmarks, we only receive statistical aggregates from the tooling and thus cannot report more detailed information. As discussed in §4.2, our modifications in L1 are limited to its kernel, which makes HyperTurtle compatible with different VMMs. As in prior work [32] we use Cloud-Hypervisor [33] as the Kata container's VMM for its high performance and support for pass-through devices. These are not supported by competing VMMs such as Firecracker [1]. Cloud-Hypervisor has better startup times and higher networking performance compared to QEMU. All nested virtualization benchmarks are conducted inside Kata containers [18, 31, 64]. For the network benchmarks, the load generator runs on a bare-metal machine connected directly via a network cable to the system under test. While all presented empirical results use Kata containers, HyperTurtle is fully compatible with classical VMMs on L1 and classical VMs as L2.

5.1 Microbenchmarks

EPT Fault Handling. We evaluate the EPT fault hyperupcall by creating a Kata container, faulting in 1GiB of memory and measuring the latencies of the EPT fault. We compare 3 configurations: (1) L2 VM: a vanilla nested VM, (2) L2 + HyperTurtle: a nested VM with the EPT fault hyperupcall, (3) L1 VM: a non-nested VM – the performance upper bound.

Component	Specification
Bare-Metal Hypervisor (L0)	2×Xeon Silver 4216 (16-core, 2.1GHz); 512GiB RAM; Nvidia MT27710; SMT off;
Nested Hypervisor (L1)	12 vCPUs; 64GiB RAM; QEMU VMM
L1/L2 Baseline (Kata Container)	1 vCPU; 2GiB RAM; Cloud-Hypervisor VMM
Networking	virtio+vhost; tx zero-copy enabled
Boot Parameters	idle=poll; nopti

Table 6: Evaluation Setup



(a) Average and 99p EPT fault latencies. (lower is better).

(b) Latency of a profiler sampling event time in different configurations (lower is better).

Figure 9: EPT Fault and profiling latency microbenchmarks (lower is better). HT ≙ HyperTurtle.

§5.1 compares the average and 99p EPT fault latencies across the listed configurations. HyperTurtle reduces the latency of an EPT fault of the vanilla nested VM by a factor of 5.25× and 4.76× for the average and 99p latency respectively. Furthermore, it has the same average latency of an EPT fault for the non-nested L1 VM, and is only 8% slower in the 99p.

Networking. HyperTurtle enables the use of DVH with the same level of control as `Nested-VirtIO`. However, its design introduces additional computation on each packet due to the execution of the respective eBPF programs in L1's data path.

We measure the UDP round-trip-time latency and TCP throughput, and compare the following configurations: (1) `Nested-VirtIO`: vanilla nested VM, (2) `Direct-Assignment`: best-case performance with DVH but without L1's control, (3) HyperTurtle + Pass: nested VM with HyperTurtle's hyperupcalls mechanism, but with an empty hyperupcall, (4) HyperTurtle + Firewall: nested VM with HyperTurtle with a `firewall` hyperupcall, which allows L1 to enforce a firewall policy on L2's network.. The `pass` and `firewall` hyperupcalls are detailed in §4.

All configurations utilize a single vhost thread in L0, and `Nested-VirtIO` utilizes another vhost thread in L1.

We use Netperf's `UDP_RR` benchmark with 1-byte UDP payload to measure network latency [41]. We measure the maximum achievable throughput with iperf3 [40]. As our goal has been to stress the eBPF code the most, we need 3 vCPUs in the Kata container for the throughput testing to saturate the vhost thread. With a single vhost thread it becomes the bottleneck without achieving line rate. Similar throughput limits have been observed by Firecracker [1].

Tab. 7 compares the average and 99p network latency and throughput. HyperTurtle + Firewall has a 11% higher average latency than `Direct-Assignment` (60.53μs vs 54.5μs) and 33% lower average latency than `Nested-VirtIO` (60.53μs vs. 90.2μs). The eBPF firewall itself causes this overhead, as HyperTurtle + Pass has near identical ($p = 0.5\%$) performance to `Direct-Assignment` (55μs vs. 54.5μs). Thus, HyperTurtle's infrastructure only has a minor impact on network latency, and the major source of overhead is the eBPF program itself.

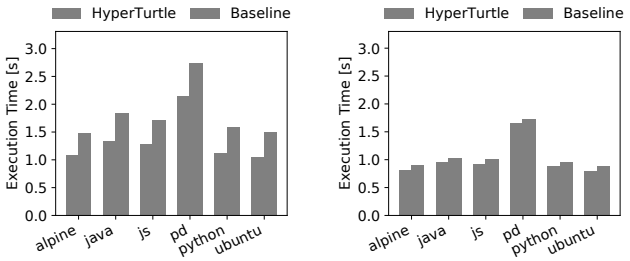
Profiling. We profile GUPS application with a 4MiB working set running in the Kata container on top of L1. We set the profiler sampling frequency to 1000Hz (default in Intel VTune) and compare the following configurations:

1. *L0* → *L1*: Perf profiler profiling a Kata container running in L0. Represents optimal performance.
2. *L1* → *L1*: Perf profiler profiling a container running in L1 from L1.
3. *L1* → *L2*: Perf profiling L2 from L1 (vanilla baseline).
4. *HyperTurtle* → *L2*: HyperTurtle profiler profiling L2.

§5.1 presents the profiling sampling event handle times. Compared to *L1* → *L2*, *HyperTurtle* → *L2* reduces the handling time by 7.15×, and is only 16% slower than the optimal.

5.2 Macrobenchmarks

Kata container startup time. We measure the time it takes a web server invoked in a Kata container to boot and respond to a single request. We evaluate different container images, as these may have a significant effect on the startup time [19], as well as both 4KiB and 2MiB page sizes. Importantly, even when 2MiB pages are used, the Kata runtime in L1 still maps the L2's kernel to its physical address space using 4KiB pages, hence HyperTurtle improves performance by lazily mapping these pages. The page sizes in L1 and L2 are identical, such that an L2 frame corresponds to a single L1 page. We discuss the challenges of huge pages in §4.3.



(a) 4KiB Pages (b) 2MiB Pages

Figure 10: Kata container startup latencies for different applications, with and without HyperTurtle (lower is better). pd ≙ Python with pandas, $CoV < 5\%$.

Fig. 10 compares the startup time of a nested VM with and without HyperTurtle. We observe that HyperTurtle reduces the L2 startup time by 27% and 8% on average when running with 4KiB and 2MiB pages respectively.

Next, we compare the behavior of larger applications - an instance of Redis with a 1GiB persistent snapshot, and the TextService application from the DeathStarBench suite [24]. Fig. 11 shows 55% and 35% performance improvements respectively. Running other micro services from the DeathStarBench resulted in similar performance (not shown).

Nginx. To evaluate web server performance, we deploy Nginx [52] and serve the GCC manuals with 10 concurrent connections using ApacheBench [22], following DVH's methodology [45]. We compare the performance of different hyperupcalls and include `Direct-Assignment` as an upper bound of performance. Fig. 12a shows that HyperTurtle increases throughput by 45% when using the *TCP-Top* and the *rate limiter* hyperupcalls - from 1.24$KQPS$ to 1.8KQS, with near-identical performance to `Direct-Assignment`.

Redis. We evaluate HyperTurtle's impact on Redis's [53] throughput, using memtier_benchmark [54], under its default mixed GET/SET workload. Fig. 12b shows that HyperTurtle increases throughput by up to 65% compared to `Nested-VirtIO`, while all non-`Nested-VirtIO` are within 6% of each other.

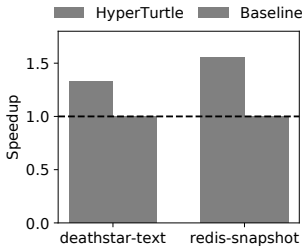
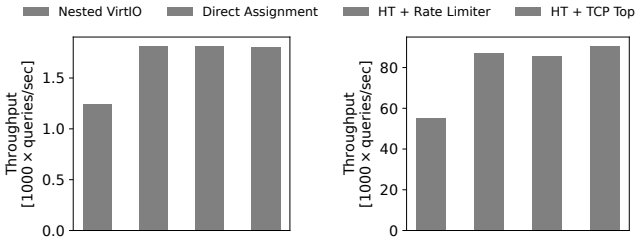


Figure 11: The launch time of a Kata containers running Redis with a 1GiB persistent snapshot, and a TextService microservice from the Deathstar suite (higher is better). $CoV < 5\%$.

Metric	`Nested-VirtIO`	`Direct-Assignment`	HyperTurtle + Pass	HyperTurtle + Firewall
Average Latency [μs] ↓	90.2	54.52 (×1.6)	55.0 (×1.6)	60.5 (×1.5)
99p Latency [μs] ↓	106.0	67.0 (×1.6)	67.0 (×1.6)	73.0 (×1.4)
Throughput [Gb/s] ↑	12.1	17.4 (×1.4)	17.3 (×1.4)	17.1 (×1.4)

Table 7: Network latency and throughput. Values in parentheses are improvements relative to `Nested-VirtIO`. The CoV is below 10% in all the experiments.



(a) Nginx (b) Redis

Figure 12: Evaluation of throughput in NGINX and Redis. Other Hyperupcalls show similar performance and are thus omitted.

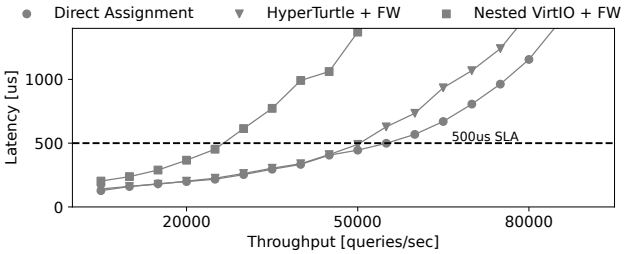
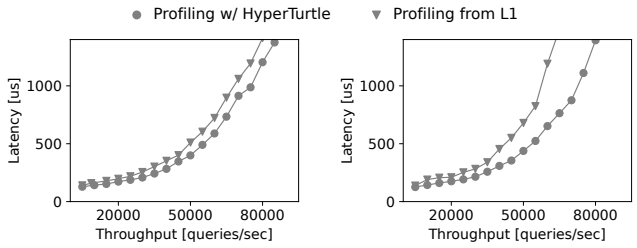


Figure 13: Memcached throughput-99p latency.

Memcached. We use Memcached to show HyperTurtle's effect on latency-sensitive applications. We use mutilate [43] as a load generator and Facebook's ETC load [6] as query distribution. We examine HyperTurtle with a Firewall eBPF, and then compare the effects of profiling Memcached from L0 and L1. All reported results are the average of 10 runs, and have a standard deviation less than 0.1. Fig. 13 compares the $99^{th}p$ latency in throughput-latency plot following the methodology of the IX project [10].

HyperTurtle provides consistent throughput improvements, notably at the 500μs SLA from 29$KQPS$ to 50$KQPS$. All while supporting L1-controlled network functions. HyperTurtle's throughput is only 9% lower than `Direct-Assignment` at 500μs (50$KQPS$ vs 55$KQPS$), which we attribute to the eBPF firewall application itself. We conclude that HyperTurtle enables the usage of L1-controlled eBPF network functions, such as firewalls, with performance on par with DVH.

We use the same setup to evaluate the HyperTurtle profiler, as the tail latency of Memcached is sensitive to sampling-based profilers. Fig. 14 compares Memcached performance when profiling at frequencies of 1000Hz (Intel Vtune profiler's recommendation [38]) and 4000Hz (Linux perf's default [59]). At a rate of 4000Hz, HyperTurtle profiler achieves 26% higher throughput (42$KQPS$ to 53$KQPS$) at 500μs query latency. Decreasing the sampling rate to 1000Hz reduces the throughput gain to 12% (49$KQPS$ to 55$KQPS$). We conclude that HyperTurtle is effective in reducing the overhead of profiling of applications running in nested virtual machines.



(a) 1000Hz (b) 4000Hz

Figure 14: 99p latencies of Memcached get requests during profiling with 1KHz and 4KHz sampling rates. Both setup use `Direct-Assignment` networking.

6 Related Work

Hyperupcalls [4] suggested the idea of safely executing L1 code in L0. It was suggested as a mechanism which facilitates complex communication between the L1 and L0, which increased performance and flexibility.

Nested virtualization. The overheads of nested virtualization are a well-known problem, exposed since the first implementation by the Turtles project [11].

There have been several efforts to reduce the number of world switches to improve the nested virtualization performance. NEVE [44] proposed a series of architectural changes to the ARM architecture, aimed at reducing the overhead of nested virtualization by reducing the number of `vm-exits`. Intel introduced VMCS shadowing [35] with their Haswell lineup of processors, allowing L0 to not trap L1 on every execution of `vm-write` and `vm-read` instructions.

DVH [45] is a collection of optimizations, including direct assignment of L0's virtual I/O devices and timers to L2. This effectively removes L1 from the control and data path. However, it limits deployment scenarios because L1 can no longer enforce policies on L2 I/O operations.

SVT [60] suggested utilizing SMT to run L0, L1 and L2 each in a different hardware thread. This approach allows for fast world-switching, which improves performance.

However, it does not reduce the increased handle times which are present in nested virtualization. Moreover, SVT requires CPU support for simultaneous multi-threading, which is often missing [2, 16], and will likely become unavailable in future CPUs [36, 37]

Virtualization-based containers. Kata containers [42] is a container runtime which utilizes microVMs for improved sandboxing. This allows the creation, destruction, and overall management of micro-VMs based on standard container images. By complying with the Open Container Initiative (OCI) interface, it achieves compatibility with Kubernetes.

Multiple works have suggested software virtualization as a cloud-native container sandboxing mechanism. LightVM [48] and X-Containers [57] propose to use Xen-based virtualization. While LightVM relies on Unikernels for container images, X-Containers uses a Linux-based Library OS. PVM [31] presents a hypervisor for software-virtualization, specially built for nested virtualization. It utilizes shadow paging and runs a para-virtual Linux guest These approaches, however, suffer from the known shortcomings of software virtualization, which has declined in popularity [9]. X-Containers [57] require the nested guest to use a specialized library OS, and PVM [31] utilizes syscall redirection and shadow paging for the nested guest, which lead to poor performance in syscall-intensive applications such as Redis and SQLite [58].

CKI [58] suggested an addition to the Protection Keys for Supervisor hardware extension, to create an isolated environment within the L1 kernel. With this approach, L2's kernel runs in a sandbox within L1's kernel, improving performance compared to traditional software-based solutions. This approach requires hardware modifications, so it cannot be used with existing systems. gVisor [29] takes a different approach to container isolation. It intercepts and handles system calls in a separate userspace process, which provides stronger isolation between containrs at the cost of higher overheads compared to traditional container runtimes.

HyperTurtle can be used to accelerate these approaches as well, as they all suffer from increased overheads from the multiple virtualization layers present in the cloud.

Avoiding nested virtualization. Some systems aim to avoid nested virtualization entirely. Peer-Pods [5] leverage cloud APIs to deploy Kata containers without nesting, while Free-The-Turtles [63] and Hyper-V [51] expose guest-level VM management interfaces. These approaches, however, give up hypervisor-level resource control, such as memory overcommitment across L2s within a single L1 VM — reducing deployment flexibility. Dichotomy [62] sidesteps nesting overhead by rapidly migrating the L2 VM between a minimal host hypervisor and a full-featured user hypervisor using memory remapping. However, it is incompatible with scenarios where L1 must remain continuously active, such as those involving Kubernetes orchestration. Out-of-Hypervisor [12] proposes exposing hardware virtualization features directly to the guest OS to support advanced tasks like checkpointing.

HyperTurtle offers the convenience of full compatibility with existing deployments and tooling, while enabling significant performance gains.

eBPF. There exists a variety of eBPF applications, especially in the form of network functions. HyperTurtle enables the compatibility of DVH with many eBPF network functions: BMC [26] implements eBPF Memcached read-cache. Cilium [14] provides powerful observability for Kubernetes clusters as eBPF programs. HyperTurtle provides the compatibility between `Direct-Assignment` and Kubernetes network introspection through Cillium. XDP [39] present an eBPF hook in the network device driver, enabling packet processing early in the kernel's network stack. Similarly, XRP [65] used eBPF to offload storage lookups from user space to the kernel, reducing the overhead of the kernel storage stack.

7 Conclusion

HyperTurtle proposes a general approach to accelerating nested virtualization by reducing the number of world switches. The idea is to use hyperupcalls, small eBPF programs, that encapsulate the relevant L1 hypervisor functionality and are invoked by L0 without transitioning to L1 when handing L2's `vm-exit`. We demonstrate the effectiveness of HyperTurtle to reduce the overheads and the startup time of applications in nested Kata conatiners. We show substantial performance gains enabled by relatively modest code changes, and believe that HyperTurtle is a practical solution that can be further effective in many nested virtualization scenarios.

8 Acknowledgments

We thank our shepherd Stella Bitchebe and the reviewers for their helpful comments and feedback. This work was partially supported by IBM – Technion Research Collaboration. We gratefully acknowledge the generous support of the Israeli Science Foundation (Grant 1998/22).

References

[1] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. Firecracker: Lightweight virtualization for serverless applications. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 419–434, Santa Clara, CA, February 2020. USENIX Association.

[2] Amazon. AWS Graviton Performance Testing: Tips for Independent Software Vendors. https://docs.aws.amazon.com/pdfs/whitepapers/latest/aws-graviton-performance-testing/aws-graviton-performance-testing.pdf#what-is-aws-graviton, 2024. Online; Last accessed 14-Jan-2025.

[3] Amazon. What is a Private Cloud? https://aws.amazon.com/what-is/private-cloud/, 2024. Online; Last accessed 13-Jan-2025.

[4] Nadav Amit and Michael Wei. The Design and Implementation of Hyperupcalls. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 97–112, Boston, MA, July 2018. USENIX Association.

[5] Pradipta Banerjee Ariel Adam. Red hat openshift sandboxed containers: Peer-pods solution overview. https://www.redhat.com/en/blog/red-hat-openshift-sandboxed-containers-peer-pods-solution-overview, 2023. Online; Last accessed 12-May-2025.

[6] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '12, page 53–64, New York, NY, USA, 2012. ACM.

[7] CNI Authors. Cni. https://www.cni.dev/. Online; Last accessed 2024-May-06.

[8] AWS. Amazon CodeGuru Security. https://aws.amazon.com/codeguru/, 2024. Online; Last accessed 2024-Dec-21.

[9] Jeff Barr. Now Available – Compute-Intensive C5 Instances for Amazon EC2. https://aws.amazon.com/blogs/aws/now-available-compute-intensive-c5-instances-for-amazon-ec2/, 2025. Online; Last accessed 14-Jan-2025.

[10] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 49–65, Broomfield, CO, October 2014. USENIX Association.

[11] Muli Ben-Yehuda, Michael D. Day, Zvi Dubitzky, Michael Factor, Nadav Har'El, Abel Gordon, Anthony Liguori, Orit Wasserman, and Ben-Ami Yassour. The Turtles Project: Design and Implementation of Nested Virtualization. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, page 423–436, USA, 2010. USENIX Association.

[12] Stella Bitchebe and Alain Tchana. Out of hypervisor (ooh): efficient dirty page tracking in userspace using hardware virtualization features. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '22. IEEE Press, 2022.

[13] Blaise Boscaccy. Introducing hornet lsm. https://lore.kernel.org/lkml/20250321164537.16719-1-bboscaccy@linux.microsoft.com/, 2025. Online; Last accessed 11-May-2025.

[14] Cilium. eBPF-based Networking, Observability, Security. https://cilium.io/. Online; Last accessed 2024-Feb-05.

[15] cilium. Bpf and xdp reference guide. https://docs.cilium.io/en/latest/reference-guides/bpf/index.html, 2026. Online; Last accessed 2025-May-11.

[16] Ampere Computing. Designed For The Modern Cloud. https://amperecomputing.com/products/processors, 2025. Online; Last accessed 14-Jan-2025.

[17] containernetworking. plugins. https://github.com/containernetworking/plugins/tree/main, 2025. Online; Last accessed 14-Jan-2025.

[18] Docker. Alternative container runtimes. https://docs.docker.com/engine/daemon/alternative-runtimes/, 2024. Online; Last accessed 2024-Dec-31.

[19] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. Catalyzer: Sub-Millisecond Startup for Serverless Computing with Initialization-Less Booting. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 467–481, New York, NY, USA, 2020. ACM.

[20] Elastic. Universal profiling. https://www.elastic.co/observability/universal-profiling, 2024. Online; Last accessed 2024-Dec-21.

[21] Arthur Fabre. L4Drop: XDP DDoS Mitigations. https://blog.cloudflare.com/l4drop-xdp-ebpf-based-ddos-mitigations, 2018. Online; Last accessed 2024-Apr-07.

[22] The Apache Software Foundation. ab - apache http server benchmarking tool. https://httpd.apache.org/docs/2.4/programs/ab.html, 2015. Online; Last accessed 02-June-2025.

[23] The Linux Foundation. Kubevirt. https://kubevirt.io, 2024.

[24] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna

Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, and Christina Delimitrou. An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 3–18, New York, NY, USA, 2019. ACM.

[25] Elazar Gershuni, Nadav Amit, Arie Gurfinkel, Nina Narodytska, Jorge A. Navas, Noam Rinetzky, Leonid Ryzhyk, and Mooly Sagiv. Simple and precise static analysis of untrusted linux kernel extensions. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, page 1069–1084, New York, NY, USA, 2019. ACM.

[26] Yoann Ghigoff, Julien Sopena, Kahina Lazri, Antoine Blin, and Gilles Muller. BMC: Accelerating Memcached using Safe In-kernel Caching and Pre-stack Processing. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 487–501. USENIX Association, April 2021.

[27] Google. Cloud Profiler documentation. https://cloud.google.com/profiler/docs, 2024. Online; Last accessed 2024-Dec-21.

[28] Google. Profile production applications in Azure with Application Insights Profiler for .NET. https://learn.microsoft.com/en-us/azure/azure-monitor/profiler/profiler-overview, 2024. Online; Last accessed 2024-Dec-21.

[29] Google. gvisor. https://github.com/google/gvisor, 2025. Online; Last accessed 14-Jan-2025.

[30] Brendan Gregg. Bpf performance analysis at netflix. https://rnx.gobailia.uk/Slides/reInvent2019_BPF_Performance_Analysis.pdf, 2019.

[31] Hang Huang, Jiangshan Lai, Jia Rao, Hui Lu, Wenlong Hou, Hang Su, Quan Xu, Jiang Zhong, Jiahao Zeng, Xu Wang, Zhengyu He, Weidong Han, Jiang Liu, Tao Ma, and Song Wu. PVM: Efficient Shadow Paging for Deploying Secure Containers in Cloud-native Environment. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 515–530, New York, NY, USA, 2023. ACM.

[32] Hang Huang, Jiangshan Lai, Jia Rao, Hui Lu, Wenlong Hou, Hang Su, Quan Xu, Jiang Zhong, Jiahao Zeng, Xu Wang, Zhengyu He, Weidong Han, Jiang Liu, Tao Ma, and Song Wu. PVM Source Code. https://github.com/virt-pvm/misc, 2024. Online; Last accessed 13-Jan-2025.

[33] Cloud Hypervisor. Cloud Hypervisor. https://www.cloudhypervisor.org/, 2023. Online; Last accessed 30-Nov-2024.

[34] inspektor gadget. inspektor-gadget. https://github.com/inspektor-gadget/inspektor-gadget. Online; Last accessed 2024-Apr-16.

[35] Intel. 4th Generation Intel® Core™ vPro™ Processors with Intel® VMCS Shadowing. https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/intel-vmcs-shadowing-paper.pdf, 2013. Online; Last accessed 2023-Jan-23.

[36] Intel. Architecture All Access: Live at Lunar Lake ITT: Next Gen P-core Lion Cove. https://www.youtube.com/watch?v=FcBRRiQuzNU, 2024. Online; Last accessed 14-Jan-2025.

[37] Intel. Intel® Core™ Ultra 9 Processor 285K. https://www.intel.com/content/www/us/en/products/sku/241060/intel-core-ultra-9-processor-285k-36m-cache-up-to-5-70-ghz/specifications.html, 2024. Online; Last accessed 14-Jan-2025.

[38] Intel. Intel® vtune™ profiler user guide. https://www.intel.com/content/www/us/en/docs/vtune-profiler/user-guide/, 2024. Online; Last accessed 30-Nov-2024.

[39] iovisor. xdp. https://www.iovisor.org/technology/xdp, 2016. Online; Last accessed 2024-Dec-31.

[40] iperf3. iperf3. https://iperf.fr, 2024. Online; Last accessed 2025-Jan-05.

[41] Rick Jones. netperf. https://github.com/HewlettPackard/netperf. Online; Last accessed 2024-Apr-19.

[42] Kata Containers. https://github.com/kata-containers/kata-containers. Online; Last accessed 2024-Apr-07.

[43] Jacob Leverich. Mutilate: High-Performance Memcached Load Generator. https://github.com/leverich/mutilate. Online; Last accessed 2024-Feb-05.

[44] Jin Tack Lim, Christoffer Dall, Shih-Wei Li, Jason Nieh, and Marc Zyngier. NEVE: Nested Virtualization Extensions for ARM. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 201–217, New York, NY, USA, 2017. ACM.

[45] Jin Tack Lim and Jason Nieh. Optimizing Nested Virtualization Performance Using Direct Virtual Hardware. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 557–574, New York, NY, USA, 2020. ACM.

[46] Linux. bpf-helpers(7) — linux manual page. https://man7.org/linux/man-pages/man7/bpf-helpers.7.html. Online; Last accessed 2024-May-13.

[47] Linux. capabilities(7) — linux manual page. https://man7.org/linux/man-pages/man7/capabilities.7.html. Online; Last accessed 2024-May-13.

[48] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. My VM is Lighter (and Safer) than your Container. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 218–233, New York, NY, USA, 2017. ACM.

[49] Meta. Katran. https://github.com/facebookincubator/katran. Online; Last accessed 2024-Feb-05.

[50] Microsoft. Preview support for kata vm isolated containers on aks for pod sandboxing. https://techcommunity.microsoft.com/t5/apps-on-azure-blog/preview-support-for-kata-vm-isolated-containers-on-aks-for-pod/ba-p/3751557. Online; Last accessed 2024-Feb-05.

[51] Microsoft. Windows hypervisor platform api definitions. https://learn.microsoft.com/en-us/virtualization/api/hypervisor-platform/hypervisor-platform, 2025. Online; Last accessed 13-May-2025.

[52] nginx. https://nginx.org/. Online; Last accessed 2025-May-14.

[53] Redis - the real-time data platform. https://redis.io/. Online; Last accessed 2025-May-14.

[54] RedisLabs. memtier_benchmark. https://github.com/RedisLabs/memtier_benchmark. Online; Last accessed 2025-May-14.

[55] Gang Ren, Eric Tune, Tipp Moseley, Yixin Shi, Silvius Rus, and Robert Hundt. Google-Wide Profiling: A Continuous Profiling Infrastructure for Data Centers. *IEEE Micro*, 30(4):65–79, 2010.

[56] Rusty Russell. virtio: towards a de-facto standard for virtual i/o devices. *SIGOPS Oper. Syst. Rev.*, 42(5):95–103, jul 2008.

[57] Zhiming Shen, Zhen Sun, Gur-Eyal Sela, Eugene Bagdasaryan, Christina Delimitrou, Robbert Van Renesse, and Hakim Weatherspoon. X-Containers: Breaking Down Barriers to Improve Performance and Isolation of Cloud-Native Containers. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 121–135, New York, NY, USA, 2019. ACM.

[58] Jiacheng Shi, Yang Yu, Jinyu Gu, and Yubin Xia. A hardware-software co-design for efficient secure containers. In *Proceedings of the Twentieth European Conference on Computer Systems*, EuroSys '25, page 1229–1245, New York, NY, USA, 2025. ACM.

[59] Linus Torvalds. Linux. https://github.com/torvalds/linux/tree/master. Online; Last accessed 2024-Nov-30.

[60] Lluís Vilanova, Nadav Amit, and Yoav Etsion. Using smt to accelerate nested virtualization. In *Proceedings of the 46th International Symposium on Computer Architecture*, pages 750–761, 2019.

[61] virtiofsd project. virtiofsd. https://virtio-fs.gitlab.io/. Online; Last accessed 2024-May-09.

[62] Dan Williams, Yaohui Hu, Umesh Deshpande, Piush K. Sinha, Nilton Bila, Kartik Gopalan, and Hani Jamjoom. Enabling efficient hypervisor-as-a-service clouds with ephemeral virtualization. In *Proceedings of The12th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '16, page 79–92, New York, NY, USA, 2016. ACM.

[63] Mengmei Ye, Angelo Ruocco, Daniele Buono, James Bottomley, and Hubertus Franke. Free the turtles: Removing nested virtualization for performance and confidentiality in the cloud. In *2023 IEEE 16th International Conference on Cloud Computing (CLOUD)*, pages 275–281, 2023.

[64] Zhang Yu. The Application of Kata Containers in Baidu AI Cloud. Technical report, Baidu, 2019. Online; Last accessed 08-April-2023.

[65] Yuhong Zhong, Haoyu Li, Yu Jian Wu, Ioannis Zarkadas, Jeffrey Tao, Evan Mesterhazy, Michael Makris, Junfeng Yang, Amy Tai, Ryan Stutsman, and Asaf Cidon. XRP: In-Kernel Storage Functions with eBPF. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 375–393, Carlsbad, CA, July 2022. USENIX Association.

Efficient Performance-Aware GPU Sharing with Compatibility and Isolation through Kernel Space Interception

Shulai Zhang[1,*], Ao Xu[1,*], Quan Chen[1], Han Zhao[1], Weihao Cui[1], Zhen Wang[2], Yan Li[2], Limin Xiao[2], Minyi Guo[1]
[1]*Shanghai Jiao Tong University* [2]*Lenovo*

Abstract

To support diverse GPU applications and ensure their performance, it is crucial to ensure compatibility, isolation, and maximizing utilization. However, existing approaches are limited to CUDA runtimes and have vulnerable isolation, where the misbehavior or crash of a single application disrupts all other applications sharing the same GPU. Moreover, existing solutions fail to efficiently orchestrate the applications.

Our investigation reveals that the limitations in compatibility and isolation stem from the user-space design of existing GPU-sharing solutions. To address these issues, we propose **KRYPTON**, a kernel-space GPU-sharing scheme that ensures compatibility and isolation. KRYPTON intercepts GPU command buffers at the kernel level to provide virtual GPU devices. Rather than relying on fixed GPU resource allocation, it employs efficient spatio-temporal sharing, enabling performance guarantees while improving resource utilization. Experimental results show that KRYPTON reduces the required GPU number by 32.1% compared with SOTA baselines, while providing robust compatibility and isolation.

1 Introduction

GPUs are widely used for many workloads, like AI services [10, 15], scientific computing [38, 59], video rendering [27, 36], et. al. Since many such tasks are lightweight and require only a fraction of a GPU to achieve their performance targets (e.g., throughput, end-to-end latency) [22, 35, 37, 44, 45, 54, 63, 80], co-running them on the same GPU greatly improves the resource efficiency [19, 23]. For instance, Amazon has commenced delivering video content utilizing fractional GPUs [22] and AI workloads also operate with fractional GPUs [39, 72]. GPU jobs may rely on different runtimes, although they share the host operating system (OS) and drivers. For instance, AI services and scientific computing often rely on CUDA runtimes [20], video and game rendering often rely on graphic runtimes such as Vulkan [29] and OpenGL [14].

*Equal contribution.

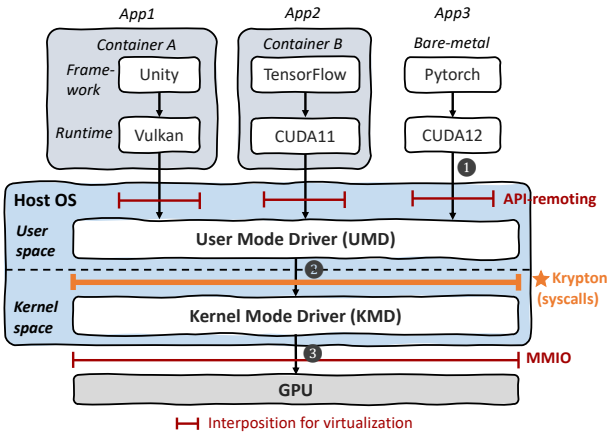


Figure 1: The system stack of virtualizing and sharing a GPU.

An efficient GPU sharing scheme should be capable of co-locating jobs with different runtimes (**compatibility**), while providing performance and fault isolation (**isolation**) for the co-located jobs. Fault isolation means that a job's crash would not destroy other co-located jobs. Performance isolation includes satisfying the isolated compute and memory requirements of tenants. To virtualize/share a GPU, the general principle is to intercept and interpose an interface, enabling the modification or extension of its behavior. Figure 1 shows the system stack of virtualizing a GPU. As observed, GPU vendors leverage the User Mode Driver (UMD) in user space to receive API calls (❶). The UMD communicates with the kernel mode driver (KMD) in kernel space via system calls, such as `ioctl` and `write` (❷). The KMD, in turn, interacts with the GPU hardware through memory-mapped I/O (MMIO) (❸).

There are already many prior works on co-locating the jobs that rely on the same runtimes on a GPU. They often intercept API calls above UMD [43, 66, 72, 76], or trap MMIO below KMD [65, 67, 68], providing poor compatibility. Intercepting user-space API calls and remote them, referred to as API-

remoting, requires adapting a large number of API interfaces for different runtimes and a large amount of manpower for maintenance, while trapping and simulating frequent MMIO access undermines performance (e.g., 100× or more [67,77]).

In terms of performance isolation, TGS [72], Orion [66], and GaiaGPU [43] adopt time-sharing, while GPUlet [35] and GSlice [39] adopt spatial-sharing based on Nvidia's Multi-Process Service (MPS) [2]. With time-sharing, GPU time slices may interfere due to the non-programmable nature of GPU context switching. With spatial sharing, co-located jobs share the GPU contexts and global memory bandwidth, suffering from poor fault isolation or performance guarantees. A job's crash would break entire GPU context, destroying all the co-located jobs. Nvidia's Multi-Instance GPU (MIG) [7] achieves hardware-level isolation by partitioning the GPU spatially. However, it sacrifices flexibility in resource management due to pre-defined resource splits.

Two key insights motivate this work. First, to ensure compatibility and low overhead, interposition and scheduling should be shifted to kernel space in host OS. As illustrated in Figure 1, kernel-space interposition eliminates the need for adapting to diverse runtimes and avoids frequent MMIO trapping, while enabling comprehensive control over GPU access. Second, instead of weighing between assigning fixed and unshared resources and resolving interference in shared resources, there is more room to directly guarantee application performance targets (such as throughput or latency). Avoiding rigid resource quotas allows for more flexible resource orchestration, reducing GPU fragmentation and improving overall resource utilization.

It is non-trivial to achieve the above design. Intercepting and remoting all functions passed from UMD to KMD in kernel space is infeasible, as these interface layers are proprietary and not disclosed by vendors. It is challenging to find a universal mechanism to capture all requests to GPU without analyzing and simulating the unveiled functions. In terms of orchestrating resources, allowing time and spatial sharing simultaneously is beneficial to reduce the GPU fragments, as it expands the orchestration space. However, with the large space, it is challenging to identify the appropriate allocation that ensures the performance target of all the applications, while minimizing the GPU fragments.

We therefore propose **KRYPTON**, a sharing scheme that comprises *an offline profiler,* a *kernel-space interception module*, a *performance feedback controller*, and a *spatio-temporal orchestrator*. The offline profiler can obtain the performance of each application with different configurations of resource provisioned, aiming for better resource allocation (§3.2). The interception module hijacks the access control to the GPU of the operating system instead of intercepting individual API calls (§3.3). For each of the co-running applications, the feedback controller (§3.4) monitors its real-time performance with the allocated hardware units and time slices. It adjusts resources adaptively to guarantee performance. We call the given hardware units, time slices, and memory allocated to an application to be an "IGPU" (isolated GPU). The hardware units are allocated with MIG that can spatially divide an A100 GPU into seven (or fewer) instances, while the time slices and memory are allocated with the kernel-space interception module. The spatio-temporal orchestrator proposes a two-stage bin-packing algorithm that orchestrates IGPUs to improve the overall utilization while promising the performance (§3.5).

We have implemented KRYPTON and evaluated it on Nvidia A100 and RTX4090 GPUs. Its effectiveness is tested with multiple versions of CUDA and Vulkan runtimes. Experimental results show that KRYPTON reduces the required GPU number by 32.1%, compared with SOTA GPU sharing systems, while maintaining the target throughput. The relative error of applications' performance is merely 3.3% on average. The main contributions are three-fold:

1) The design of kernel-space interception that ensures compatibility and isolation to use GPU. The interposition supports various runtimes and drivers.

2) The design of a feedback-based performance control policy. With adaptive time slice allocation in real-time, the performance of applications can be guaranteed.

3) The design of a spatio-temporal orchestration policy that efficiently reduces GPU fragments. KRYPTON cuts down the number of GPUs while ensuring the performance targets of the co-located applications.

2 Investigating GPU Sharing

In this section, we introduce the prevalent techniques in supporting existing GPU-sharing solutions and investigate their weaknesses, thereby motivating KRYPTON.

2.1 Techniques beneath GPU-sharing

Existing GPU sharing solutions require the GPU resource management capability as fundamental for the ability to "share". They leverage either time sharing supported by API-remoting [43, 64, 66, 72], spatial sharing supported by MPS [39, 42, 81] or MIG [53], or a combination of them [35].

All these techniques provide process-level resource management. API-remoting-based methods intercept GPU function calls of tenants to let them run in a temporal round-robin manner. MPS [2] merges the CUDA contexts of different processes into a unified context. Then multiple processes can run on the GPU simultaneously with the compute resource controlled. MIG [7] is a hardware-isolated solution for Nvidia computing GPUs. MIG divides a GPU into fully isolated instances with separate compute and memory resources but it only supports GPU partitioning with fixed configurations.

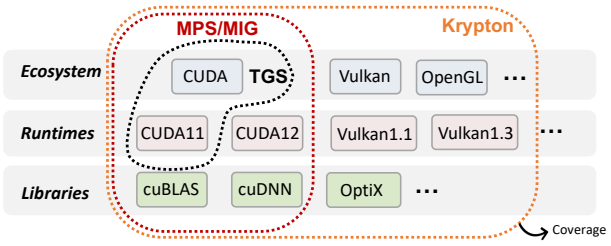


Figure 2: Software coverage of GPU sharing methods.

2.2 Poor Compatibility and Isolation

2.2.1 Limited Compatibility

Compatibility refers to the capability of the sharing mechanism to cover as many software scenarios as possible without much effort for adaptation. As depicted in Figure 2, the software functionalities an Nvidia GPU can support can be categorized into various ecosystems, runtimes and libraries.

API-remoting methods [43,55,64,66,72,76] do not promise robustness to the diversity and evolution of interposed interfaces, because they can only hijack explicit APIs of user-space runtimes. Thus, achieving compatibility in API-remoting methods is costly: Various ecosystems such as CUDA, Vulkan, and OpenGL offer fundamentally different interface APIs. Besides, user-space runtimes often undergo frequent updates, and this is particularly evident in the CUDA ecosystem, where Nvidia has released more than 70 versions of the CUDA runtime so far [21]. With each new version, API changes can introduce significant variations, requiring continuous adaptation of the API-remoting mechanism to maintain compatibility.

Beyond the core CUDA runtime, there are numerous dynamic libraries within the CUDA ecosystem. Since API-remoting methods rely on intercepting APIs, they must be updated frequently to accommodate changes in these libraries' interfaces. Because of the large effort to maintain compatibility, many API-remoting sharing systems only cover a subset of the software. For example, TGS [72], as an API-remoting sharing system, only supports CUDA11.2 runtime.

MPS-based methods, although have no requirements to adapt for different libraries, are also limited for CUDA applications and cannot be used for graphic runtimes (e.g. OpenGL, Vulkan) [2]. MIG as another vendor solution, although can provide fully isolated compute and memory system for each tenant, also targets CUDA applications and lacks key features on A100 machines(e.g., support for graphic APIs, device P2P communication) [5]. Thus, limited in poor compatibility, existing GPU sharing platforms only target applications using CUDA (e.g., deep learning workloads).

2.2.2 Weak Isolation under Threat

We design a threat model to reveal the poor isolation using existing GPU-sharing methods. In our threat model, we assume an adversary is a program that has legitimate access to a GPU shared with other tenants. The program can initialize an arbitrary number of processes to access the GPU. To examine the isolation of existing solutions, we run all 8 benchmarks based on CUDA runtime in Table 2 respectively as the victims.

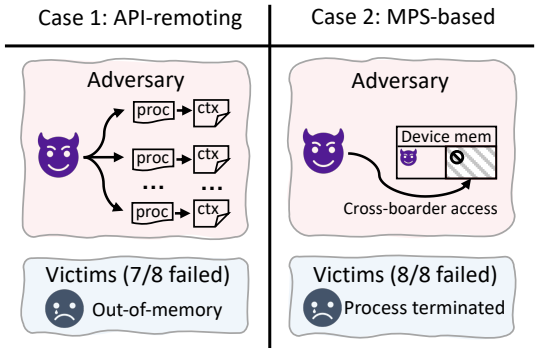


Figure 3: The adversary fails the victims with different sharing mechanisms.

In TGS [72], a GPU sharing system sits on API-remoting, victims suffer out-of-memory errors in 7 out of 8 cases and the adversary simply spawns 48 processes that use the GPU through CUDA runtime (Figure 3(a)). This is because user-space API-remoting cannot hijack implicit memory-related commands (e.g., the CUDA context creation implicitly allocates about 400MB of GPU memory), potential memory contentions occur and break isolation among applications. The processes in the adversary program allocate device memory implicitly for the extra GPU contexts, which encroach the provisioned memory of the victims.

In GPU-sharing systems based on MPS (e.g., GSlice [39]), if the adversary performs a cross-boarder GPU memory access, then all 8 victims crash due to the fatal GPU memory error (Figure 3(b)). Such error propagation problem has also been reported in prior researches [73, 81]. The key reason for the error propagation is that MPS funnels kernels from separate processes and runs them in the same context.

The experimental results show that API-remoting and MPS-based methods do not promise isolation, because API-remoting methods ignore implicit GPU operations and co-running applications share the same GPU context with MPS. MIG-based methods can promise isolation, benefiting from the stringent hardware-level isolation.

2.2.3 Potential Solution

To achieve compatibility and isolation, we investigate the workflow of using a GPU as shown in Figure 4. The workflow is analyzed from the codes open sourced by Nvidia [12]. Prior researches [17, 18, 50, 51, 67] have similar observations.

Specifically, applications use runtime APIs (e.g., CUDA, Vulkan) to generate tasks (kernels, data transfers, etc). The

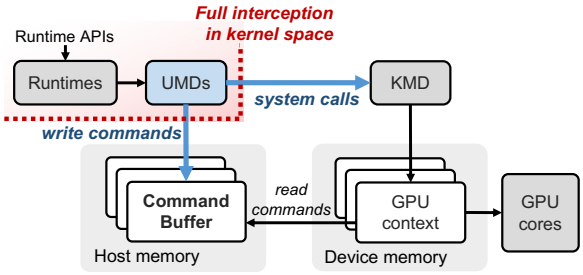


Figure 4: The workflow of using a GPU.

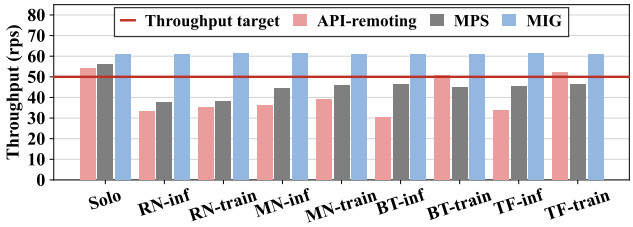


Figure 5: The practical throughput of a ResNet50 inference application with a 50 rps throughput target when co-located with another application using different GPU sharing mechanisms. The 'Solo' pertains to the scenario where the ResNet50 inference application runs stand-alone.

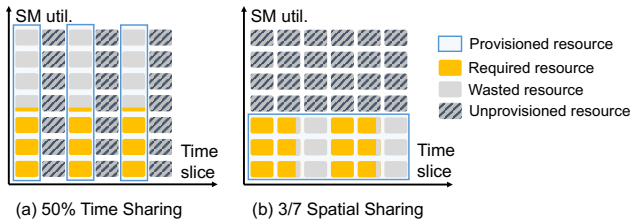


Figure 6: Resource utilization in time and spatial sharing when the *ResNet50*'s throughput target is achieved.

runtime APIs are parsed and forwarded to *User Mode Drivers (UMD)* and UMDs write commands to *Command Buffers*. The commands cover the control of the GPU, including commands to copy data from the host to the device memory, launch kernels, and so forth. The UMD then uses system calls to guide the *Kernel Mode Driver (KMD)* to inform the GPU to read commands from command buffers. Typically, a command buffer is bonded to a GPU context that encapsulates all the resources needed to execute operations on the GPU, including distinct address space, memory allocations, etc. The GPU contexts are isolated, as they do not share address space. We therefore get the first insight.

Insight-1: Kernel space interception is capable of enabling compatibility and isolation. Based on the observation above, we confirm that it is a good option to intercept command buffers and system calls in kernel space to take full control of the entire GPU. The compatibility across runtimes is then guaranteed naturally because all commands pass the command buffers. To promise isolation among applications, all processes run within their isolated GPU contexts and all memory-related operations can be precisely recorded and controlled in kernel space.

2.3 Inefficiencies in Achieving Performance Target and High Utilization

Even when co-running applications without accounting for compatibility and isolation, existing methods still fail to guarantee the performance of co-located applications and the efficient utilization of GPUs. To demonstrate these shortcomings, we conduct two experiments to present these inefficiencies.

2.3.1 Achieving Performance Target

In this experiment, we use a *ResNet50* inference application as a tenant on an A100 GPU to illustrate the performance violations caused by existing methods. First, we profile the resource quotas (e.g., time slices or SMs) required by the ResNet50 inference application to meet its performance target of 50 requests per second. For instance, ResNet50 requires 32% of the time slices to achieve this target. Next, we co-run the ResNet50 with other benchmarks listed in Table 2 using current methods. The co-running benchmarks use the remaining resource quotas.

Figure 5 illustrates the performance of ResNet50 when API-remoting, MPS, and MIG techniques are employed to share the GPU with other benchmarks. In the figure, the x-axis represents the benchmarks co-running with ResNet50. As observed, the throughput of ResNet50 drops to 30.6 RPS with API-remoting and 37.5 RPS with MPS. This decline is attributed to shared resource contention, such as global memory bandwidth. Consequently, API-remoting and MPS fail to guarantee the performance of co-located applications.

2.3.2 Achieving High Utilization

Figure 6 presents the GPU resource utilization when the performance target for *ResNet50* is set to 80 RPS, and the required resources are allocated using temporal-sharing (API-remoting) and spatial-sharing (MIG).

Specifically, while ResNet50 requires 50% of the temporal quota to achieve the target performance of 80 RPS using the API-remoting method, the average SM utilization is only 44%. Similarly, when ResNet50 uses a 3/7 fractional GPU instance under the MIG method, only 62% of the GPU instance's time slices are needed to achieve the desired throughput. This demonstrates that with the time-sharing method, applications cannot fully utilize all SMs, and with the spatial-sharing method, they cannot fully utilize allocated time slices.

These wasted computing resources (SMs, GPU time slices) are also GPU fragments, in addition to memory fragments.

In summary, although MIG guarantees the performance target as shown in Figure 5, it results in low GPU utilization due to its coarse-grained resource partitioning. MPS enables fine-grained resource usage but faces severe performance violations. Meanwhile, the API-remoting method experiences both performance violations and low GPU utilization.

2.3.3 Potential Solution

While the above results highlight the low utilization of current methods, they also reveal an opportunity to address this issue through spatio-temporal orchestration. For instance, if ResNet50 could utilize 44% of SMs with 50% time slices, or 62% of time slices on a 3/7 GPU instance, its performance target could be met precisely.

Insight-2: Spatio-temporal orchestration may bring higher utilization. Since we expand the adjustable parameter of GPU partitions into two dimensions of spatial and temporal, there exist multiple resource configurations to satisfy the performance target of a workload and the flexibility in orchestration is increased.

2.4 Challenges

It is non-trivial to achieve the potential solutions mentioned above in an efficient way.

For kernel-space interception, it is difficult to obtain the command buffer address because it is unveiled and not exposed by vendors. This requires us to accurately analyze the behavior of the KMD to create GPU context and malloc command buffer. We have to trace and manipulate the command buffers delicately to handle their dynamic address across processes and ensure their consistency.

For spatio-temporal orchestration, it is challenging to find the optimal orchestration plan to maximize the GPU utilization, because the orchestration space increases significantly with both spatial sharing space and time sharing space. Moreover, an efficient orchestrator is required to identify the resource allocation for all the co-located applications simultaneously while guaranteeing performance targets and minimizing GPU fragments at the same time.

3 KRYPTON Methodology

We therefore propose KRYPTON that promises the compatibility, isolation and performance targets of co-located applications, while reducing the GPU fragments.

3.1 Overview of KRYPTON

Figure 7 shows the design of KRYPTON. To find appropriate GPU resources for applications, each application is first

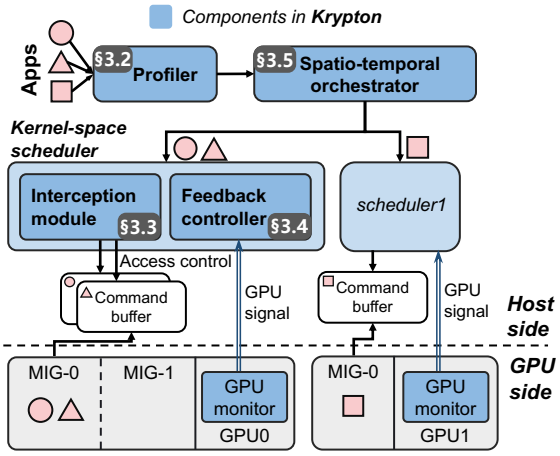


Figure 7: The overview of KRYPTON.

profiled with an offline profiler (§3.2). Prior works on GPU sharing [35, 41] also require offline profiling to attain information on applications. After each application specifies its performance target (e.g., throughput, end-to-end latency), the spatio-temporal orchestrator of KRYPTON then assigns GPU resources for the applications and deploys them properly to guarantee their performance targets (§3.5). The applications can either run in bare metal or be wrapped in containers and use their own runtimes.

The GPU resources include hardware units, GPU time slices, and device memory. The hardware units are divided using MIG, while time slices and device memory are allocated using kernel-space schedulers, with each scheduler managing the resources on the corresponding physical GPU. Each scheduler comprises the interception module (§3.3) and the feedback controller (§3.4). The interception module manages the access control of command buffers of applications to achieve time-sharing of the GPU. It also realizes GPU memory management in kernel space. The feedback controller leverages the GPU hardware's real-time signal from GPU monitors to adaptively adjust the time slices of applications to ensure performance-aware GPU sharing. It promises that the co-located applications will not impact each other and achieve the performance target.

3.2 Offline Profiling

It is important to understand the performance of applications under different resources. We use offline profiling to achieve this. The profiling is simple and non-intrusive to applications.

3.2.1 Performance Profiling of Applications

We use IGPU(s,t,m) to represent the GPU resources assigned to an application, where s is the quota of assigned GPU hardware units (i.e., the streaming multiprocessors, memory band-

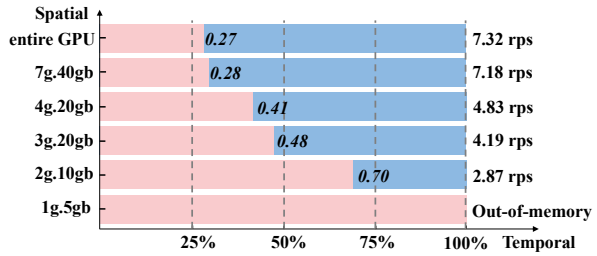


Figure 8: The performance chart of a Transformer-XL training job (bs=8) on A100 GPU. The configurations within the blue region can promise a throughput target of 2 rps. The x-axis represents the time slice quota and the y-axis represents spatial configurations.

width, cache and memory that are partitioned using MIG), t is the quota of assigned GPU time slices, and m is the assigned GPU memory size.

For static workloads such as training and offline rendering, the throughput can be treated as the performance metric, and the profiler records the throughput of an application on every possible IGPU configuration, represented as $Thr(s,t,m)$. For workloads that use GPU dynamically such as large language model serving, the profiler can still explore the maximum capability (e.g., tokens/s) that each IGPU configuration can offer the application as the guaranteed performance.

The profiler collects the performance of applications within a fixed configuration set. Specifically, for an Nvidia A100 GPU, the throughput of each application is collected on 6 different configurations IGPU$(s_i, 100\%, m_i)$, including the entire GPU and 5 types of MIG instances (7g.40gb, 4g.20gb, 3g.20gb, 2g.10gb and 1g.5gb*). The throughput is recorded as $Thr(s_i, 100\%, m_i)$, correspondingly. Besides, the memory consumption of applications is also recorded. Then the throughput of applications with various spatial and temporal quotas can be estimated directly with the profiled data, calculated as $Thr(s,t,m) = Thr(s,100\%,m) \times t$. The formula holds when strong isolation can be achieved and the throughput is proportional to the GPU time slice quota.

3.2.2 Find Feasible Configurations

KRYPTON leverages the profiled performance to find feasible resource configurations. As an example, Figure 8 reveals the feasible configurations that can satisfy the throughput requirement of a Transformer-XL training workload. The workload requires 6196MB GPU memory thus it cannot be offloaded on a 1g.5gb instance. Larger MIG instances can accommodate the workload. For example, in a 3g.20gb MIG instance, configurations with time slice quota larger than 48% can guarantee the throughput target of the workload. If a tenant has a preferred resource configuration, it can also bypass offline profiling and directly designate the required configuration.

*Nvidia A100 40GB can be sliced into 7 instances. 4g.20gb represents a MIG instance that uses 4/7 of hardware units and 20GB device memory.

3.3 Kernel Space Intercepting

IGPUs' hardware units are spatially assigned through MIG. To assign GPU time slices and device memory to an IGPU, KRYPTON leverages interception in kernel space.

3.3.1 Command Buffer Interception

To share a GPU temporally, KRYPTON intercepts the access to GPU command buffers without resolving specific commands. Specifically, KRYPTON hijacks the access control of the GPU, by identifying the addresses of command buffers, and controlling the access permission of the command buffers.

Command buffer address. It is non-trivial to obtain the address of command buffers since it is wrapped in Nvidia's kernel mode driver. We identify that the command buffer address is assigned through the `ioctl` system call every time a GPU context is created. We then intercept the `ioctl` calls and if the argument corresponds to the memory object allocation of GPU engines, we record the address of the memory object as one piece of the command buffer. Note that multiple command buffers and GPU contexts can be possessed by one process and the command buffers' addresses remain unchanged until destroyed.

Modify access permission of command buffers. The interception module locks the command buffers of a process when it is not permitted to control the GPU and unlocks them when the process is active.

As shown in Figure 9, to lock the command buffer of a process ❶, the interception module changes the protection flag of the corresponding memory page to read-only through the `do_mprotect_pkey` function, which is the kernel's internal implementation of the `mprotect` system call [24]. Then, whenever a user-space process tries to write to its GPU command buffer to launch kernels or submit other operations ❷, it will cause a segmentation fault. We lead the fault signal to a pre-registered user-mode signal handler ❸. The signal handler then inquires the interception module in a blocking manner to check if it is authorized to write to the command buffers ❹. It restores the process's execution once the command buffer is unlocked ❺.

Since we only handle the command buffers that interact with the GPU, the other CPU work of the process would not be influenced compared with techniques that directly pause the whole process [52]. The mechanism of command buffer interception is also promising for other PCIe devices that leverage command buffers (e.g., AMD GPUs [16]).

3.3.2 GPU Memory Management

KRYPTON also intercepts memory-related commands such as device memory allocation to precisely manage the device

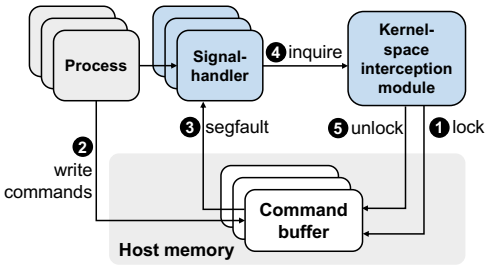


Figure 9: The interception of command buffer.

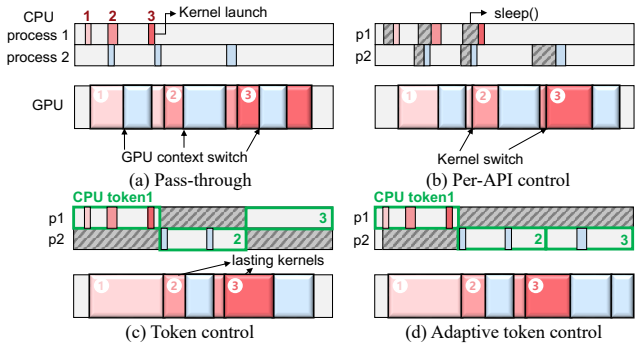


Figure 10: GPU execution pattern with different scheduling policies. Two processes share the GPU, and the kernels from different processes (red and blue) are interleaved with each other on the GPU.

memory usage of IGPUs. Those commands are triggered by UMD as `ioctl` system calls and KRYPTON also intercepts them in kernel space. Compared to the user space interception, the kernel space interception method can handle implicit memory allocation requests, such as those occurring during GPU context creation. KRYPTON leverages a central memory allocator to manage all device memory-related requests and records the actual memory usage of each workload. Requests either be passed through directly or return an error when the provisioned memory quota is exceeded.

3.4 Performance Guarantee with Feedback Control

The performance with time-sharing is promised by precisely controlling their allocated time slices. Since vendor driver and hardware scheduler monopolize the GPU context switch ability, KRYPTON controls the GPU time slice resources used by each process by managing *CPU tokens*. When a process possesses a token, it is permitted to utilize the GPU. Through the management of CPU tokens, we can schedule and govern the time slice resources of the GPU. On this basis, we make further adaptive adjustments to the control of CPU tokens in accordance with the feedback signals from GPU monitors.

3.4.1 Token Control at Host Side

Due to the unveiled internal control mechanism and varying architectural design of commodity GPUs, it is difficult to modify drivers to realize GPU time slice allocation. The GPU context switch is decided by the driver as shown in Figure 10(a) when processes use GPU in pass-through mode.

To this end, researchers exploit to control GPU time slice usage at the host side. Some works schedule with per-API control policies [43, 72, 76]. They adjust the kernel launch rate by adding sleep time between the API trap and the API emulation as shown in Figure 10(b). The sleep duration is determined by the kernel properties (i.e., CUDA block number). In some other work, the host treats the time that the CPU holds the token as the GPU time [56]. However, because of the asynchronous execution manner of the GPU, limiting the CPU token window length for an application cannot promise to achieve the same limitation on the GPU time. For example, if long kernels are launched within the first CPU token window as Process 1 illustrated in Figure 10(c), then with the launched kernels residing on the GPU, process 1 would occupy longer GPU time slices than expected, squeezing the GPU time slices that should have been assigned to process 2 in the second token window.

We observe that it is possible to adjust the GPU time slice allocation by controlling CPU tokens adaptively. As shown in Figure 10(d), since Process 1 occupies more GPU time slices than anticipated, the CPU token within the third time window is compensated to Process 2. Such an adaptive CPU token control can promise precise GPU utilization, thereby guaranteeing the required performance. It is achievable by leveraging GPU feedback information.

3.4.2 Control GPU Utilization with Feedback

With real-time GPU utilization signal as feedback, KRYPTON compensates for applications with lower-than-expected utilization and constrains the usage of overused applications. This is achievable by using Nvidia's user-space hardware management libraries (e.g., Nvidia's NVML [13], DCGM [9]) to monitor the per-process GPU time slice utilization[†], which is the sum of the GPU utilization of all GPU contexts created by the process.

We are then able to control the GPU usage of applications adaptively with the GPU utilization information as feedback, following Algorithm 1. Specifically, the CPU token is allocated by the scheduler and the scheduler guarantees that only one application can hold the token at any given time. The activated application is the one that has the lowest relative GPU utilization compared to the promised time slice quota. Such an elastic policy can fully utilize the GPU and promise

[†]The percentage of time slices the process uses the GPU, with at least one SM computing.

Algorithm 1 Adaptive CPU token control policy

```
function SELECTAPP(app_list)
    while true do
        UpdateAppUtil(app_list, Utils)
        selected_app = argmin(app_list.(util/quota))
        Activate(selected_app)
        Pause(app_list\selected_app)
        CPUSleep(token_length)
function UPDATEAPPUTIL(app_list, GPUUtil, ProcessUtil)
    if Monitor == DCGM then
        last_selected_app.util += GPUUtil
    if Monitor == NVML then
        for app ∈ app_list do
            for p ∈ app.process_list do
                app.util += app.p.ProcessUtil
    for app ∈ app_list do
        MovingAverage(app.util)
```

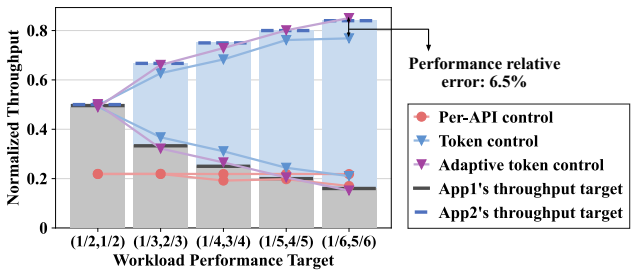


Figure 11: The effectiveness of different scheduling policies on satisfying performance targets. Two ResNet50-bs32 inference applications (App1 and App2) share an Nvidia A100 GPU and have different performance targets. The fractions annotated on the x-axis represent the ratios between the target throughput and the stand-alone throughput of App1 and App2, respectively.

applications' performance simultaneously.

When the GPU load of an application is light, its GPU utilization may be lower than the assigned proportion of CPU tokens. To avoid always selecting such an application, we regard the application as fully occupying a GPU time slice if only its own is observed within that time slice, no matter how much it occupies that GPU time slice.

With the above design, the proposed adaptive CPU token control policy can precisely promise performance as shown in Figure 11. In particular, the token length is set to 100*ms* in KRYPTON unless otherwise stated. Adaptive token control promises an average relative error no larger than 0.9% compared with the performance target. Without feedback control, the throughput of co-located applications deviates from the performance target and the relative error is 5.3% on average. The per-API control policy [43] performs even worse and can not isolate the throughputs as expected.

3.5 Spatio-Temporal Orchestration

The goal of the orchestration is to reduce the number of GPUs to use, while maintaining the performance requirements of workloads. It is non-trivial to allocate appropriate resources for workloads and improve overall cluster utilization. Suppose there are M workloads to be orchestrated and there are N configurations for each workload. In this case, there are total N^M possible IGPU configurations. To place M workloads onto P GPU instances, there are also P^M possibilities. The search space complexity for orchestrating M workloads is as large as $O(N^M P^M)$.

3.5.1 Resource Defragmentation Strategy

To reduce the searching complexity, we design a two-stage bin packing algorithm that heuristically searches for a near-optimal orchestration plan, aiming to minimize the GPU fragments. The searching process is an offline procedure as shown in Figure 12. (1) Initialization: each workload is assigned a qualified IGPU that has the minimum spatial resources. (2) Temporal packing: KRYPTON tries to fuse the IGPUs that have the same spatial configurations to reduce the idle time slices. (3) Fragment reduction: KRYPTON further reduces the temporal fragments within the IGPUs, by migrating workloads from spatially smaller IGPUs to spatially larger IGPUs. Specifically, for IGPUs with the same spatial configurations, we sort them in descending order of the left time slice proportion and attempt to migrate. In this way, IGPUs with large temporal fragments are first eliminated. Note that, the memory residual of IGPUs is also considered during the migration. (4) Instance packing: When the MIG instances are finally determined, bin-packing is used to allocate MIG instances to reduce the number of used GPUs. Note that we do not need to pack the real IGPU instances or migrate workloads in reality in each step, but only need to orchestrate applications following the final result of this planning algorithm.

The complexity of the proposed two-stage bin-packing algorithm is $O(MP)$.

3.5.2 Workloads on Multi-GPU

KRYPTON also allows workloads to use more than one GPU. When the application itself can run on multiple GPUs, KRYPTON can use fewer GPUs to satisfy the performance. For example, if a distributed training workload requires 2 GPUs to run but only aims for 60% of the solo throughput, KRYPTON can assign 60% of the time slices on each physical GPU to it. KRYPTON carefully aligns the tokens to use different GPUs in kernel space, because the workload running on isolated GPUs may communicate with each other through NCCL [6], and correlated GPUs must be active simultaneously. When searching for the orchestration plan, the configuration of workloads that use multi-GPU should remain unchanged.

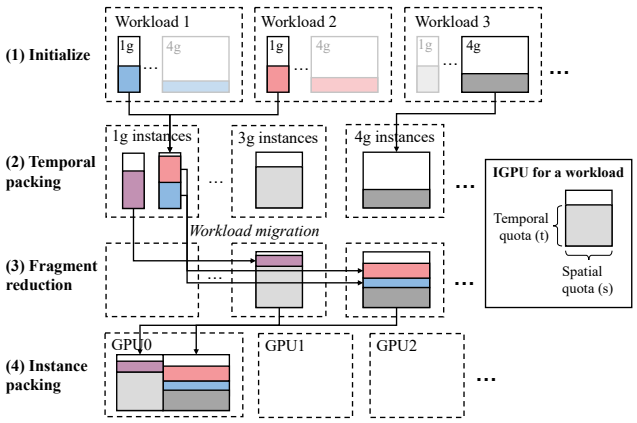


Figure 12: Steps of the resource defragmentation strategy.

4 Implementation

We implement the kernel-space interception and performance feedback controller in 3K lines of C code and the spatio-temporal orchestrator is composed of 1K lines of Python code.

The kernel-space scheduler is implemented as a loadable kernel module within the Linux operating system. Moreover, the scheduler does not interfere with the GPU driver. Users can either install the open source kernel module driver [12] or directly install the official drivers. The entire system has no requirement to modify the application code, imposing zero programming burden on users.

In kernel space, for each GPU instance, there is a corresponding simulated device file `/dev/igpu` which is mounted to the device file `/dev/nvidia`. When processes within the container try to access the Nvidia device files, the requests are forwarded to the simulated device file and controlled by the interception module of the kernel space scheduler. The kernel space thread calls the `do_mprotect_key` to modify the protection flag of a virtual space address. Since the `do_mprotect_key` API is not exported by the Linux kernel, we leverage the kallsyms mechanism to find its symbol.

The kernel module additionally controls the device file to interact with the signal handlers and the GPU monitor. Specifically, the blocking check of the signal handler sends the `read` system call to the device file to inquire if it is activated. The GPU monitor delivers the GPU utilization information through the `ioctl` system call to the feedback controller in kernel space. The GPU monitor runs as an independent daemon process in user space. When the GPU is not in the MIG mode, the `nvmlGetProcessUtilization` API from the NVML library is enabled. When the MIG mode is enabled, the DCGM library is used as the monitor since NVML's functions are not fully supported with MIG. Applications run in docker containers. Each container compiles a dynamic linked library when initialized and adds it to the `LD_PRELOAD` environmental variable. Then the container can interact with the kernel space through the dynamic library.

Table 1: Hardware and software specifications.

System Overview	
CPU	Intel Xeon Silver 4216 64-core
GPU	Nvidia A100 40GB & 2 GeForce RTX 4090
Kernel version	5.15.0-91-generic
Operating System	Ubuntu 20.04
Docker version	24.0.7
Nvidia Driver	550.90.07
Runtimes	CUDA 12.1, Vulkan 1.3
AI framework	Pytorch 2.2.1

Table 2: Benchmarks used in the evaluation ($*$ represents batch size = 16 due to the large GPU memory usage).

Application	Type	Throughput (rps)		Memory (MB)	
		bs=8	bs=32	bs=8	bs=32
ResNet50 (RN) [46]	infer.	162.2	73.7	964	1222
	train	42.3	24.7	1706	3608
MobileNet_V2 (MN) [62]	infer.	184.1	148.5	788	1156
	train	42.3	34.4	1452	3290
BERT-Large (BT) [69]	infer.	6.6	1.9	2032	2752
	train	1.8	1.0*	13700	22384*
Transformer-XL (TF) [69]	infer.	42.5	12.6	1536	2524
	train	7.3	2.9	6196	12094
Computecullandlod (ccl)	render	172.2 fps		54	
Deferredmultisampling (dms)	render	263.0 fps		963	
Indirectdraw (idd)	render	194.9 fps		60	
Throughput requirement	R1	[2/3, 1/2, 1/3, 1/6, 1/12]			
	R2	[1/2, 1/4, 1/6, 1/8, 1/10]			

5 Evaluation of KRYPTON

In this section, we evaluate KRYPTON in reducing the required GPU number while promising the compatibility, isolation and performance of co-running workloads.

5.1 Experimental Setup

Table 1 shows the specifications of the experimental platform. To justify the compatibility, as shown in Table 2, the used applications cover AI applications (inference and training) that use CUDA runtime [20] and rendering applications [30] that use Vulkan runtime [29]. Each application runs in a separate container equipped with Nvidia Container Toolkit [25]. For each application, there are two sets of throughput requirements: *R1* and *R2*. For example, the $2/3$ throughput requirement represents that the throughput target is set to be $2/3$ of the solo throughput running on an entire physical GPU.

We compare KRYPTON with baselines that ensure isolation: *Temporal* [8, 28], *Best-fit-MIG*, and *GPUlet-MIG* [35]. With *Temporal*, workloads share the GPU temporally, with each workload allocated a fixed quota of time slices [8, 28]. The quota represents the minimum calculated time slice required

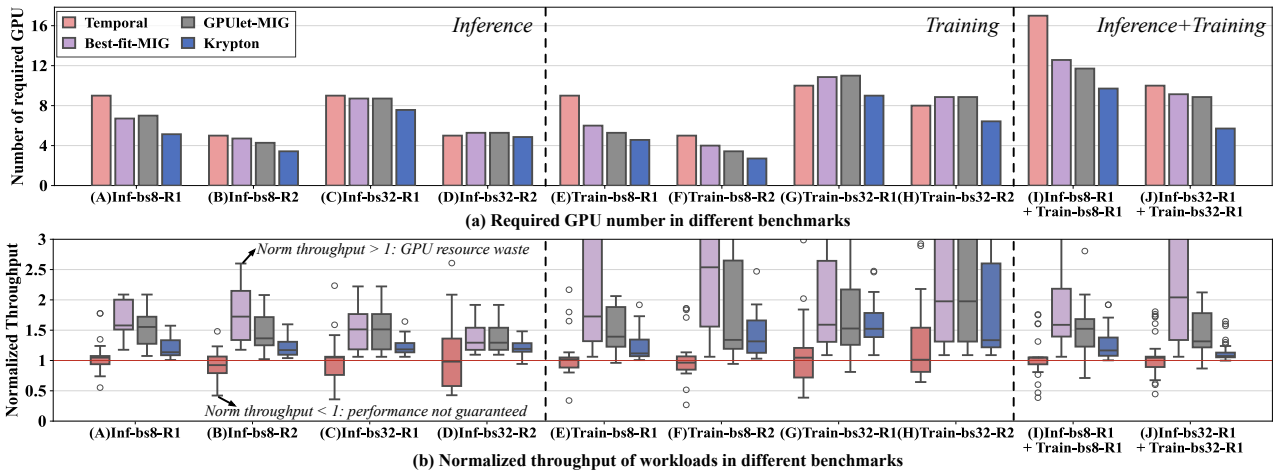


Figure 13: Comparison between KRYPTON and other methods in reducing GPU number and promising performance.

to meet the specified throughput target. With *Best-fit-MIG*, each workload is assigned the smallest MIG instance capable of satisfying its performance requirements. MIG instances are bin-packed for placement within the cluster. This approach is equivalent to the greedy best-fit algorithm described in GPUlet [35]. With *GPUlet-MIG*, GPUlet [35] identifies the optimal spatial configuration of a workload and packs workloads in temporal duty cycles. It is originally implemented using MPS. We have improved GPUlet using MIG and let workloads share a GPU instance temporally as *Temporal*.

We use two metrics in the evaluation. For evaluating the resource efficiency, we use the metric "**number of required GPUs**" to deploy workloads while guaranteeing their performance target. Fewer GPUs indicate better GPU utilization. For evaluating the performance, we use the metric "**normalized throughput**" of applications (the actual throughput normalized to the throughput target). As long as the normalized throughput is larger than 1, the performance target is promised. Note that, if a benchmark's normalized throughput is too high, it may harm the utilization of the whole system. This is because excessively high throughput often indicates over-allocation of resources, resulting in wastage.

5.2 Overall Performance

We build 10 test cases to cover scenarios involving inference jobs co-running, training jobs co-running, and inference jobs co-running with training jobs. The case for runtimes other than CUDA is evaluated in Section 5.3. For easing of description, a test case is named in the form of "Inf/Train-bs8/32-R1/R2". For instance, "Inf-bs8-R1" represents the case that 20 workloads co-run (4 inference models $\times$ 5 workloads with different throughput requirements as R1 for each model). The batch size of workloads in "Inf-bs8-R1" is 8 and the throughput requirement list R1 is in Table 2. There are 20 workloads in inference-only (A-D) and training-only (E-H) test cases, and 40 workloads in test cases I and J.

Figure 13(a) shows the number of required GPUs to host the workloads in the 10 test cases. As observed, KRYPTON reduces the total required GPU number by 32.1%, 23.1%, and 20.5% compared with *Temporal*, *Best-fit-MIG*, and *GPUlet-MIG*, respectively. The reduction of the required GPU number indicates the improvement of GPU utilization. We can also observe that KRYPTON performs better when the batch sizes of are small, since small workloads are more prone to be packed together, thereby reducing the GPU fragments. KRYPTON also performs better when the number of workloads is large, as the increasing workload number helps to fulfill GPUs.

Corresponding to Figure 13(a), Figure 13(b) shows the normalized throughputs of the workloads in each test case. As observed, some workloads' performance is not promised with *Temporal*, because it lacks feedback control, leading to interference among GPU time slices. The performance targets are guaranteed with *Best-fit-MIG*, *GPUlet-MIG* and KRYPTON. Workloads have the highest throughput with *Best-fit-MIG*, because it configures each workload to independently use a MIG instance. A higher normalized throughput above 1 indicates wasted GPU fragments, as a workload may not need a whole MIG instance. *GPUlet-MIG* reduces the fragments by allowing GPU partitions with the same spatial quota to fuse temporally. However, since the configurations of workloads are fixed in *GPUlet-MIG*, GPU fragments are still not fully explored. KRYPTON leverages the property that the throughput target of a workload can be promised by various configurations, thereby further reducing the fragments temporally and spatially. The average normalized throughput of KRYPTON is the closest to 1, which indicates that the workloads have fully utilized the allocated resources.

To understand how the workloads are actually orchestrated, Figure 14 shows the orchestration of the workloads with

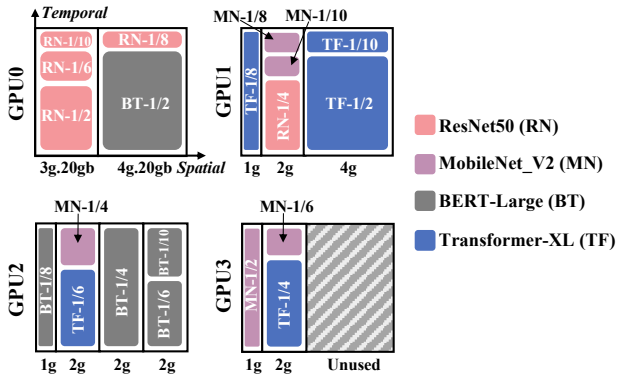


Figure 14: Spatio-temporal orchestration of workloads in the Inf-bs8-R2 benchmark on 4 A100 GPUs. The horizontal dimension refers to the partitioning strategy of spatial resources (MIG instances) and the vertical dimension refers to the temporal partitioning ratio.

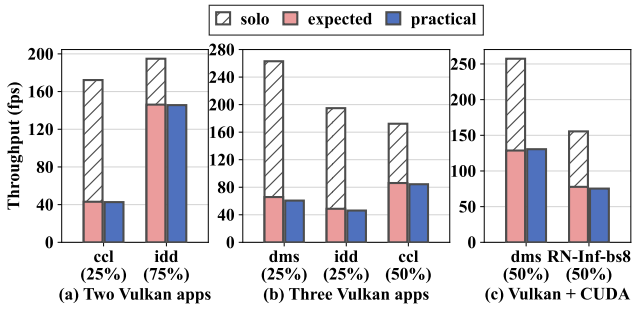


Figure 15: The performance of KRYPTON with Vulkan.

KRYPTON in the test case Inf-bs8-R2. As observed, the spatio-temporal orchestration of KRYPTON can efficiently deploy the 20 workloads onto 3.4 A100 GPUs, while promising the performance targets.

5.3 Performance with the Vulkan Runtime

This subsection evaluates the effectiveness of KRYPTON for runtimes beyond CUDA. Since both API-remoting and MIG do not support other runtimes (e.g., Vulkan), *Temporal*, *Best-fit-MIG* and *GPUlet-MIG* cannot run here. Meanwhile, only time-sharing is enabled in KRYPTON in this case.

In this experiment, we evaluate scenarios where two Vulkan applications co-run, three Vulkan applications co-run, and a Vulkan application co-runs with a CUDA-based AI application. Figure 15 presents the results for three example combinations, with other combinations yielding similar outcomes. As shown, the average relative error in applications' throughput, compared to their performance targets, remains below 3.4%. This demonstrates that KRYPTON can effectively support multiple runtimes through kernel-space interception.

5.4 Effectiveness of Feedback Controller

This subsection evaluates the effectiveness of the performance feedback controller of KRYPTON in promising the performance targets. We compare KRYPTON with the pass-through method [25], the user-space API-remoting method GaiaGPU [43], and KRYPTON w/o adaptive control, a variant of KRYPTON that disables the adaptive token control policy introduced in §3.4.2. In this experiment, we let 2, 4, and 8 *RN* training jobs (batch size = 32) time-share an entire A100 GPU. The feedback controller is responsible for ensuring the performance targets.

Figure 16(a) shows the achieved normalized throughput of the co-running applications. The pass-through method cannot promise the throughput target, and the relative throughput error is 53.6% compared with the throughput target. The overhead of GaiaGPU is large, resulting in a 24.2% lower throughput of applications than expected. In KRYPTON w/o adaptive control, the throughput of applications deviates from the target and the relative error is 21.6% on average. After applying the adaptive control policy, the relative error of applications' throughput is merely 3.0% on average.

In more detail, Figure 16(b) showcases the GPU utilization when two applications co-run within two minutes. GPU utilization is strongly correlated with the throughput of applications. A closer GPU utilization to the expected normalized throughput indicates a more accurate assurance of performance. As shown, the utilization curve for KRYPTON with adaptive control shows less volatility and maintains a steadier rate, closer to the target utilization levels (25% and 75%), suggesting that KRYPTON outperforms the other methods.

We also evaluate KRYPTON in a multi-GPU scenario by launching a Transformer-XL distributed training job on two RTX4090 GPUs. We set its throughput target to 10%, 20%, $\cdots$, 100% of its solo throughput and then assign it the corresponding temporal quota. Experimental results show that the throughput of the distributed training workload is still proportional to the time slice ratio assigned and the relative error of throughput is only 1.8%.

5.5 Guaranteeing the Latency QoS

KRYPTON could also be used to provide services that have latency-related Quality-of-Service (QoS).

Assume an application is active for 50ms and asleep for another 50ms with token control. In this scenario, the requests that arrive within the active token can be delivered to the GPU properly. Otherwise, they may wait 50*ms* and violate the QoS target that is smaller than 50*ms*. Reducing the token length implies more frequent switching between applications, which would increase overhead.

We use the inference of Transformer-XL with different batch sizes to show the performance of the latency QoS guarantee. The arrival rate of the requests follows the Poisson

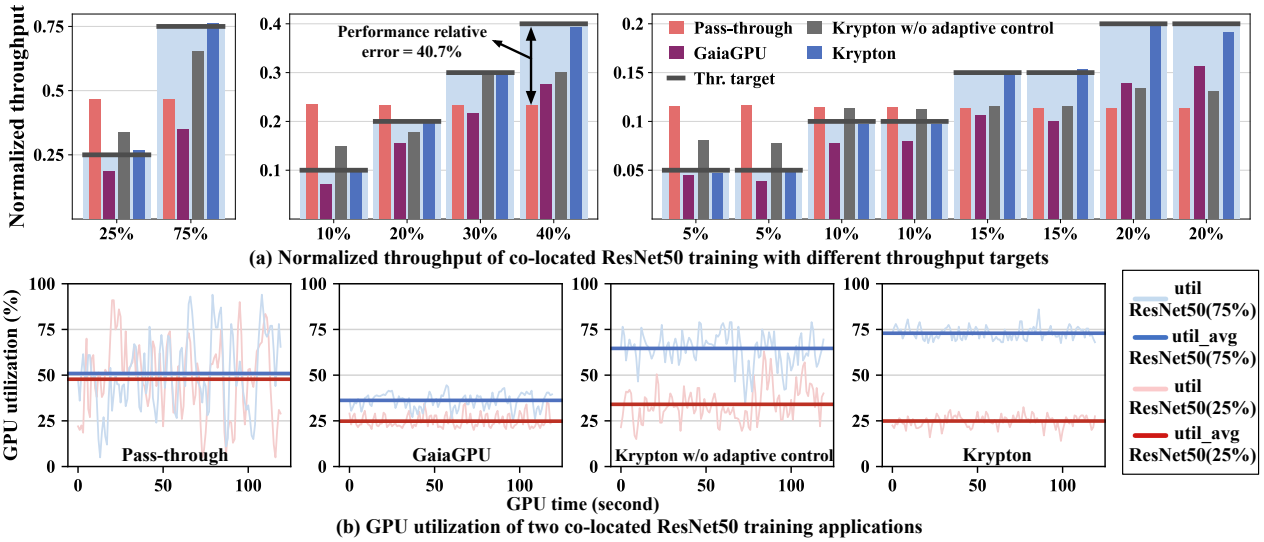


Figure 16: Comparison of different methods in promising the performance target of ResNet50 training applications.

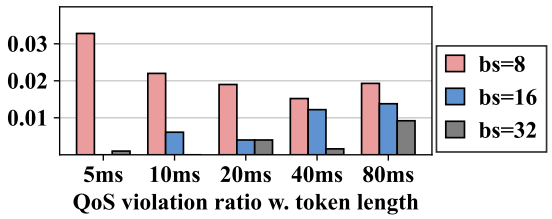


Figure 17: The QoS violation ratio and overhead of KRYPTON with different token lengths.

distribution as MLPerf [61], and the input rps is half of the maximum throughput that the service can achieve. The QoS target is set to the p99 latency, which means there will be 1% QoS violation in this original setting. When the performance requirement is halved, we halve the input rps and assign 50% time slices to the service correspondingly. We check if KRYPTON can still promise the original QoS target.

As shown in Figure 17, for smaller batch size, QoS violations increase as token lengths decrease. This trend arises because shorter tokens introduce more overhead, which affects smaller batches where individual request latencies are inherently low. In contrast, larger batch sizes exhibit an increase in QoS violations with longer token lengths. This is because tasks with larger batch sizes can hardly be finished within a single time slice. Thus, longer token lengths lead to extended waiting times for tasks to acquire time slices, thereby increasing the likelihood of QoS violations.

Overall, the average QoS violation rate across all configurations is measured at 1.23%, underscoring KRYPTON's ability to guarantee latency-related QoS targets.

5.6 Elastic GPU Sharing

KRYPTON guarantees an application's performance regardless of the co-running applications. When applications are assigned fixed GPU time slices using feedback control, the average throughput relative error of a Transformer-XL-Inf-bs8 workload is 1.3% when it co-runs with different applications.

The adaptive token control algorithm (Algorithm 1), however, can further improve performance, by allowing an application to use the unused time slices of its located GPU instance. Compared with methods that provide fixed GPU time slices to workloads, the elastic sharing characteristic improves the performance of workloads as much as possible. As shown in Figure 18, when a ResNet50-Train-bs32 job runs on a GPU alone, its utilization can approach 99%. When a new task Transformer-XL-Inf-bs8 arrives, the two jobs will share the complete GPU instantly. A utilization burst can be observed when a new task arrives. This is because the scheduler always selects the workload that has the lowest relative utilization as introduced in Algorithm 1. The burst only lasts several seconds, and the scheduler can then adaptively adjust the time slices assigned to each workload afterward.

5.7 Overhead

Except for the overhead related to CPU token as discussed in §5.5, KRYPTON does not introduce extra overhead when monitoring utilization. KRYPTON does not break the original logic of context switching handled by the vendor driver. As for the overhead at the host side, KRYPTON's kernel module occupies 4.8MB of CPU memory, mainly storing the information of running processes. Besides, the GPU monitor has to occupy a CPU thread to communicate with the kernel module.

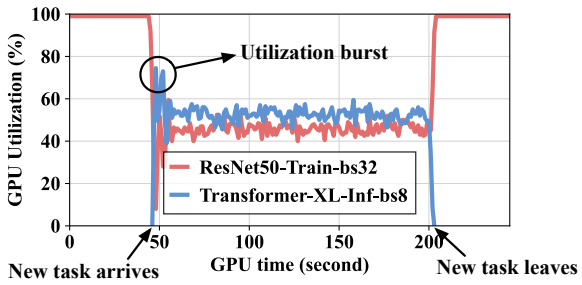


Figure 18: Elastic sharing of a GPU.

6 Related Works

GPU Sharing on Cloud. There are many works on multiplexing a GPU to deploy applications in specific scenarios (e.g., AI training [31, 48, 54, 57, 71, 74], AI inference [37, 39, 42, 44, 63], cloud gaming [34, 79]). The applications are deployed either in virtual machines, or lightweight environments such as bare-metal and containers. To control the GPU usage inside virtual machines, full virtualization [65, 67, 77] and mediated pass-through [60, 68] intercept the hardware interface of the GPU (MMIO). Paravirtualization [4, 40] necessitates a custom driver in every supported guest OS for the virtual device. GPU sharing across containers or bare-metal vastly uses API-remoting to manage GPU resources. The host decides whether or when to pass the intercepted call. GaiaGPU [43], qcuda [55], and vcuda [64] all leverage API-remoting in user space to restrict the applications' resources to use CUDA. TGS [72] and Orion [66] co-locate production jobs and opportunistic jobs based on API-remoting on a GPU.

Resource management in GPU sharing. Typically, GPU sharing techniques can be classified to be time sharing [11, 43, 64, 72] and spatial sharing [1, 2, 7, 32]. **Time sharing** is the default pattern when the GPU is shared in the pass-through mode [70, 75]. The GPU controls the GPU context switch in a round-robin manner [11, 32, 33]. Nvidia proposes vGPU GRID [26] to share the GPU among VMs with fixed quotas temporally. Some virtualization works [51, 67] customize hardware drivers and apply various scheduling schemes [47, 56, 67] to share a GPU temporally. **Spatial sharing** methods such as MPS [2] and MIG [7] manage the SM and memory resources in GPU sharing. To fully utilize the GPU, efforts in spatio-temporal sharing have also been explored vastly. Existing works either leverage multiple streams [1, 3] to multiplex a GPU within an application [54, 71], or leverage MPS to achieve so [39, 78]. There are also other works [45, 49, 58] that collaborate with the efforts in code compilation to multiplex a GPU. However, these works do not consider the isolation demand of tenants.

7 Discussion

KRYPTON has been deployed within the production environment. In actual practice, the minimum splittable GPU time slice ratio is set at 1%, and the minimum unit of memory allocation is 1MB. In this section, we shall deliberate on how KRYPTON maintains system security and the potential limitations it may encounter.

System Security. The kernel-space scheduler of KRYPTON is designed and implemented as a loadable kernel module (LKM), and we do not have to make any changes to the kernel or Nvidia driver. Therefore, KRYPTON maintains equivalent security guarantees to other LKM-based approaches. To date, no security risks associated with kernel-space interception have been reported that could introduce system vulnerabilities. Additionally, KRYPTON has been deployed in production-scale data center clusters featuring thousands of GPUs, operating reliably for over one year.

Limitations. KRYPTON does not support real-time spatial resource re-configuration with MIG. This may pose a potential constraint when a dynamic application is required to manage fluctuating workloads. For instance, in the case where the load of a large language model serving application decreases and demands fewer resources, the scheduler within the KRYPTON is unable to directly and elastically recycle the resources of the tenant. The live migration of GPU resources while ensuring continued isolation remains a promising area for future research.

8 Conclusion

In this paper, we propose KRYPTON, a kernel-space GPU sharing system that promises the compatibility and isolation of applications that share GPUs. KRYPTON intercepts GPU command buffers at the kernel level to provide virtual GPU devices. We design a feedback-based performance control policy to guarantee the performance of applications. Leveraging the spatio-temporal orchestration policy, KRYPTON reduces GPU fragments, thereby reducing the required number of GPUs for workloads. We evaluate KRYPTON with numerous applications and benchmarks. KRYPTON reduces the required GPU number by 32.1% compared with state-of-the-art systems.

Acknowledgments

We sincerely thank our shepherd, Michael Le, and the anonymous reviewers for their insightful comments to improve the paper. This work is partially sponsored by the National Key Research and Development Program of China (2023YFB3001504), National Natural Science Foundation of China (62302302, 62232011), and Natural Science Foundation of Shanghai Municipality (24ZR1430500). Quan Chen is the corresponding author.

References

[1] Nvidia cuda stream management. https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__STREAM.html, 2012.

[2] Nvidia multi-process service. https://docs.nvidia.com/deploy/mps/index.html, 2012.

[3] Amd rocm stream management. https://rocmdocs.amd.com/projects/HIP/en/develop/.doxygen/docBin/html/group___stream.html, 2016.

[4] Vmware svga3d guest driver. https://www.mesa3d.org/vmware-guest.html, 2019.

[5] Application considerations in mig. https://docs.nvidia.com/datacenter/tesla/mig-user-guide/index.html#app-considerations, 2020.

[6] Nvidia collective communications library. https://developer.nvidia.com/nccl, 2020.

[7] Nvidia multi-instance gpu. https://www.nvidia.com/en-us/technologies/multi-instance-gpu/, 2020.

[8] Alibaba cloud cgpu. https://www.alibabacloud.com/help/en/elastic-gpu-service/latest/what-is-the-cgpu-service, 2021.

[9] Nvidia data center gpu manager. https://developer.nvidia.com/dcgm, 2021.

[10] Chatgpt. https://openai.com/blog/chatgpt, 2022.

[11] Nvidia drive os linux sdk developer guide. https://developer.nvidia.com/docs/drive/drive-os/6.0.8.1/public/drive-os-linux-sdk/common/topics/graphics_content/TegraGPUSchedulingImprovements1.html, 2022.

[12] Nvidia linux open gpu kernel module source. https://github.com/NVIDIA/open-gpu-kernel-modules, 2022.

[13] Nvidia management library. https://developer.nvidia.com/management-library-nvml, 2022.

[14] Opengl. https://developer.nvidia.com/opengl, 2022.

[15] Stable diffusion. https://stability.ai/stable-diffusion, 2022.

[16] Amd gpuopen. amd gpu isa documentation. https://gpuopen.com/amd-gpu-architecture-programming-documentation/, 2023.

[17] Envytools. https://github.com/envytools/envytools, 2023.

[18] Nouveau: Accelerated open source driver for nvidia cards. https://nouveau.freedesktop.org/, 2023.

[19] containerd. https://containerd.io/, 2024.

[20] Cuda toolkit. https://docs.nvidia.com/cuda/index.html, 2024.

[21] Cuda toolkit archive. https://developer.nvidia.com/cuda-toolkit-archive, 2024.

[22] Delivering video content with fractional gpus in containers on amazon eks. https://aws.amazon.com/blogs/containers/delivering-video-content-with-fractional-gpus-in-containers-on-amazon-eks/, 2024.

[23] docker. https://www.docker.com/, 2024.

[24] Linux manual page: mprotect. https://man7.org/linux/man-pages/man2/mprotect.2.html, 2024.

[25] Nvidia container toolkit. https://docs.nvidia.com/datacenter/cloud-native/container-toolkit/latest/index.html, 2024.

[26] Nvidia virtual gpu software. https://docs.nvidia.com/grid/latest/grid-vgpu-user-guide/, 2024.

[27] Openai sora. https://openai.com/sora/, 2024.

[28] Tencent kubernetes engine qgpu. https://www.tencentcloud.com/document/product/457/42973, 2024.

[29] Vulkan. https://www.vulkan.org/, 2024.

[30] Vulkan c++ examples and demos. https://github.com/SaschaWillems/Vulkan, 2024.

[31] Zhihao Bai, Zhen Zhang, Yibo Zhu, and Xin Jin. {PipeSwitch}: Fast pipelined context switching for deep learning applications. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 499–514, 2020.

[32] Joshua Bakita and James H Anderson. Hardware compute partitioning on nvidia gpus. In *2023 IEEE 29th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 54–66. IEEE, 2023.

[33] Nicola Capodieci, Roberto Cavicchioli, Marko Bertogna, and Aingara Paramakuru. Deadline-based scheduling for gpu with preemption support. In *2018 IEEE Real-Time Systems Symposium (RTSS)*, pages 119–130. IEEE, 2018.

[34] Binghao Chen, Han Zhao, Weihao Cui, Yifu He, Shulai Zhang, Quan Chen, Zijun Li, and Minyi Guo. Maximizing the utilization of gpus used by cloud gaming through adaptive co-location with combo. In *Proceedings of the 2023 ACM Symposium on Cloud Computing*, pages 265–280, 2023.

[35] Seungbeom Choi, Sunho Lee, Yeonjae Kim, Jongse Park, Youngjin Kwon, and Jaehyuk Huh. Serving heterogeneous machine learning models on {Multi-GPU} servers with {Spatio-Temporal} sharing. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 199–216, 2022.

[36] Per Christensen, Julian Fong, Jonathan Shade, Wayne Wooten, Brenden Schubert, Andrew Kensler, Stephen Friedman, Charlie Kilpatrick, Cliff Ramshaw, Marc Bannister, et al. Renderman: An advanced path-tracing architecture for movie rendering. *ACM Transactions on Graphics (TOG)*, 37(3):1–21, 2018.

[37] Weihao Cui, Han Zhao, Quan Chen, Ningxin Zheng, Jingwen Leng, Jieru Zhao, Zhuo Song, Tao Ma, Yong Yang, Chao Li, et al. Enable simultaneous dnn services based on deterministic operator overlap and precise latency prediction. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15, 2021.

[38] Lorenzo Dematté and Davide Prandi. Gpu computing for systems biology. *Briefings in bioinformatics*, 11(3):323–333, 2010.

[39] Aditya Dhakal, Sameer G Kulkarni, and KK Ramakrishnan. Gslice: controlled spatial sharing of gpus for a scalable inference platform. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, pages 492–506, 2020.

[40] Micah Dowty and Jeremy Sugerman. Gpu virtualization on vmware's hosted i/o architecture. *ACM SIGOPS Operating Systems Review*, 43(3):73–82, 2009.

[41] Anshuman Goswami, Jeffrey Young, Karsten Schwan, Naila Farooqui, Ada Gavrilovska, Matthew Wolf, and Greg Eisenhauer. Gpushare: Fair-sharing middleware for gpu clouds. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 1769–1776. IEEE, 2016.

[42] Jianfeng Gu, Yichao Zhu, Puxuan Wang, Mohak Chadha, and Michael Gerndt. Fast-gshare: Enabling efficient spatio-temporal gpu sharing in serverless computing for deep learning inference. In *Proceedings of the 52nd International Conference on Parallel Processing*, pages 635–644, 2023.

[43] Jing Gu, Shengbo Song, Ying Li, and Hanmei Luo. Gaiagpu: Sharing gpus in container clouds. In *2018 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Ubiquitous Computing & Communications, Big Data & Cloud Computing, Social Computing & Networking, Sustainable Computing & Communications (ISPA/IUCC/BDCloud/SocialCom/SustainCom)*, pages 469–476. IEEE, 2018.

[44] Arpan Gujarati, Reza Karimi, Safya Alzayat, Wei Hao, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace. Serving {DNNs} like clockwork: Performance predictability from the bottom up. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 443–462, 2020.

[45] Mingcong Han, Hanze Zhang, Rong Chen, and Haibo Chen. Microsecond-scale preemption for concurrent {GPU-accelerated}{DNN} inferences. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 539–558, 2022.

[46] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[47] Cheol-Ho Hong, Ivor Spence, and Dimitrios S Nikolopoulos. Gpu virtualization and scheduling methods: A comprehensive survey. *ACM Computing Surveys (CSUR)*, 50(3):1–37, 2017.

[48] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, et al. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *Advances in neural information processing systems*, 32, 2019.

[49] Saksham Jain, Iljoo Baek, Shige Wang, and Ragunathan Rajkumar. Fractional gpus: Software-based compute and memory bandwidth reservation for gpus. In *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 29–41. IEEE, 2019.

[50] Shinpei Kato. Implementing open-source cuda runtime. In *Proc. of the 54the Programming Symposium*, 2013.

[51] Shinpei Kato, Michael McThrow, Carlos Maltzahn, and Scott Brandt. Gdev:{First-Class}{GPU} resource management in the operating system. In *2012 USENIX*

Annual Technical Conference (USENIX ATC 12), pages 401–412, 2012.

[52] Oren Laadan and Jason Nieh. Transparent checkpoint-restart of multiple processes on commodity operating systems. In *USENIX Annual Technical Conference*, pages 323–336, 2007.

[53] Baolin Li, Tirthak Patel, Siddharth Samsi, Vijay Gadepally, and Devesh Tiwari. Miso: exploiting multi-instance gpu capability on multi-tenant gpu clusters. In *Proceedings of the 13th Symposium on Cloud Computing*, pages 173–189, 2022.

[54] Gangmuk Lim, Jeongseob Ahn, Wencong Xiao, Youngjin Kwon, and Myeongjae Jeon. Zico: Efficient {GPU} memory sharing for concurrent {DNN} training. In *2021 USENIX Annual Technical Conference (ATC 21)*, pages 161–175, 2021.

[55] Yu-Shiang Lin, Chun-Yuan Lin, Che-Rung Lee, and Yeh-Ching Chung. qcuda: Gpgpu virtualization for high bandwidth efficiency. In *2019 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 95–102. IEEE, 2019.

[56] Konstantinos Menychtas, Kai Shen, and Michael L Scott. Disengaged scheduling for fair, protected access to fast computational accelerators. *ACM SIGARCH Computer Architecture News*, 42(1):301–316, 2014.

[57] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R Devanur, Gregory R Ganger, Phillip B Gibbons, and Matei Zaharia. Pipedream: generalized pipeline parallelism for dnn training. In *Proceedings of the 27th ACM symposium on operating systems principles*, pages 1–15, 2019.

[58] Kelvin KW Ng, Henri Maxime Demoulin, and Vincent Liu. Paella: Low-latency model serving with software-defined gpu scheduling. In *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP 23)*, pages 595–610, 2023.

[59] Mohit Pandey, Michael Fernandez, Francesco Gentile, Olexandr Isayev, Alexander Tropsha, Abraham C Stern, and Artem Cherkasov. The transformational role of gpu computing and deep learning in drug discovery. *Nature Machine Intelligence*, 4(3):211–221, 2022.

[60] Bo Peng, Haozhong Zhang, Jianguo Yao, Yaozu Dong, Yu Xu, and Haibing Guan. {MDev-NVMe}: A {NVMe} storage virtualization solution with mediated {Pass-Through}. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 665–676, 2018.

[61] Vijay Janapa Reddi, Christine Cheng, David Kanter, Peter Mattson, Guenther Schmuelling, Carole-Jean Wu, Brian Anderson, Maximilien Breughe, Mark Charlebois, William Chou, et al. Mlperf inference benchmark. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 446–459. IEEE, 2020.

[62] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4510–4520, 2018.

[63] Haichen Shen, Lequn Chen, Yuchen Jin, Liangyu Zhao, Bingyu Kong, Matthai Philipose, Arvind Krishnamurthy, and Ravi Sundaram. Nexus: A gpu cluster engine for accelerating dnn-based video analysis. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 322–337, 2019.

[64] Lin Shi, Hao Chen, Jianhua Sun, and Kenli Li. vcuda: Gpu-accelerated high-performance computing in virtual machines. *IEEE Transactions on Computers*, 61(6):804–816, 2011.

[65] Jike Song, Zhiyuan Lv, and Kevin Tian. Kvmgt: A full gpu virtualization solution. In *KVM Forum*, volume 2014, 2014.

[66] Foteini Strati, Xianzhe Ma, and Ana Klimovic. Orion: Interference-aware, fine-grained gpu sharing for ml applications. In *Proceedings of the Nineteenth European Conference on Computer Systems (EuroSys 24)*, pages 1075–1092, 2024.

[67] Yusuke Suzuki, Shinpei Kato, Hiroshi Yamada, and Kenji Kono. {GPUvm}: Why not virtualizing {GPUs} at the hypervisor? In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 109–120, 2014.

[68] Kun Tian, Yaozu Dong, and David Cowperthwaite. A full {GPU} virtualization solution with mediated {Pass-Through}. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 121–132, 2014.

[69] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[70] John Paul Walters, Andrew J Younge, Dong In Kang, Ke Thia Yao, Mikyung Kang, Stephen P Crago, and Geoffrey C Fox. Gpu passthrough performance: A comparison of kvm, xen, vmware esxi, and lxc for cuda and opencl applications. In *2014 IEEE 7th international conference on cloud computing*, pages 636–643. IEEE, 2014.

[71] Guanhua Wang, Kehan Wang, Kenan Jiang, Xiangjun Li, and Ion Stoica. Wavelet: Efficient dnn training with tick-tock scheduling. *Proceedings of Machine Learning and Systems (MLSys 21)*, 3:696–710, 2021.

[72] Bingyang Wu, Zili Zhang, Zhihao Bai, Xuanzhe Liu, and Xin Jin. Transparent {GPU} sharing in container clouds for deep learning workloads. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 69–85, 2023.

[73] Hao Wu, Wei Liu, Yifan Gong, and Jiangming Jin. Safe process quitting for gpu multi-process service (mps). In *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*, pages 1169–1170. IEEE, 2020.

[74] Wencong Xiao, Shiru Ren, Yong Li, Yang Zhang, Pengyang Hou, Zhi Li, Yihui Feng, Wei Lin, and Yangqing Jia. {AntMan}: Dynamic scaling on {GPU} clusters for deep learning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 533–548, 2020.

[75] Chao-Tung Yang, Jung-Chun Liu, Hsien-Yi Wang, and Ching-Hsien Hsu. Implementation of gpu virtualization using pci pass-through mechanism. *The Journal of Supercomputing*, 68:183–213, 2014.

[76] Ting-An Yeh, Hung-Hsin Chen, and Jerry Chou. Kubeshare: A framework to manage gpus as first-class and shared resources in container cloud. In *Proceedings of the 29th international symposium on high-performance parallel and distributed computing*, pages 173–184, 2020.

[77] Hangchen Yu and Christopher J Rossbach. Full virtualization for gpus reconsidered. In *Proceedings of the Annual Workshop on Duplicating, Deconstructing, and Debunking*, 2017.

[78] Shulai Zhang, Quan Chen, Weihao Cui, Han Zhao, Chunyu Xue, Zhen Zheng, Wei Lin, and Minyi Guo. Improving gpu sharing performance through adaptive bubbleless spatial-temporal sharing. In *Proceedings of the Twentieth European Conference on Computer Systems*, pages 573–588, 2025.

[79] Wei Zhang, Binghao Chen, Zhenhua Han, Quan Chen, Peng Cheng, Fan Yang, Ran Shu, Yuqing Yang, and Minyi Guo. {PilotFish}: Harvesting free cycles of cloud gaming with deep learning training. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 217–232, 2022.

[80] Wei Zhang, Weihao Cui, Kaihua Fu, Quan Chen, Daniel Edward Mawhirter, Bo Wu, Chao Li, and Minyi Guo. Laius: Towards latency awareness and improved utilization of spatial multitasking accelerators in datacenters. In *Proceedings of the ACM international conference on supercomputing*, pages 58–68, 2019.

[81] Yihao Zhao, Xin Liu, Shufan Liu, Xiang Li, Yibo Zhu, Gang Huang, Xuanzhe Liu, and Xin Jin. Muxflow: Efficient and safe gpu sharing in large-scale production deep learning clusters. *arXiv preprint arXiv:2303.13803*, 2023.

AnchorNet: Bridging Live and Collaborative Streaming with a Unified Architecture

Deployed Systems Track

Tong Meng Wei Zhang* Dong Chen Zhen Wang Quanqing Li Changqing Yan
Wei Yang Chao Yuan Le Zhang Jianxin Kuang Jianlin Xu
ByteDance

Abstract

Collaborative streaming has emerged as a popular mode in modern live streaming applications. In the meanwhile of improving interactivity between broadcasters and viewers, it requires the live streaming architecture to smoothly switch between two streaming modes (*e.g.*, traditional live streaming with a single broadcaster and collaborative streaming with at least two collaborative broadcasters). In this paper, we present AnchorNet, the new live streaming architecture for one of the most popular streaming applications. The core of AnchorNet is a unified stream path from the broadcaster to the viewer, enabling the host broadcaster of a live channel to switch between streaming modes within a continuous application session. It also proposes audio stream splicing techniques to further minimize unpleasant audio glitches during streaming mode switching. Practical deployment shows that AnchorNet can significantly reduce rebuffering during mode switching by over 60%, and increase user engagement by up to 3.83%.

1 Introduction

Thanks to advanced social media platforms (*e.g.*, Instagram, TikTok, Twitch, *etc.*), live streaming is no longer exclusive to large events and professional media production, and has become easily accessible to individual broadcasters with their own camera and microphone feeds [1]. Furthermore, driven by user needs for enhanced interactivity and engagement, collaborative streaming has been an integrated feature on many platforms (*e.g.*, [2–4]). A viewer has the opportunity to share personal stories with other audience when invited by host of a live channel as a collaborative broadcaster (co-broadcaster). Multiple hosts can audio/video chat with each other (*e.g.*, online karaoke) while streaming together to their perspective channels.

Compared with traditional single-broadcaster live streaming[1], collaborative streaming presents media contents from multiple co-broadcasters, and induces several new challenges to the live streaming architecture.

- It introduces the task of stream mixing. The live streaming architecture needs to determine where to mix media streams of co-broadcasters before they are rendered at viewers. There is no optimal choice among the three options: on broadcaster side, on server side, or on viewer side. Each of them faces tradeoffs between user experience and cost.
- It requires the WebRTC stack to fulfill more stringent latency requirements between co-broadcasters. Considering that many major applications still heavily rely on RTMP and HTTP-based ingest and distribution for the basic live streaming mode [5–7], a possibly additional protocol stack increases the implementation complexity.
- A live channel should be able to smoothly switch between the two streaming modes. As the host broadcaster interacts with different viewers and friends, viewers should perceive as little choppy audio or frozen video as possible.

As one of the primary social media platforms, TikTok adopts server-side stream mixing to serve over 1 billion users. Such a design choice aims to avoid setting barriers on user device performance, and to mitigate server-side burdens from serving massive stream subscriptions.

Our first-generation live streaming architecture, DualNet, as its name suggests, is based on decoupled implementations for live and collaborative streaming. Of the two modes, we first launched live streaming. Therein, a broadcaster directly publishes an RTMP stream to CDN for cost-effective distribution to viewers. Like many live streaming platforms [8, 9], AAC audio codec [10] is used end-to-end. Then, when extending to collaborative streaming several years later, we leveraged RTC media servers to exchange RTP streams between co-broadcasters. Similar to many RTC applications [11–14], co-broadcasters change to Opus audio codec [15], because it has superior encoding latency, compression ratio, and error resilience. Meanwhile, to be compatible with the original live streaming mode, RTP streams from a live channel are mixed and transcoded to an AAC-encoded RTMP stream on the server side as the ingest to CDN for distribution.

DualNet resulted from our engineering decision not to alter basic live streaming implementation while additively rolling out collaborative streaming. However, it makes switching between the two streaming modes quite complicated. Due to its decoupled form, viewers effectively subscribe to two different streams when a channel switches between live and collaborative streaming, which follow different publishing and ingest paths to CDN. The two modes also require different transport protocols and audio codecs to be used by broadcasters. Consequently, audio and video rebuffering tends to occur during mode switching. What is worse, optimizations atop

*Wei Zhang is the corresponding author: zhangwei.666@bytedance.com

[1] Henceforth, when referring to mode of live streaming, we directly use live streaming to represent the scenario with a single broadcaster.

such a decoupled architecture usually involve complicated coordination between multiple components, each corresponding to a separate internal code repository. That adds to both development and maintenance overheads.

Therefore, as our user traffic continues to expand, we seek to evolve TikTok's live streaming architecture, and build AnchorNet in order to toggle the streaming mode more smoothly. While doing so, we retain the design choice of server-side stream mixing in collaborative streaming. Lessons learned from DualNet further yield three objectives.

- We prefer to preserve an identical publishing path to CDN in both streaming modes. Each live channel only ingests to a single CDN node, eliminating origin re-addressing during a host broadcaster session. That facilitates in-order delivery of packets belonging to different modes.
- The broadcaster may maintain the same application session throughout the streaming process. Otherwise, upon entering a new streaming mode, the switchover delay from ramping up a separate congestion controller increases the risk of rebuffering. That requires the broadcaster to rely on the same set of application and transport protocols.
- No valid audio signal is lost, and audio glitch is minimized even if host broadcaster of a channel changes audio codec along with streaming mode. That is not possible with the broadcaster simply switching the codec, because of encoder delay [16] inherent to most audio encoding algorithms.

AnchorNet unifies the architecture for basic live streaming and collaborative streaming, and meets all the above objectives. We do not claim it to be the optimal solution in all cases. Yet it provides insights on the feasibility of large-scale collaborative streaming deployment based on server-side stream mixing. To the best of our knowledge, we are the first to share practical experiences on extending live streaming architecture to support collaborative streaming. Our main contributions are highlighted as follows.

- The AnchorNet architecture uses RTC servers as the bridge between broadcasters and CDN, so that the channel host publishes to the same CDN node in an uninterrupted session when the streaming mode switches repeatedly.
- We propose audio sample-level operation techniques to resolve inherent issues of audio codecs such as encoder delay. Through coordination between broadcaster and RTC server, the host broadcaster's audio samples in different streaming modes can be seamlessly spliced, *e.g.*, no silence is inserted and no valid audio signal is discarded.
- We evaluate the performance of AnchorNet in large-scale practical deployment. Compared with DualNet, it significantly improves the smoothness of user experience, *e.g.*, the rebuffering time during streaming mode switching is reduced by over 60% and up to 78.9%. That also contributes to a 2.15% and 3.44% increase in per co-broadcaster and per viewer engagement, respectively.

This work does not raise any ethical issues.

2 Design Choice on Where to Mix Stream

In fact, we have implemented all three options on stream mixing aimed at various application scenarios. For example, co-broadcasters fetch and render each other's streams independently, effectively leveraging viewer-side stream mixing. We start with a comprehensive analysis of the rationale behind the foremost design choice in both generations of our live streaming architecture: server-side stream mixing before delivering to viewers in collaborative streaming.

2.1 Server Side *vs.* Broadcaster Side

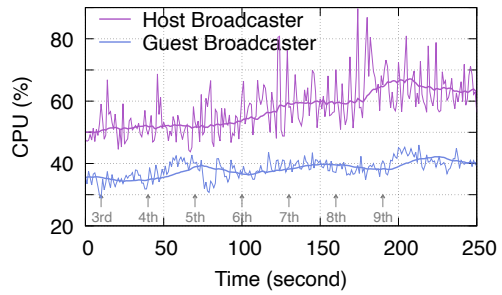
With broadcaster-side stream mixing, the host broadcaster of a live channel uploads at least two streams in collaborative streaming: one personal stream fed to the other co-broadcasters, the other mixed stream for viewers. It offers several advantages over server-side stream mixing.

- The server side only has two roles: 1) RTC SFU (Selective Forwarding Unit) that forwards streams between co-broadcasters, and 2) CDN for distribution to viewers. No stream mixing is needed, saving cost on server machines.
- The host broadcaster controls timing and smoothness of switching between streaming modes at the source. The above two server-side roles remain largely independent. So does the development of live and collaborative streaming.

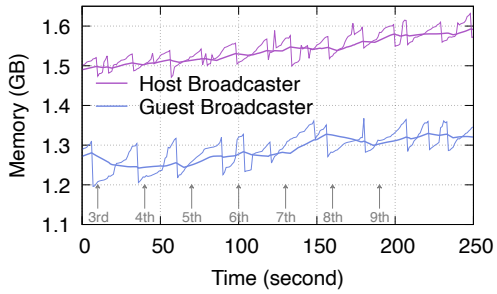
However, we assess that its issues outweigh the advantage.

- Broadcaster-side stream mixing increases computation overhead and battery power consumption on user device. The maximum number of co-broadcasters may thus be limited by the performance of user device. We do not want to discourage potential users, *e.g.*, those with relatively moderate smartphones. Nor do we like such a barrier when we extend to more complex streaming scenes in the future.
- The broadcaster will have to maintain two uplink connections in collaborative streaming. In case of restrictive first-mile bandwidth, the bitrate received by viewers will approximately be halved compared to that in live streaming mode. Besides, the congestion controllers of the two connections compete with each other, impacting stability of both.
- Compared with the server-side option, the stream mixing task is delayed by an one-way latency at the access link, while waiting for the host broadcaster to gather the streams of other co-broadcasters from the connected SFU. The tail latency over 100 milliseconds on the rapidly fluctuating access link [17, 18] makes viewer experience more fragile.

To show its overhead more intuitively, we evaluate a broadcaster-side stream mixing prototype. When we increase the number of co-broadcasters from 2 to 9, the trends of CPU and memory usage on the host and the first guest co-broadcaster are shown in Figure 1. Taking the difference of their CPU usage at the start of the trial, the stream mixing task needs about 15% CPU time at the host broadcaster. Having 7 more co-broadcasters consumes 15% CPU time on the



(a) CPU usage



(b) Memory usage

Figure 1: Overhead of broadcaster-side stream mixing: measurements start with a host broadcaster and a 2nd co-broadcaster, with the join time of subsequent co-broadcasters explicitly marked (30-second average CPU/memory usage shown in thicker lines)

host's device, and less than 5% on the guest co-broadcaster. In the end of the trial, the host broadcaster consumes an additional 25% CPU time on average compared with the guest co-broadcaster, and the temporary CPU usage even reaches 90%. Meanwhile, the host needs more than 250 MB memory in extra. All co-broadcasters (including the host) use Xiaomi 14 smartphones in our test. The 25% CPU time fully occupies 2 out of the 8 CPU cores. That would exceed the capacity of some low-price smartphones released several years ago.

In addition, there can easily be at least an order of magnitude more viewers than co-broadcasters per channel. Therefore, server-side cost of stream mixing may be amortized by serving a large number of viewers, generating positive ROI (Return on Investment) given our user population.

2.2 Server Side *vs.* Viewer Side

The so-called viewer-side stream mixing actually does not involve the task of mixing and transcoding co-broadcasters' streams. Instead, each viewer directly renders multiple streams after decoding them. Thus, its most important advantages are shorter end-to-end delay and reduced server-side computation. By fetching independent co-broadcasters' streams to viewers, it also has the benefit of a customizable user interface. Its issues are listed below.

- The media streams of co-broadcasters can be distributed to viewers via SFU or CDN. An SFU-based realization effectively implements a live channel as a multi-party conference room [19, 20]. It requires us to deploy dramatically more RTC servers to accommodate all the viewers, and signal the connection and disconnection of each co-broadcaster to orders of magnitude more users. So it is not as viable as CDN-based distribution, where we can leverage plenty of 3rd-party CDN providers to lower our expenses.
- In collaborative streaming, a viewer needs to subscribe to the same number of streams as co-broadcasters. Even if we ignore the complexity of signaling those streams to all the viewers, multiple per-co-broadcaster streams tend to lower compression efficiency [21] and consume more bandwidth than a mixed stream. That pulls up bandwidth cost on both server side and viewer side.

We do not list the issue of requirements on viewer device performance. Because unlike the broadcaster and server-side options, viewer does not need to re-encode and forward the received streams. As demonstrated by CPU and memory consumption of the guest co-broadcaster in Figure 1, receiving, decoding and rendering streams of several co-broadcasters only mildly raises the pressure on device performance.

Again, there is usually a disparity in the order of magnitude between the number of viewers and co-broadcasters. It is not uncommon for a live channel to have thousands of concurrent viewers. At that scale, the increased bandwidth cost by viewer-side stream mixing may outweigh its savings of computation overhead compared with server-side stream mixing.

We should note that our considerations on where to mix streams represent one perspective among many. We notice some platforms make the same choice of server-side stream mixing (*e.g.*, Tecent TRTC [22], Alibaba ApsaraVideo RTC [23]), but developers may find client-side options more suitable provided different deployment focus.

3 Background

3.1 Decoupled DualNet Architecture

Our first-generation live streaming architecture is depicted in Figure 2. We strengthen several key points here.

RTC server roles. A proportion of server instances are used as RTC servers. They may be assigned two roles, including SFU and MCU (Multipoint Control Unit) media servers. Similar to a typical multi-party conferencing setup, cascaded SFUs forward RTP streams between co-broadcasters without decoding them [20]. Each live channel further selects an MCU to gather co-broadcasters' RTP streams from SFUs, where compute-intensive tasks such as steam mixing are accomplished. The MCU server also ingests the mixed stream to CDN in collaborative streaming.

Broadcaster codecs. In our DualNet implementation, the audio codec used by broadcasters is statically determined by the streaming mode, *i.e.*, AAC in live streaming and Opus in collaborative streaming. The video codec should otherwise

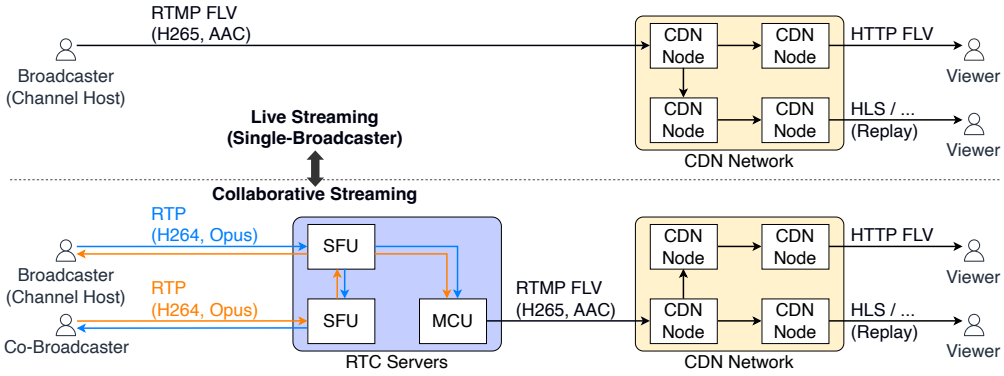


Figure 2: DualNet live streaming architecture

adapt to capabilities on various devices. As exemplified in Figure 2, the host broadcaster of a channel may negotiate a different video codec with invited co-broadcasters (H264) from the one already used by ongoing live streaming (H265).

Consistent codecs to viewers. From the perspective of CDN, the output audio and video codecs of a channel remain unchanged throughout each broadcast. If a host broadcaster changes codecs upon entering collaborative streaming, the RTC MCU media server should conduct transcoding while mixing streams (H264/Opus to H265/AAC in Figure 2) and ensure codec consistency. Then, when the channel returns to live streaming mode, its host broadcaster may also switch back to original codecs. Without changing codecs, broadcasters can always adapt some detailed configurations, *e.g.*, they can use larger GOP (Group of Pictures) and enable B-frames in reaction to decreased access bandwidth.

Distribution Protocols. Technically, any protocol supported by CDN can be used for distribution to viewers, *e.g.*, RTP if leveraging recent CDN feature of low-latency streaming [24]. We use HTTP FLV to ensure backward compatibility, because it has served our live streaming viewers for years at the time we rolled out collaborative streaming. In the meanwhile, CDN may record and store the live streams in other formats (*e.g.*, HLS). The host broadcaster of a channel can make some history live clips available for on-demand replay afterwards.

3.2 Issues of DualNet

To motivate architecture redesign, we elaborate on critical issues of DualNet during streaming mode switching.

CDN ingestion node change. In live streaming mode, the broadcaster queries an edge CDN node through DNS. When it comes to collaborative streaming, our logically centralized scheduling backend allocates an ingestion node in CDN based on proximity of the MCU server. Although the allocation refers to the edge node assigned in live streaming, it may still have to select a different node. Below are two example scenarios where that happens.

- There are CDN PoPs (Points of Presence) but no RTC servers in the country of the broadcaster. Detouring to the same CDN ingestion node as in live streaming mode induces additional transport delay.
- The MCU server and the original edge CDN node belong to two poorly peered ISPs, and the path between them suffers from high packet loss or fluctuating RTT.

Thus, CDN may have to splice two RTMP streams ingested from different locations into a single stream for distribution. That increases engineering workloads and risk of rebuffering.

Different publishing paths. Even if a channel always ingests to the same CDN node, the two streaming modes still have separate publishing paths from the broadcaster to the ingestion node. In particular, the publishing path in collaborative streaming involves multiple transport connections, running over two transport protocols. There is not necessarily a straightforward way to coordinate and migrate states between several independent congestion controllers, not to mention the delay from establishing a new connection at least during the first streaming mode switching. In the worst case, both the RTP streams from co-broadcasters and the mixed stream by MCU have to go through slow start upon entering collaborative streaming, leading to degraded bitrate and even rebuffering. Additionally, inconsistent delay on the two publishing paths is likely to cause out-of-order arrival of media frames, *e.g.*, the first collaborative streaming frame may arrive at the ingestion node earlier than the last live streaming frame. That adds to the complexity of session switchover.

Implementation ossification. Ironically, DualNet's decoupled implementations for the two streaming modes leave us with multiple tightly coupled components, complicating optimization of user experience during streaming mode switching. For instance, to adapt to updated interfaces of a 3rd-party CDN provider, both our broadcaster-side and RTC server SDKs must be modified. Moreover, the application logic at the broadcaster calls separate SDKs for live streaming and collaborative streaming. In large-scale systems, that often results in ossification. For more efficient iteration of new features and optimizations, we are devoted to reduce inter-component coupling by a unified live streaming architecture.

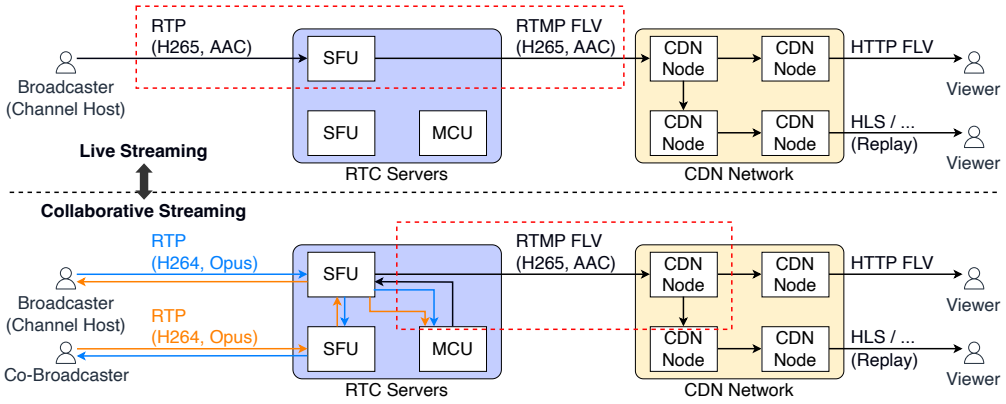


Figure 3: AnchorNet live streaming architecture (Key differences from DualNet marked in red dashed boxes)

4 AnchorNet Architecture

We name the new live streaming architecture AnchorNet (Figure 3). Drawing lessons learned from the first-generation architecture, it follows one most important principle, *i.e.*, to unify the publishing path from the broadcaster to CDN. For that purpose, RTC SFU is introduced into the publishing path as the "anchor" in both streaming modes. That not only alleviates the challenges of aligning two streams from separate paths, but also helps mitigate coupling between components.

In this section, we walk through key details of AnchorNet, focusing on our underlying considerations.

4.1 Same Publisher to CDN

A unified publishing path should have the same publisher to CDN in both streaming modes. Provided server-side stream mixing in collaborative streaming, if the channel host is the publisher, the mixed stream will have to be forwarded back to the broadcaster before being published to CDN. That shares many drawbacks of broadcaster-side stream mixing (§2.1). Thus, the RTC server should act as the publisher to CDN.

Then, among the two roles of RTC server, MCU is not involved in live streaming mode. According to our current traffic data, live streaming still accounts for a much higher volume than collaborative streaming. Therefore, we assign the CDN publisher role to SFU media server. As shown in Figure 3, an MCU server paired with a channel feeds the transcoded mixed stream back to the SFU serving the host broadcaster, instead of directly ingesting to CDN as in DualNet. However, the round trip between SFU and MCU induces additional delay to collaborative streaming, especially considering that MCU servers are mostly centrally deployed now. We anticipate that the additional delay may decrease with increasing deployment of edge computing servers [25–28]. Yet at the moment, we rely on the jitter buffers within CDN and at the viewer to prevent such a round trip from inducing rebuffering when switching from live to collaborative streaming.

We also note that when switching between streaming modes, SFU should guarantee that the ingestion node in CDN sees consecutive sequence numbers.

4.2 Protocols for CDN Ingest

With an RTC SFU media server sitting between the broadcaster and CDN on the publishing path, there are two scenarios where a protocol needs to be selected: 1) the protocol used by the broadcaster to publish to SFU, and 2) that used by the SFU to ingest to CDN.

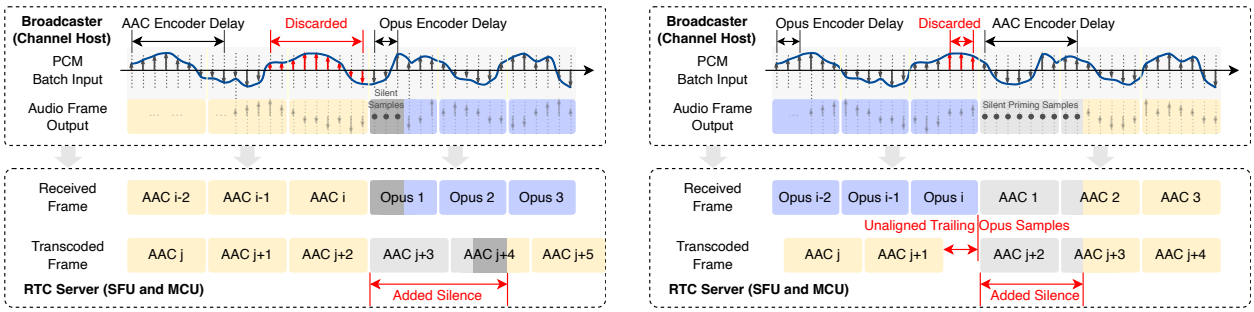
First, given that co-broadcasters exchange streams using RTP, we escalate RTP to be the unified uplink protocol for broadcasters across the two streaming modes. By doing so, a channel host can broadcast a continuous RTP stream in the same session while switching between modes, without the need to establish another transport connection. A side benefit is that the live streaming mode can leverage self-developed and optimized congestion control algorithms in user-space, instead of limited to those less frequently updated algorithms supported by kernel on user devices.

As for the ingest protocol from SFU to CDN, an important objective is to ensure compatibility with most CDN providers. Considering that RTMP-based ingest has been supported by almost all CDN providers [29–32] for years and Internet paths between RTC and CDN servers tend to be more stable than first/last-mile access links, SFU continues to use RTMP to push media streams to CDN. A benefit of using RTMP as the ingest protocol is that it requires minimal modification to our previous implementation on CDN and viewer side.

4.3 Selection of Audio Codecs

The principal differences between AnchorNet and DualNet are explained above in §4.1&4.2. On that basis, we stick with the selection of audio codecs in DualNet. On the broadcaster side, Opus is used for collaborative streaming, and AAC is used for live streaming. On the server side, AAC is used for CDN ingest. Correspondingly, MCU needs to transcode and mix Opus audio streams of all co-broadcasters of a channel into an AAC stream in collaborative streaming.

We do not adopt an all-Opus scheme, mainly because the AAC codec is still much more widely compatible with RTMP,



(a) Live streaming to collaborative streaming

(b) Collaborative streaming to live streaming

Figure 4: Problems of switching streaming mode without coordination (Other guest co-broadcasters' streams are omitted from received and transcoded frames. Also, low-level details such as overlapped MDCT transform window are not shown.)

albeit some explorations into extending RTMP to support contemporary codecs including Opus [33]. Considering that, we did not unify the host-side audio codec to Opus, either. That would extend the expense of MCU transcoding from collaborative streaming mode only to both streaming modes. It could lead to a significant server-side cost increment, because the duration of live streaming is generally longer than collaborative streaming in our production environment. In the long term, we would certainly like to have a single audio codec in most cases, regardless of the streaming mode.

4.4 Mitigated Ossification

AnchorNet does not eliminate coupling components totally. As will be explained in §5, the broadcaster and the RTC servers still need to coordinate in audio stream splicing. However, it mitigates implementation ossification, indeed. The broadcaster logic always runs on RTC SDK now, and does not need to maintain two sets of transport protocols and congestion control algorithms depending on the streaming mode. More importantly, with RTC SFU as the only CDN publisher, the broadcaster does not directly interact with CDN any more. That simplifies many tasks such as scheduling among multiple CDNs and debugging between server-side and broadcaster-side issues. To be specific, we contract with different 3rd-party CDN providers for enhanced fault tolerance. When a CDN cluster of a provider encounters connection issues in the ingest direction, both fault localization and temporary blacklist update happen on server side in AnchorNet. No patches to user-facing modules are needed as with DualNet.

5 Audio Stream Splicing

In AnchorNet, the RTC SFU needs to splice the audio/video stream from the host broadcaster (live streaming) and the mixed stream from the MCU server (collaborative streaming) when the streaming mode is toggled. For convenience, we refer to the streaming mode before and after a mode change as the *egress* and *ingress streaming mode*, respectively. The codec used by the host broadcaster is called accordingly as the *egress* and *ingress codec*, as well.

For video streams, the host broadcaster always switches the streaming mode at the frame boundary. It suffices to comply with two packet forwarding paths (*i.e.*, with and without MCU) depending on the streaming mode. The last frame of the egress streaming mode can directly be followed by the first frame of the ingress streaming mode. In comparison, as illustrated in Figure 4, smooth audio stream splicing is by no means as straightforward as that, owing to challenges inherent to the audio encoding algorithms.

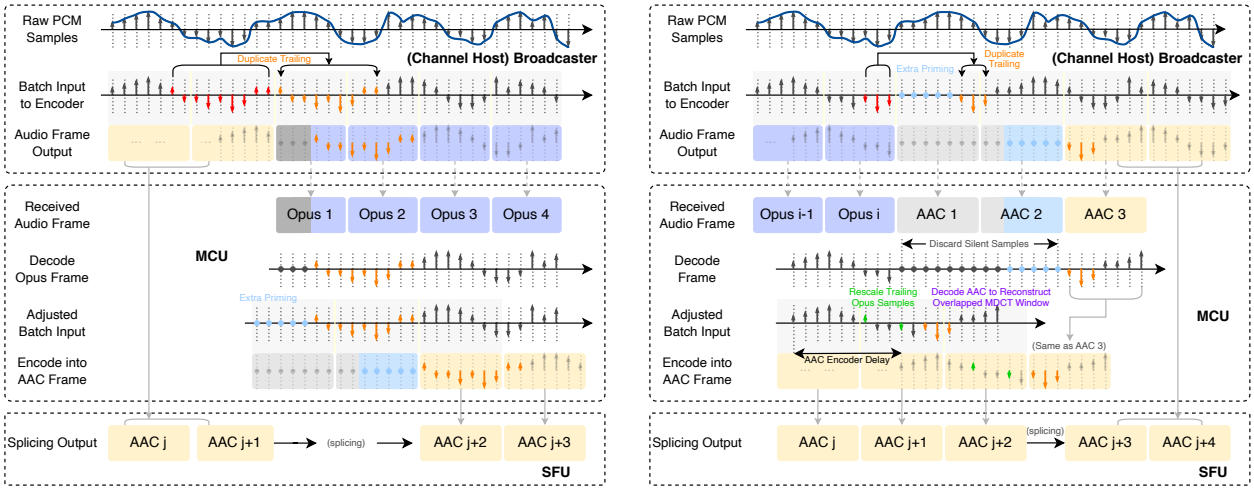
5.1 Challenges from Audio Codec

5.1.1 Encoder Delay

At a high level, each fixed-duration input batch of PCM (pulse-code modulation) samples generates an audio frame that lasts for the same duration. However, if we look at the serialized output sample sequence by the decoder, it generally starts with several silent samples, leading to an encoder delay between the input and output indices of the same raw sample. One of the most important reasons for such an encoder delay is the overlapped window while conducting MDCT (modified discrete cosine transform) [15, 16, 34], which is adopted universally by many encoding algorithms. Briefly, when transforming time-domain samples represented by an audio frame to frequency domain, the transform window is overlapped with the subsequent frame to avoid audio artifacts at window boundaries. To properly output an audio frame for the first input batch (*e.g.*, with insufficient samples on its own to fill an entire transform window), an encoder conducts priming [34], by inserting silent samples before the first raw input sample.

If the host broadcaster were to rigidly switch between AAC and Opus and the MCU server did not process the received PCM samples any further before transcoding, the encoder delay would cause two issues in Figure 4. First, the host broadcaster starts every ingress streaming mode with a leading silence, resulting in perceivable audio glitch. The silent samples are introduced:

1. by ingress codec on host broadcaster's user device, and
2. by MCU at the start of transcoding upon entering collaborative streaming.



(a) Live streaming to Collaborative streaming

(b) Collaborative streaming to Live streaming

Figure 5: Audio stream splicing when switching between the two streaming modes (In collaborative streaming, stream mixing happens in the step marked *Encode into AAC Frame*. The streams of other guest co-broadcasters, as well as other details such as overlapped MDCT window and the SFU between host broadcaster and MCU, are not shown for brevity.)

Codec (Profile)	Frame Size	Priming Samples
AAC (HE-AAC v1 [35])	2048	5058
Opus (48 kHz [36])	960	325

Table 1: A Typical Codec Configuration in TikTok

Moreover, as exemplified in Table 1, the number of priming samples may not be an integer multiple of audio frame size. Simply dropping audio frames containing such samples is very likely to lose valid audio signals.

Second, tail samples of the egress codec, which are still un-encoded in the encoding buffer when the host broadcaster triggers a streaming mode switch, will be discarded. The number of discarded samples equals the number of inserted leading silent samples, and may not be divisible by the frame size. So even if most codec libraries support flushing remaining tail samples, that induces padded silence in the last frame and is not an ideal solution, either.

5.1.2 Inconsistent Frame Sizes

Another challenge in stream splicing is the inconsistent frame sizes between AAC and Opus (*e.g.*, 2048 *vs.* 960 as in Table 1). When exiting collaborative streaming, the trailing samples decoded from Opus frames may not fill an entire AAC frame (Figure 4b). Without any optimization, the MCU server either discards or flushes those samples, with both choices generating similar glitches in the host broadcaster's audio due to encoder delay (§5.1.1).

5.2 Splicing Techniques

Broadcaster-server coordination is necessary to mitigate potential audio glitches due to encoder delay and inconsistent frame sizes. Four PCM sample-level optimization techniques are proposed here. They operate on the audio stream of the host broadcaster, before it is mixed with other co-broadcasters' audio streams. Figure 5 shows when and where they take effect in the whole stream splicing process.

5.2.1 Extra Priming to Avoid Partial Silent Frames

If the encoder delay is a multiple of frame size, the priming samples will be encoded into different frames from the raw input samples. Then, the server side can safely skip leading silent frames in the ingress streaming mode without losing valid audio signals. We thus supplement N_{extra} extra priming samples to the transcoding or ingress codec, aiming at:

$$(N_{priming} + N_{extra}) \mod frame_size = 0.$$

Take the setup in Table 1 as an example. When entering collaborative streaming, MCU precedes the decoded Opus samples with an additional 761 silent samples, forming an extra priming of 1086 samples (including 325 Opus priming samples already inserted by host broadcaster). Similarly, when switching back to live streaming, the host broadcaster adds the same extra priming of 1086 samples to its AAC codec. In both cases, there will be 3 leading silent AAC frames, enabling smooth stream splicing starting from the 4th AAC frame.

Extra priming, as well as the other techniques, require the knowledge on encoder delay and frame size. We obtain and store their value during codec initialization.

5.2.2 Duplicate Trailing Samples at Host Broadcaster

To avoid tail samples of the egress codec from being discarded, the host broadcaster duplicates those samples into the ingress codec. That is achieved by maintaining the most recent audio samples fed to the currently used codec in a trailing sample ring buffer. The buffer is bound to the codec, and

its size equals the codec's encoder delay (*e.g.*, length of 325 samples for Opus if following the setup in Table 1). When the streaming mode changes, the ring buffer of the egress codec will be freed once sample duplication is complete.

Importantly, MCU does not need to be aware of trailing sample duplication. It simply treats the duplicated samples as raw audio samples.

5.2.3 Rescale Trailing Opus Samples at MCU

Algorithm 1: Transcoding and Sample Rescaling

```
1  N = 0 ;
2  Initialize buffer Q with N_extra^AAC extra priming samples ;
3  while streaming mode is collaborative do
4      s_pcm = Decode(next Opus frame) ;
5      remove_priming(s_pcm) ;      // 1st frame only
6      Q.push(s_pcm) ;
7      N = N + |s_pcm| ;
8      while N > 1.5 · l_AAC do  // l_AAC: frame size
9          Q.pop(l_AAC) ;   // batch input to codec
10         N = N − l_AAC ;
11 Rescale remaining raw samples to l_AAC samples ;
12 Q.pop(l_AAC) ;  // N_extra^AAC samples remains in Q
```

Extra priming guarantees that the first raw audio sample in collaborative streaming is the first frame in a transcoded frame. On that basis, as long as the number of raw samples (excluding extra priming) fed for transcoding is a multiple of AAC frame size, there will be no unaligned trailing samples decoded from Opus frames. Since the raw Opus samples are not controlled by the MCU, it needs to rescale the trailing samples to enforce the condition. Algorithm 1 outlines its behaviors in collaborative streaming mode.

Upon entering collaborative streaming, the MCU server initializes a pre-encode buffer with N_{extra}^{AAC} extra priming samples required by its AAC codec. It stores raw samples decoded from Opus frames (excluding priming samples in the first frame) in the buffer. Whenever there are more than 1.5 times the frame size (l_{AAC}) of raw samples in the buffer, it pops a batch as the input to AAC codec. Hence, in the end of collaborative streaming, the number of raw Opus samples in the buffer lies between $0.5 \cdot l_{AAC}$ and $1.5 \cdot l_{AAC}$. The MCU up-samples or down-samples them to exactly l_{AAC} samples, and feed one last input batch to AAC codec. Such rescaling inevitably deviates from the original timescale of input samples. Algorithm 1 keeps the extent of deviation below half an AAC frame (*e.g.*, around 20 ms under frame size of 2048 and 48 kHz sampling rate), and induces only minimal negative influence on the output audio effect.

5.2.4 Reconstructing Overlapped MDCT Window

The pre-encode buffer is initialized with extra priming samples, but trailing sample rescaling only aligns raw samples to an integer multiple of frame size. Therefore, after the last batch of input to AAC codec in Algorithm 1 (line 12), there will be N_{extra}^{AAC} samples left in the buffer, equal to extra priming inserted in the beginning of collaborative streaming (line 2). At the same time, there are un-encoded samples by the codec due to encoder delay. To finish encoding all decoded Opus samples, a simple method is to directly feed the remaining rescaled samples to the encoder and flush the codec. Silent samples will be padded automatically, and rescaling ensures that the last rescaled sample is the last sample represented by the final transcoded AAC frame. However, such a frame will not be encoded based on an MDCT window overlapped with the first non-priming live streaming frame, which should be the next frame after stream splicing.

To avoid audio artifacts therein, we pad raw audio samples from the first non-priming live streaming frame to the pre-encode buffer before feeding remaining samples to AAC codec. That reconstructs overlapped window for the final transcoded AAC frame, so that it will splice into the live streaming stream more smoothly. For that purpose, MCU should receive and decode the first several live streaming frames from SFU (Figure 5b).

We note that the MCU does not need to reconstruct MDCT window when switching from live to collaborative streaming. With duplicate trailing samples (§5.2.2), the last live streaming AAC frame is guaranteed to have a transform window overlapped with the first collaborative streaming frame.

5.3 Scope of Application

Discussion in this section so far is based on selection of audio codecs as explained in §4.3. In fact, the above challenges and optimization techniques are universally applicable to broadcaster-side and server-side stream mixing, regardless of the audio codecs used by the broadcaster and MCU.

Specifically, both broadcaster-side and server-side stream mixing require decoding co-broadcasters' audio streams to mix their PCM samples. The re-encoding process following that is subject to influence of encoder delay and overlapped MDCT window. For instance, leading silence is always generated upon entering collaborative streaming. Thus, even in the extreme case where the same audio codec and codec profile are used in both streaming modes, extra priming and reconstructed MDCT window are still indispensable for seamless audio stream splicing. Trailing sample duplication and rescaling would also be necessary if different profiles of the same codec are used for live and collaborative streaming.

6 Micro-Benchmark

Before practical deployment, we verify the effectiveness of AnchorNet in optimizing user experience during streaming mode switching with small-scale experiments. We compare it with DualNet, as well as two other popular live streaming applications. The two competitors represent two design choices on streaming architecture. According to packet dump analysis, Competitor-1 adopts viewer-side stream mixing, and imple-

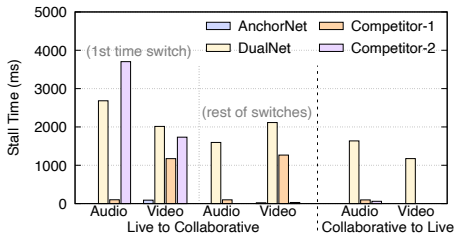


Figure 6: Stall per streaming mode switch

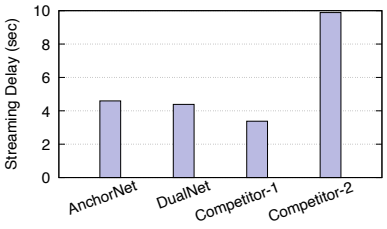


Figure 7: Live streaming E2E delay

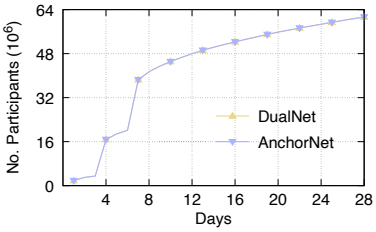


Figure 8: Num. of users in A/B test

ments both live and collaborative streaming with a unified RTC stack. Competitor-2 is based on server-side stream mixing, and similar to DualNet, requires the host broadcaster to establish separate connections for the two streaming modes.

Because the maximum number of co-broadcasters allowed varies across different applications, we test the most basic scene with only two co-broadcasters (*i.e.*, host plus an invited guest). We use three Xiaomi 14 smartphones as channel host, guest co-broadcaster, and viewer, respectively. In each trial, the host starts in live streaming and triggers mode switching 6 times by repeatedly inviting and hanging up on the guest co-broadcaster, while the viewer stays in the channel. We place the three devices at the same location and connect them to the same dedicated office WiFi, so that the evaluation reflects native performance of the architecture (*e.g.*, without influence of weak access network, overlay path computation policy, and bitrate adaptation strategy, *etc.*).

Rebuffering. As an indicator of rebuffering time per streaming mode switch, we measure the overall duration of audio and video stalls at the viewer in a 10-second period before and after each switch.[2] We report average results across multiple trials in Figure 6. Obviously, upgrading to AnchorNet brings superior mode switching experience. Thanks to the optimized audio splicing techniques, it encounters no audio rebuffering in all cases. AnchorNet also outperforms the other tested schemes regarding video rebuffering. In comparison, DualNet always suffers from second-level stalls for both audio and video. Competitor-1, although based on viewer-side stream mixing, consistently experiences a short audio pause of around 100 ms in every switch. Notably, Competitor-2 induces a long stall when the channel switches to collaborative streaming for the first time, but performs comparable to AnchorNet in subsequent streaming mode switching. Correspondingly, we notice that Competitor-2 starts each live streaming viewer session in media audio mode, and changes to call mode [37] when the channel first switches to collaborative streaming. It stays in call audio mode afterwards. We infer that some implementation-specific module restart is triggered in that process, causing the long rebuffering.

Streaming Delay. Then, we compare the broadcaster-to-viewer streaming delay of the tested schemes. For convenience of calculation, we put a timer in the camera of the host broadcaster, and record the difference between the displayed time at the host and the viewer multiple times during each trial. As shown in Figure 7, the streaming delay of Competitor-2 more than doubles that of AnchorNet, possibly due to a much larger jitter buffer within the network. That may contribute to Competitor-2's rebuffering performance during streaming mode switching, which is quite close to AnchorNet other than the long stall in the first switch to collaborative streaming. Competitor-1, leveraging a unified RTC stack, achieves the shortest delay under 4 seconds. AnchorNet and DualNet are not far behind, with AnchorNet being slightly higher due to the round trip between the RTC SFU and MCU. Such a streaming delay, together with the least perceivable glitches during streaming mode switching, delivers satisfying ROI for a commercially large-scale deployed system like TikTok.

[2]The same as [20], we filter instances of audio and video stalls using empirical thresholds of 80 ms and 200 ms, respectively. Micro stalls below the thresholds are ignored.

7 Evaluation

7.1 User Experience Gains

Currently, we are in the process of deploying AnchorNet. To compare it with DualNet, we present QoS and QoE results from one of our large-scale A/B tests in production environment. The test was conducted in 2024Q4 and lasted for 4 weeks. It involved almost the same number of users served by AnchorNet and DualNet, respectively (Figure 8). Table 2 summarizes the performance of AnchorNet relative to DualNet. All listed metrics are statistically significant under a confidence level of at least 95%. More detailed analysis and results are presented as follows, with the data normalized against the maximum value from respective metric.

Rebuffering during mode switching. Similar to §6, we first look at rebuffering during streaming mode switching, which is the most important category of metrics AnchorNet aims to optimize. Unsurprisingly, AnchorNet significantly reduces both the duration and count of video/audio rebuffering. Figure 9 showcases the trend of video rebuffering throughout the 4-week period. In the case of switching from live streaming to collaborative streaming, each viewer session encounters 60.3% shorter stall duration and 60.1% less stall count on average under AnchorNet. When switching back to live streaming, AnchorNet's gains on the same two metrics reach 78.9% and 78.5%. It is normal that the gains from practical A/B test are not as extreme as achieved in our benchmarking

Rebuffering during mode switching (per viewer session)	Live to collaborative streaming		Collaborative to live streaming	
	Rebuffering Count	Rebuffering Duration	Rebuffering Count	Rebuffering Duration
	Video: −60.1% Audio: −64.5%	Video: −60.3% Audio: −67.4%	Video: −78.5% Audio: −76.3%	Video: −78.9% Audio: −77.1%
User engagement (daily active time)	Host streaming	Co-broadcaster streaming	Viewer watching	
	+0.53%	+2.15%	+3.83%	
Live streaming mode only	Rebuffering	Host-side push FPS	Host device pressure	
	Video: +13.47% Audio: +11.02%	+3.11%	CPU: −1.33% GPU: +0.80% MEM: +0.31%	

Table 2: A/B Test Results (Metrics on which AnchorNet outperforms DualNet are marked by underlined numbers)

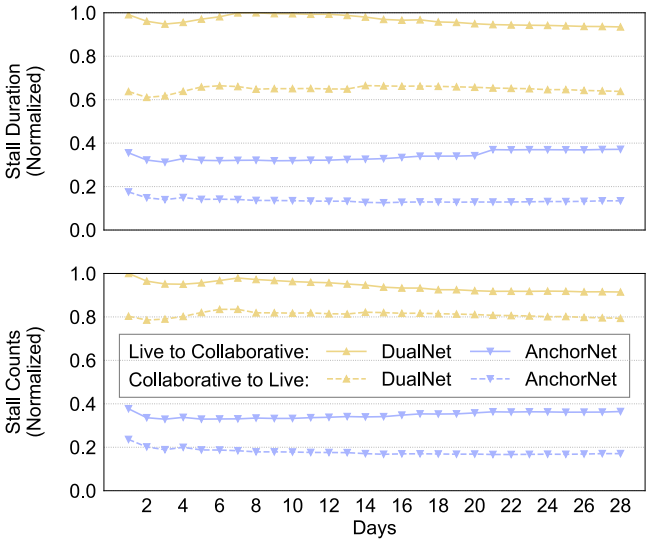


Figure 9: Duration and count of video rebuffering per viewer session

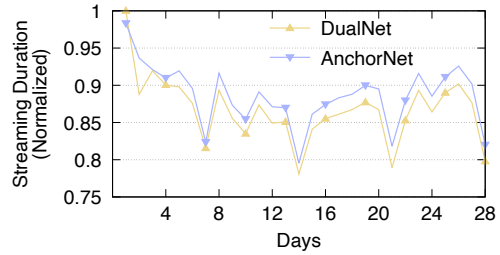


Figure 10: Daily active time per co-broadcaster

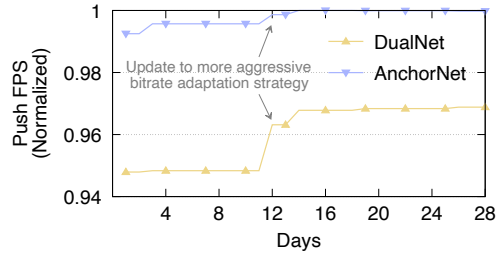
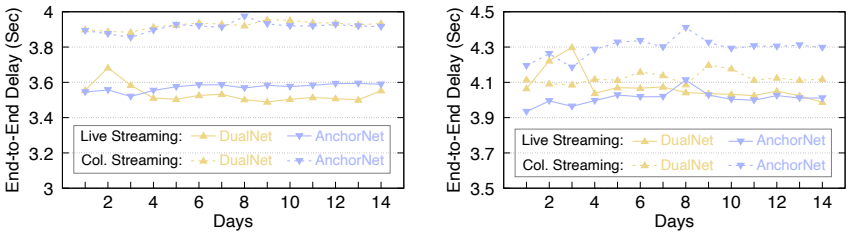


Figure 11: Push FPS in live streaming mode

test in §6. That is mainly because users' access links are much more complicated than our ideal setup. Audio rebuffering also follows a similar trend, so we omit due to limited space.

User engagement. Frequent rebuffering during streaming mode switching may cause some impatient users to quit, leading to impaired user retention and stickiness and even commercial loss. We expect a smoother switching experience, while making our users more satisfied, can ultimately improve user engagement. Fortunately, that is validated by the A/B test. AnchorNet increases the daily active time for all user roles, *i.e.*, 0.53% for channel host, 2.15% for guest co-broadcaster, and 3.83% for viewers. More intuitively, as illustrated in Figure 10, AnchorNet outperforms DualNet in terms of daily duration each co-broadcaster stays in collaborative streaming throughout the A/B test. We note that the active time of each viewer session encompasses every streaming mode switching of the watched channel. The reduction of viewer-side rebuffering during streaming mode switching cannot directly contribute to increment in viewer watching duration. Thus, the longer active time as in Table 2 reflects true user engagement growth.

Live streaming mode improvements. As explained in §4.2, viewer's experience in the live streaming mode may benefit from leveraging optimized user-space congestion control algorithms. Indeed, thanks to better resistance against uplink fluctuations (*e.g.*, random packet losses), AnchorNet lowers video and audio rebuffering in live streaming mode by as high as 13.47% and 11.02%, respectively (Table 2). The average frame rate pushed by channel hosts is increased by 3.11%, as well. It is worth nothing that we rolled out a more aggressive broadcaster-side bitrate adaptation strategy on the 12th day of the A/B test. AnchorNet maintains its advantage in push FPS (video frames per second ingested by the host in live streaming mode) after that (Figure 11). Meanwhile, since congestion and flow control are moved from kernel to user-space, AnchorNet puts slightly higher pressure on the host device, *e.g.*, it consumes 0.31% more memory in live streaming. The reduction of CPU consumption by 1.33% in Table 2 is because we prioritize hardware codec acceleration in our AnchorNet implementation, which leads to a 0.8% increment in GPU usage. We regard those slight increments as acceptable compared to gains in rebuffering and frame rate.



(a) Global (b) A selected country

Figure 12: End-to-end streaming delay

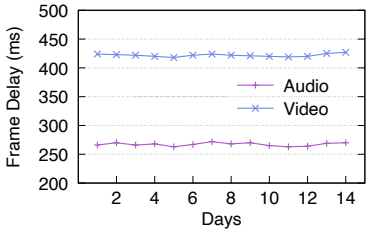


Figure 13: SFU-MCU round trip (including MCU processing time)

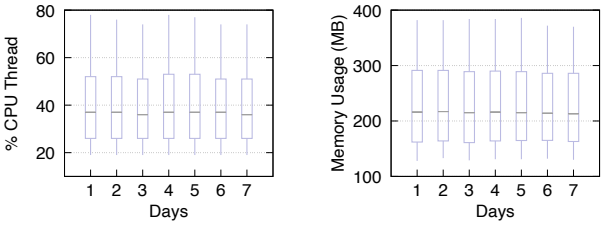
7.2 Server-Side Stream Mixing Feasibility

To provide exhaustive practical insights, we also demonstrate the performance and overhead of server-side stream mixing, which is common in both AnchorNet and DualNet.

End-to-end delay. As explained in §4.1, a concern in AnchorNet is the additional delay due to the round-trip between SFU and MCU in collaborative streaming. We track the end-to-end streaming delay in a two-week period in December 2024 in Figure 12. It turns out that on a global scale, AnchorNet has an average streaming delay comparable to DualNet (Figure 12a). This is because DualNet conservatively reserves a large jitter buffer at CDN upon entering collaborative streaming in reaction to the rebuffering during streaming mode switching. In many cases, that happens to counteract or even surpass the increased delay due to the SFU-MCU round trip in AnchorNet. Nevertheless, in those areas far from the nearest MCU cluster, AnchorNet may experience a longer collaborative streaming delay than DualNet due to the additional round trip. For example, Figure 12b shows the streaming delay in such a selected country. In collaborative streaming, AnchorNet raises the end-to-end delay by 170 ms on average compared with DualNet. We plan to further optimize on that front with wider edge MCU deployment in the future.

Stream mixing delay. Another observation from Figure 12 is that collaborative streaming has an end-to-end delay hundreds of milliseconds longer than live streaming mode. Besides the transport delay on the SFU-MCU round trip, such a delay gap also includes the time consumed by stream mixing and transcoding at MCU. Figure 13 presents the average per-frame delay in collaborative streaming between SFU and MCU. In AnchorNet, that delay is concealed by CDN and viewer-side jitter buffers. As explained in §8, we also adjust the playback speed at the viewer when switching between streaming modes to further avoid rebuffering.

Stream mixing overhead. At last, to quantify stream mixing overhead, we record average CPU and memory usage for each stream mixing task. Such a task is initiated when a live channel enters collaborative streaming, and terminated when the channel exits collaborative streaming. The whole streaming session of a host broadcaster may start multiple stream mixing tasks. Figure 14 shows the two metrics at different percentiles over one week in December 2024. In the median case, each task costs less than 0.4 logical CPU core and 220 MB memory. For a server instance with 128 threads (64 physical CPU cores) and above 500 GB memory, 100 such MCUs may accommodate up to 15 $\sim$ 30K concurrent collaborative streaming channels. According to our own assessment, we consider such server machine costs as affordable.



(a) CPU Overhead (b) Memory Overhead

Figure 14: Overhead per stream mixing task (Results at 10th, 25th, 50th, 75th, 90th percentile at dipicted)

8 Deployment Experiences

8.1 User Experience Optimization

AV synchronization. The switching of streaming mode comes with a change in user interface layout, *e.g.*, between a full screen view of the host broadcaster and a grid gallery of all co-broadcasters. In our internal subjective tests, we find that it is hard to notice momentary video stalls during such a layout change. Therefore, we can inject artificial video rebuffering (*i.e.*, by re-encoding or replaying several previous video frames) when switching the streaming mode for the purpose of AV (audio-to-video) synchronization. While doing so, we still try to deliver seamless audio, and manage to "conceal" video rebuffering during UI transitions.

Adaptive playback speed. Adaptive playback speed control [38, 39] has been a common technique in live streaming to resist rebuffering. In AnchorNet, it is also used to handle the gap of end-to-end streaming delay between the two streaming modes. When switching to collaborative streaming, the viewer-side player may slow down the live streaming playback, if the remaining buffered frames has the risk of being exhausted before arrival of the first collaborative streaming frame. When switching to live streaming, the player may otherwise speed up the playback to lower the end-to-end delay.

Adaptive video bitrate. On the publishing path to CDN, the (co-)broadcaster adjusts the video resolution and bitrate according to not only available uplink bandwidth, but also

real-time device performance. Just like insufficient bandwidth, conditions such as low battery power and overheating also cause bitrate degradation. Simulcast [20] is another option for adaptive bitrate between co-broadcasters. We do not activate simulcast in AnchorNet to save bandwidth consumption on broadcaster side. On the distribution path to viewer, we rely on CDN to transcode the mixed stream into lower-bitrate versions, to accommodate viewers with a weak access network.

8.2 Tradeoff between Experience and Cost

After fulfilling the dominant majority of users, we tend to prioritize cost control over the pursuit of extreme QoE.

RTC server distribution. When analyzing issues of our first-generation architecture in §3.2, we exemplify a scenario where there are only CDN PoPs but no RTC servers in a country. That is a common result of lowering server-side cost. For CDN distribution, we can consume standard service from 3rd-party providers in addition to constructing our own clusters. Yet for RTC servers, we need to purchase server instances to deploy our server-side SDK. The latter is often more expensive. Furthermore, we do not need RTC servers to be distributed as widely as CDN PoPs, because non-RTC services (*e.g.*, on-demand short video streaming) currently account for a significantly higher traffic volume at TikTok. At this stage, we geographically plan RTC clusters so that end-to-end latency of our RTC services satisfy industry norms.

Streaming protocol selection. Aimed at sub-second level end-to-end delay, some pioneering CDN and cloud providers have supported end-to-end WebRTC-based live streaming [30, 40–42]. However, that is not yet universally supported, *e.g.*, it is listed as a beta feature at some providers [43]. Therefore, WebRTC-based streaming service is usually more expensive than more mature RTMP ingest and HTTP distribution [44]. As of now, some CDN providers double their charges for WebRTC-based distribution over traditional distribution [45, 46]. Although end-to-end WebRTC can further reduce the streaming delay, that only brings marginal benefits considering that AnchorNet already satisfies most of our viewers' latency requirements. Therefore, we stick to the choices of RTMP ingest and HTTP distribution for now.

8.3 Future Direction: Enhanced Flexibility

AnchorNet adopts a fixed publishing path (*e.g.*, SFU as CDN publisher, stream mixing on server side) and static streaming protocols (*e.g.*, RTMP ingest, HTTP distribution), to lower implementation complexity and balance server-side cost. As AnchorNet approaches full deployment, we plan to introduce more flexibility to our streaming architecture.

Flexible stream mixing spot. AnchorNet effectively implements viewer-side stream mixing between co-broadcasters. Each co-broadcaster receives other co-broadcasters' streams from the connected SFU, and renders them altogether. To balance server-side cost and user experience more flexibly, AnchorNet can be combined with the other two stream mixing choices accordingly. For example, for a host broadcaster using a recently released high-performance smartphone, we can offload stream mixing to the broadcaster side when the number of co-broadcasters is below a small threshold. In certain use cases urging tight end-to-end latency (*e.g.*, e-commerce), a proportion of high-value customers may utilize viewer-side stream mixing, by pulling (co-)broadcasters' streams from SFU using RTP instead of HTTP FLV.

There are also many other open research problems not listed here that are orthogonal to streaming architecture design. For example, the subjective QoE study based on a unified system like AnchorNet will need to synthesize different user roles with possibly heterogeneous requirements (e.g., co-broadcasters as RTC users, and viewers as non-RTC streaming consumers).

9 Related Works

Stream splicing. One of the early discussions on the stream splicing problem dealt with the switching between television programs. Some works [47, 48] proposed to splice the video streams of two programs at the boundary of two video frames. Since audio signals before and after program switching come from different sources, a short gap is acceptable. In the context of VoIP, [49–52] worked on switching audio codec amid an ongoing session, *e.g.*, selecting the optimal codec based on network conditions. They focused more on the audio fidelity (*e.g.*, MOS, Mean Opinion Score) achieved by different codecs, and did not eliminate the audio frame-level glitches as we do in §5. Similar codec switching was also studied in video streaming [53–55]. Different from the audio case, seamless video can be ensured as long as the codec switching does not break the completeness of individual video frames.

CDN streaming architecture. CDN-based content distribution has been extensively studied aimed at both on-demand web contents (*e.g.*, DASH video [56]) [57–60] and live streaming [61–63]. Recently, the surge of real-time communication poses the requirement for low-latency delivery. To fulfill more stringent end-to-end latency, LiveNet [24] presented a flat CDN architecture with dynamically optimized per-live stream overlay distribution paths. In addition, some evolved CDN architectures were explored. PCDN [64] enabled end users to download from multiple more affordable edge nodes in parallel for better server-side cost-effectiveness. Tian *et al.* proposed to offload user traffic to cheaper alternative edge nodes (*e.g.*, user-leased set-top boxes) [65]. Those works are complementary to AnchorNet, but they mostly focused on the live streaming mode only, *e.g.*, each viewer subscribes to a live stream with a single broadcaster.

RTC conferencing architecture. A typical scenario of interactive real-time communication is multi-party conferencing. On that front, some early-stage WebRTC implementations adopted a P2P approach [66, 67], letting end users directly connect to each other without intermediate server participation.

While scaling to larger conferences, most service providers deploy the MCU or SFU architecture using media servers. From the perspective of where to mix streams, MCU belongs to server-side stream mixing, and SFU enforces viewer-side stream mixing. SFU can be utilized by simulcast [19, 20, 68] to adapt to heterogeneous access links at different participants, *i.e.*, each publisher uploads the same stream in different bitrates, and SFU determines which to forward. With lower server-side computation overhead, SFU has been the dominating architecture in modern multi-party video conferencing [20, 69, 70]. In AnchorNet, co-broadcasters effectively form a multi-party conference room connected via SFUs, and an MCU is leveraged towards the viewers of a channel. As explained in §2.2, the latter choice is mainly owing to the fact that the number of viewers per live channel may be dramatically larger than that of participants per conference room.

First/Last-mile transport optimization. Another research direction that are orthogonal to AnchorNet is the optimization of transport protocols and congestion control algorithms targeted at fluctuating first/last-mile access links. Multipath transport protocols attract widespread attention due to its ability to combine bandwidth of different links and avoid temporary latency inflation on a specific link [71–74]. Congestion control algorithms are specifically designed to satisfy low latency requirements of live media contents [75–77]. A big category of works adopt learning approaches to adapt to complicated network characteristics. For example, Antelope [78] trained a machine learning model to dynamically select the congestion control algorithm best suited for given network conditions. SLVS [79] tuned the bitrate for live streaming by learning from various network features. AutoRec [80] took advantage of the on-off pattern in the transmission of live streams in an online learning-based loss recovery policy.

10 Conclusion

In this paper, we share the design and deployment of AnchorNet, TikTok's new live streaming architecture. Built on basis of our first-generation architecture, DualNet, we set smooth switching between live and collaborative streaming as the top objective of AnchorNet. For that purpose, it unifies the publishing path from the broadcaster to CDN by placing RTC servers between them, and leverages various splicing techniques to avoid audio glitches induced due to inherent codec issues such as encoder delay. AnchorNet significantly outperforms DualNet in our large-scale A/B test. More importantly, AnchorNet, representing our own design perspectives, provides practical insights on operating a live streaming service on a global scale.

References

[1] A beginner's guide on how to stream Just Chatting on Twitch. https://resources.overlays.uno/post/how-to-stream-just-chatting-on-twitch. Accessed: 2025.1.

[2] Invite guests to join your live broadcast on Instagram. https://help.instagram.com/126662934757177. Accessed: 2025.1.

[3] TikTok LIVE multi-guest. https://support.tiktok.com/en/live-gifts-wallet/tiktok-live/tiktok-live-multi-guest. Accessed: 2025.1.

[4] Invite a guest to your live stream with YouTube Go Live Together. https://support.google.com/youtube/answer/12112663. Accessed: 2025.1.

[5] Twitch State of Engineering 2023. https://blog.twitch.tv/en/2023/09/28/twitch-state-of-engineering-2023/. Accessed: 2025.1.

[6] YouTube Live Streaming Ingestion Protocol Comparison. https://developers.google.com/youtube/v3/live/guides/ingestion-protocol-comparison. Accessed: 2025.1.

[7] Instagram Live Producer. https://about.instagram.com/blog/tips-and-tricks/instagram-live-producer. Accessed: 2025.1.

[8] Twitch Broadcasting Guidelines. https://help.twitch.tv/s/article/broadcasting-guidelines. Accessed: 2025.1.

[9] YouTube Live Encoder Settings, Bitrates, and Resolutions. https://support.google.com/youtube/answer/2853702. Accessed: 2025.1.

[10] Karlheinz Brandenburg. Mp3 and aac explained. In *Audio Engineering Society Conference: 17th International Conference: High-Quality Audio Coding*, 1999.

[11] Jean-Marc Valin and Cary Bran. WebRTC Audio Codec and Processing Requirements. RFC 7874, May 2016.

[12] WebRTC Opus Codec. https://webrtc.googlesource.com/src/+/refs/heads/main/modules/audio_coding/codecs/opus/. Accessed: 2025.1.

[13] Discord Voice Connections. https://discord.com/developers/docs/topics/voice-connections. Accessed: 2025.1.

[14] MLow: Meta's low bitrate audio codec. https://engineering.fb.com/2024/06/13/web/mlow-metas-low-bitrate-audio-codec/. Accessed: 2025.1.

[15] Jean-Marc Valin, Koen Vos, and Timothy B. Terriberry. Definition of the Opus Audio Codec. RFC 6716, September 2012.

[16] Marc Gayer, Manfred Lutzky, Gerald Schuller, Ulrich Krämer, and Stefan Wabnik. A guideline to audio codec delay. In *Audio Engineering Society Convention 116*. Audio Engineering Society, 2004.

[17] Arvind Narayanan, Eman Ramadan, Jason Carpenter, Qingxu Liu, Yu Liu, Feng Qian, and Zhi-Li Zhang. A first look at commercial 5g performance on smartphones. In *WWW*, pages 894–905, 2020.

[18] Zili Meng, Yaning Guo, Chen Sun, Bo Wang, Justine Sherry, Hongqiang Harry Liu, and Mingwei Xu. Achieving consistent low latency for wireless real-time communications with the shortest control loop. In *ACM SIGCOMM*, pages 193–206, 2022.

[19] Tong Meng, Wenfeng Li, Chao Yuan, Changqing Yan, and Le Zhang. AsTree: An Audio Subscription Architecture Enabling Massive-Scale Multi-Party Conferencing. In *USENIX NSDI*, pages 653–666, 2025.

[20] Xianshang Lin, Yunfei Ma, Junshao Zhang, Yao Cui, Jing Li, Shi Bai, Ziyue Zhang, Dennis Cai, Hongqiang Harry Liu, and Ming Zhang. GSO-simulcast: global stream orchestration in simulcast video conferencing systems. In *ACM SIGCOMM*, pages 826–839, 2022.

[21] Yihua Cheng, Ziyi Zhang, Hanchen Li, Anton Arapin, Yue Zhang, Qizheng Zhang, Yuhan Liu, Kuntai Du, Xu Zhang, Francis Y Yan, et al. GRACE: Loss-Resilient Real-Time Video through Neural Codecs. In *USENIX NSDI*, pages 509–531, 2024.

[22] Tencent TRTC: Mic Connect. https://www.tencentcloud.com/document/product/1071/42210. Accessed: 2025.1.

[23] Alibaba ARTC: Cloud-based stream mixing and relay. https://www.alibabacloud.com/help/en/live/user-guide/cloud-mixing-and-retweeting. Accessed: 2025.1.

[24] Jinyang Li, Zhenyu Li, Ri Lu, Kai Xiao, Songlin Li, Jufeng Chen, Jingyu Yang, Chunli Zong, Aiyun Chen, Qinghua Wu, et al. Livenet: a low-latency video transport network for large-scale live streaming. In *ACM SIGCOMM*, pages 812–825, 2022.

[25] Arthi Padmanabhan, Neil Agarwal, Anand Iyer, Ganesh Ananthanarayanan, Yuanchao Shu, Nikolaos Karianakis, Guoqing Harry Xu, and Ravi Netravali. Gemel: Model Merging for Memory-Efficient,Real-Time Video Analytics at the Edge. In *USENIX NSDI*, pages 973–994, 2023.

[26] Shadi A Noghabi, Landon Cox, Sharad Agarwal, and Ganesh Ananthanarayanan. The emerging landscape of edge computing. *GetMobile: Mobile Computing and Communications*, 23(4):11–20, 2020.

[27] AWS Outposts. https://aws.amazon.com/outposts/. Accessed: 2025.1.

[28] Azure Stack Edge. https://azure.microsoft.com/en-us/products/azure-stack/edge/. Accessed: 2025.1.

[29] Overview of the Live Stream API (Google Cloud). https://cloud.google.com/livestream/docs/overview. Accessed: 2025.1.

[30] Alibaba Cloud Real Time Streaming. https://www.alibabacloud.com/en/product/apsaravideo-for-live/rts. Accessed: 2025.1.

[31] Amazon IVS Real-Time Streaming now supports RTMP ingest. https://aws.amazon.com/about-aws/whats-new/2024/09/amazon-ivs-real-time-streaming-rtmp-ingest. Accessed: 2025.1.

[32] Live streaming with Azure Media Services v3. https://learn.microsoft.com/en-us/azure/media-services/latest/stream-live-streaming-concept. Accessed: 2025.1.

[33] Slavik Lozben. Enhanced RTMP (V2). https://veovera.org/docs/enhanced/enhanced-rtmp-v2. Accessed: 2025.1.

[34] AAC encoding background. https://developer.apple.com/documentation/quicktime-file-format/background_aac_encoding. Accessed: 2025.1.

[35] Jürgen Herre and Martin Dietz. Mpeg-4 high-efficiency aac coding [standards in a nutshell]. *IEEE Signal Processing Magazine*, 25(3):137–142, 2008.

[36] Opus reference implementation API. https://chromium.googlesource.com/chromium/deps/opus/+/refs/heads/main/include/opus.h. Accessed: 2025.1.

[37] What is the difference between in-call volume and media volume? https://docs.byteplus.com/en/docs/byteplus-rtc/docs-651068. Accessed: 2025.1.

[38] Mark Kalman, Eckehard Steinbach, and Bernd Girod. Adaptive media playout for low-delay video streaming over error-prone channels. *IEEE Transactions on Circuits and Systems for Video Technology*, 14(6):841–851, 2004.

[39] Liyang Sun, Tongyu Zong, Siquan Wang, Yong Liu, and Yao Wang. Tightrope walking in low-latency live streaming: Optimal joint adaptation of video rate and playback speed. In *ACM MMSys*, pages 200–213, 2021.

[40] Tencent Live Event Broadcasting (LEB). https://www.tencentcloud.com/document/product/267/42141. Accessed: 2025.1.

[41] Wowza: Real-Time Streaming at Scale. https://www.wowza.com/video/real-time-video-streaming. Accessed: 2025.1.

[42] Dolby.io Real Time Streaming. https://dolby.io/solutions/real-time-streaming/. Accessed: 2025.1.

[43] Cloudflare Stream: WebRTC. https://developers.cloudflare.com/stream/webrtc-beta/. Accessed: 2025.1.

[44] David Zhao Russ d'Sa, Neil Dwyer. A tale of two protocols: comparing WebRTC against HLS for live streaming. https://blog.livekit.io/webrtc-vs-hls-livestreaming/. Accessed: 2025.1.

[45] Alibaba ApsaraVideo Live Pricing. https://www.alibabacloud.com/en/product/apsaravideo-for-live/pricing. Accessed: 2025.1.

[46] Billing of Tencent Live Event Broadcasting (LEB). https://www.tencentcloud.com/document/product/267/39969. Accessed: 2025.1.

[47] SMPTE STANDARD. Splice points for mpeg-2 transport streams. 2001.

[48] Seyfullah Halit Oguz and Sorin Faibish. Seamless audio splicing for iso/iec 13818 transport streams. In *142nd Technical Conference and Exhibition*, pages 1–16. SMPTE, 2000.

[49] Ismet Aktas, Florian Schmidt, Elias Weingärtner, Cai-Julian Schnelke, and Klaus Wehrle. An adaptive codec switching scheme for sip-based voip. In *Internet of Things, Smart Spaces, and Next Generation Networking: 12th International Conference, NEW2AN 2012, and 5th Conference, ruSMART 2012, St. Petersburg, Russia, August 27-29, 2012. Proceedings*, pages 347–358. Springer, 2012.

[50] Maja Sulovic, Darijo Raca, Mesud Hadzialic, and Nasuf Hadziahmetovic. Dynamic codec selection algorithm for voip. In *Proc. 6th International Conference on Digital Telecommunications (ICDT 2011)*, pages 74–79, 2011.

[51] Haytham Assem, Mohamed Adel, Brendan Jennings, David Malone, Jonathan Dunne, and Pat O'Sullivan. A generic algorithm for mid-call audio codec switching. In *2013 IFIP/IEEE International Symposium on Integrated Network Management (IM 2013)*, pages 1276–1281. IEEE, 2013.

[52] MARCEL Waltermann, Blazej Lewcio, Pablo Vidales, and Sebastian Moller. A technique for seamless voip-codec switching in next generation networks. In *IEEE ICC*, pages 1772–1776. IEEE, 2008.

[53] Daniele Lorenzi, Farzad Tashtarian, Hermann Hellwagner, and Christian Timmerer. Medusa: A dynamic codec switching approach in http adaptive streaming. *ACM Transactions on Multimedia Computing, Communications and Applications*, 2024.

[54] Babak Taraghi, Hadi Amirpour, and Christian Timmerer. Multi-codec ultra high definition 8k mpeg-dash dataset. In *ACM MMSys*, pages 216–220, 2022.

[55] Daniele Lorenzi, Farzad Tashtarian, Hadi Amirpour, Christian Timmerer, and Hermann Hellwagner. Mcomlive: A multi-codec optimization model at the edge for live streaming. In *International Conference on Multimedia Modeling*, pages 252–264. Springer, 2023.

[56] ISO Standard. Dynamic adaptive streaming over http (dash)-part 1: Media presentation description and segment formats. *ISO/IEC*, pages 23009–1, 2014.

[57] Stefan Saroiu, Krishna P Gummadi, Richard J Dunn, Steven D Gribble, and Henry M Levy. An analysis of internet content delivery systems. *ACM SIGOPS Operating Systems Review*, 36(SI):315–327, 2002.

[58] Al-Mukaddim Khan Pathan, Rajkumar Buyya, et al. A taxonomy and survey of content delivery networks. *Grid computing and distributed systems laboratory, University of Melbourne, Technical Report*, 4(2007):70, 2007.

[59] Athena Vakali and George Pallis. Content delivery networks: Status and trends. *IEEE Internet Computing*, 7(6):68–74, 2003.

[60] Volker Stocker, Georgios Smaragdakis, William Lehr, and Steven Bauer. The growing complexity of content delivery networks: Challenges and implications for the internet ecosystem. *Telecommunications Policy*, 41(10):1003–1016, 2017.

[61] Ramesh K Sitaraman, Mangesh Kasbekar, Woody Lichtenstein, and Manish Jain. Overlay networks: An akamai perspective. *Advanced Content Delivery, Streaming, and Cloud Services*, pages 305–328, 2014.

[62] Leonidas Kontothanassis, Ramesh Sitaraman, Joel Wein, Duke Hong, Robert Kleinberg, Brian Mancuso, David Shaw, and Daniel Stodolsky. A transport layer for live streaming in a content delivery network. *Proceedings of the IEEE*, 92(9):1408–1419, 2004.

[63] Konstantin Andreev, Bruce M Maggs, Adam Meyerson, and Ramesh K Sitaraman. Designing overlay multicast networks for streaming. In *ACM SPAA*, pages 149–158, 2003.

[64] Rui-Xiao Zhang, Haiping Wang, Shu Shi, Xiaofei Pang, Yajie Peng, Zhichen Xue, and Jiangchuan Liu. Enhancing Resource Management of the World's Largest PCDN System for On-Demand Video Streaming. In *USENIX ATC*, pages 951–965, 2024.

[65] Yu Tian, Zhenyu Li, Matthew Yang Liu, Jian Mao, Gareth Tyson, and Gaogang Xie. Cost-saving streaming: Unlocking the potential of alternative edge node resources. In *ACM IMC*, pages 580–587, 2024.

[66] Khalid Ibn Zinnah Apu, Nafiz Mahmud, Firoz Hasan, and Sabbir Hossain Sagar. P2p video conferencing system based on webrtc. In *2017 International Conference on Electrical, Computer and Communication Engineering (ECCE)*, pages 557–561. IEEE, 2017.

[67] Yang Xu, Chenguang Yu, Jingjiang Li, and Yong Liu. Video telephony for end-consumers: Measurement study of google+, ichat, and skype. In *ACM IMC*, pages 371–384, 2012.

[68] David Zhao. An introduction to WebRTC Simulcast. https://blog.livekit.io/an-introduction-to-webrtc-simulcast-6c5f1f6402eb/. Accessed: 2025.1.

[69] Amazon Chime Video Simulcast. https://aws.github.io/amazon-chime-sdk-js/modules/simulcast.html. Accessed: 2025.1.

[70] Jitsi Videobridge SFU. https://github.com/jitsi/jitsi-videobridge. Accessed: 2025.1.

[71] Zhilong Zheng, Yunfei Ma, Yanmei Liu, Furong Yang, Zhenyu Li, Yuanbo Zhang, Jiuhai Zhang, Wei Shi, Wentao Chen, Ding Li, et al. Xlink: Qoe-driven multi-path quic transport in large-scale video services. In *ACM SIGCOMM*, pages 418–432, 2021.

[72] Yunzhe Ni, Zhilong Zheng, Xianshang Lin, Fengyu Gao, Xuan Zeng, Yirui Liu, Tao Xu, Hua Wang, Zhidong Zhang, Senlang Du, et al. Cellfusion: Multipath vehicle-to-cloud video streaming with network coding in the wild. In *ACM SIGCOMM*, pages 668–683, 2023.

[73] Sandesh Dhawaskar Sathyanarayana, Kyunghan Lee, Dirk Grunwald, and Sangtae Ha. Converge: Qoe-driven multipath video conferencing over webrtc. In *ACM SIGCOMM*, pages 637–653, 2023.

[74] Yuhan Zhou, Tingfeng Wang, Liying Wang, Nian Wen, Rui Han, Jing Wang, Chenglei Wu, Jiafeng Chen, Longwei Jiang, Shibo Wang, et al. AUGUR: Practical Mobile Multipath Transport Service for Low Tail Latency in Real-Time Streaming. In *USENIX NSDI*, pages 1901–1916, 2024.

[75] Gaetano Carlucci, Luca De Cicco, Stefan Holmer, and Saverio Mascolo. Analysis and design of the google congestion control for web real-time communication (webrtc). In *ACM MMSys*, pages 1–12, 2016.

[76] Shibo Wang, Shusen Yang, Xiao Kong, Chenglei Wu, Longwei Jiang, Chenren Xu, Cong Zhao, Xuesong Yang, Jianjun Xiao, Xin Liu, et al. Pudica: Toward Near-Zero Queuing Delay in Congestion Control for Cloud Gaming. In *USENIX NSDI*, pages 113–129, 2024.

[77] Wei Zhang, Xuefeng Tao, and Jianan Wang. Naorl: Network feature aware offline reinforcement learning for real time bandwidth estimation. In *ACM MMSys*, pages 326–331, 2024.

[78] Jianer Zhou, Xinyi Qiu, Zhenyu Li, Gareth Tyson, Qing Li, Jingpu Duan, and Yi Wang. Antelope: A framework for dynamic selection of congestion control algorithms. In *IEEE ICNP*, pages 1–11. IEEE, 2021.

[79] Guanghui Zhang, Ke Liu, Mengbai Xiao, Bingshu Wang, and Vaneet Aggarwal. An intelligent learning approach to achieve near-second low-latency live video streaming under highly fluctuating networks. In *ACM MM*, pages 8067–8075, 2023.

[80] Bo Wu, Tong Li, Cheng Luo, Xu Yan, Fuyu Wang, Xinle Du, and Ke Xu. Toward timeliness-enhanced loss recovery for large-scale live streaming. In *ACM MM*, pages 7891–7899, 2024.

KATZ: Efficient Workflow Serving for Diffusion Models with Many Adapters

Suyi Li[†*], Lingyun Yang[†*], Xiaoxiao Jiang[†], Hanfeng Lu[†], Dakai An[†], Zhipeng Di,
Weiyi Lu, Jiawei Chen, Kan Liu, Yinghao Yu, Tao Lan, Guodong Yang, Lin Qu, Liping Zhang, Wei Wang[†]
[†]*Hong Kong University of Science and Technology* *Alibaba Group*
{slida, lyangbk, weiwa}@cse.ust.hk

Abstract

Text-to-image (T2I) generation using diffusion models has become a blockbuster service in today's AI cloud. A production T2I service typically involves a serving *workflow* where a base diffusion model is augmented with many ControlNet and LoRA adapters to control the details of output images, such as shapes, outlines, poses, and styles. In this paper, we present KATZ, a system that efficiently serves a T2I workflow with many adapters. KATZ differentiates compute-heavy ControlNets from compute-light LoRAs, where the former introduces significant computational overheads while the latter is bottlenecked by loading. KATZ proposes to take ControlNet off the critical path with a ControlNet-as-a-Service design, in which ControlNets are decoupled from the base model and deployed as a *separate, independently scalable service* on dedicated GPUs, thus enabling ControlNet caching, parallelization, and sharing. To hide the high LoRA loading overhead, KATZ employs *bounded asynchronous loading* that overlaps LoRA loading with initial base model execution by a maximum of K steps, while maintaining the same image quality. KATZ further accelerates base model execution across multiple GPUs with *latent parallelism*. Collectively, these designs enable KATZ to outperform the state-of-the-art T2I serving systems, achieving up to 7.8× latency reduction and 1.7× throughput improvement in serving SDXL models on H800 GPUs, without compromising image quality.

1 Introduction

Text-to-image (T2I) generation using diffusion models is a transformative AI technology that enables the creation of high-quality, contextually accurate images from textual descriptions. This technology has gained immense popularity, with a plethora of commercial T2I services available in the cloud, such as DALL·E [9], Midjourney [5], and Firefly [11].

A production T2I service is typically deployed as a *workflow* consisting of multiple components. At its core is a base stable diffusion model [19, 32, 39]. This model is trained to produce a coherent image through a *reverse diffusion process* [40]: it starts with an image composed of random noises and progressively denoises this random input in iterations, until the output image aligns with the provided text description. A base diffusion model is often augmented with various *adapters* to better control the details of the output images, such as shapes, outlines, poses, and styles. Fig. 1 illustrates the effects of using ControlNet [69] and LoRA (Low-Rank Adaptation [28]), the two most popular adapters used in production. ControlNet allows users to input a reference image to guide the spatial composition of the generated image; LoRA produces an image with customized stylistic effects. In our production platform, over 98% of requests demand at least one ControlNet, and over 95% utilize at least one LoRA (§3).

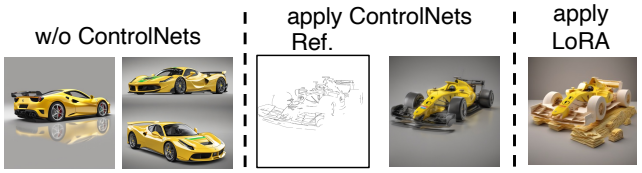


Figure 1: Effects of ControlNet and LoRA in image generation with SDXL under the same prompt: racing game car, yellow Ferrari. **Left**: without ControlNet, the generated images can have different compositions. **Center**: ControlNet uses a reference image to control the composition. **Right**: using LoRA to generate image in a papercut style.

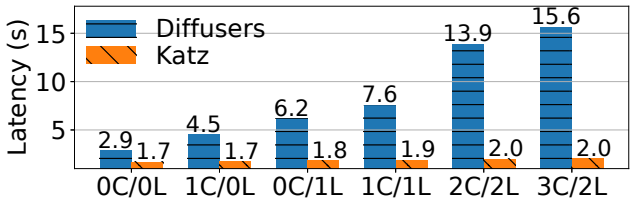


Figure 2: **C**ontrolNets and **L**oRAs introduce additional latency overhead. In each workflow, a base SDXL [39] model is augmented with m ControlNets and n LoRAs (mC/nL), served by DIFFUSERS [50] and KATZ on H800 GPUs.

However, the use of adapters poses new performance challenges. To illustrate this problem, we configure a T2I workflow where a base SDXL model [39] is augmented with a varying number of ControlNets and LoRAs. We depict in Fig. 2 the serving latency of these workflows (blue bar). Compared with serving the base model alone (zero ControlNet and

*Equal contributions.

LoRA, or 0C/0L), serving it together with many adapters results in significant delay, which is increasing as more adapters are in use. This delay comes from two sources. **First**, as the desired ControlNets and LoRAs vary across requests, they must be fetched from storage and loaded into GPU memory, introducing non-trivial loading overhead. In our platform, on average each request undergoes loading one ControlNet and one LoRA, which accounts for 37% of the end-to-end serving latency (§3). Given the large population and sizes of these adapters, pre-caching all of them in GPU memory is *infeasible*: our production trace reports nearly 150 distinct ControlNets (∼3 GiB each) and 14,500 LoRAs (hundreds of MiB each) requested by users for the SDXL model [39]. **Second**, ControlNet serving is compute-intensive: on an H800 GPU, adding one ControlNet to the serving workflow extends the generation latency by 1.6 seconds, which is 1.6× longer than serving the base SDXL model alone (Fig. 2). As more ControlNets are utilized, their computational overhead accumulates, leading to a significant latency increase (Fig. 2).

Despite these challenges, efficiently serving a T2I workflow with many adapters has been largely unexplored; prior works [12, 30, 50, 56] primarily focus on improving the serving latency and image quality of a single diffusion model. In this paper, we propose KATZ[1], a system that efficiently serves a T2I workflow where a base diffusion model is augmented with many ControlNets and LoRAs, a typical setting in production deployment. Driven by a characterization study in a production platform (§3), KATZ employs three key designs to optimize the per-request serving latency [9, 12].

ControlNet-as-a-Service. Efficient ControlNet serving requires addressing the GPU loading and computational overhead. Our characterization study reveals that ControlNets exhibit the *skewed popularity*; that is, a small number of ControlNets (9–11%) are invoked frequently by a large number of user requests (95–98%). Caching these popular ControlNets in GPU memory largely eliminates the loading overhead, with only modest memory footprint. To accelerate computation, KATZ *concurrently executes* ControlNet(s) with the base model on *multiple GPUs*, achieving *close-to-ideal speedup* compared to the current sequential execution scheme [50].

KATZ proposes *ControlNet-as-a-Service* to enable ControlNet caching and parallelization. It decouples ControlNets from the base model and deploys them as a *separate, independently scalable service on dedicated GPUs*. It only caches a small number of top popular ControlNets in GPU memory, eliminating the loading overhead by a large degree. To request certain ControlNets, users simply invoke this service, which executes the requested ControlNets in parallel with the base model. This design additionally enables ControlNet sharing, where a single ControlNet can be multiplexed by multiple base models.

[1]Our system is named after Morris Katz, the world's fastest painter according to the Guinness World Record.

Bounded asynchronous LoRA loading. Unlike ControlNet, LoRA is compute-light and LoRA loading is the main performance bottleneck. Given their large populations, LoRA adapters are usually maintained in storage (local disk or remote memory) and must be brought into GPU memory on-demand. Caching top popular LoRAs offers limited benefits as LoRA popularity exhibits a *heavy-tailed distribution* (§3.2).

To address this challenge, we analyze the T2I generation process and find that LoRA computations take effect largely in the *later stages* of the denoising process (§6). Based on this observation, we propose to hide the LoRA loading overhead through *bounded asynchronous loading* (BAL). That is, while the requested LoRAs are being loaded into GPUs, KATZ simultaneously executes the base model to early start the image generation process by *up to K* steps, after which the LoRA adapters must be patched to the base model to continue the remaining generation steps. By tuning the asynchronous bound K, KATZ effectively overlaps LoRA loading with base model execution, while achieving the same image quality as synchronous loading (§9.2).

Latent parallelism for CFG computation. KATZ also exploits parallelization opportunities to accelerate base model execution. As mentioned earlier, T2I generation is essentially a denoising process, where the diffusion model progressively refines a *latent tensor* through multiple steps [19, 32, 39]. Each step performs classifier-free guidance (CFG) [27], where the input latent tensor is duplicated and the two replicas are respectively denoised, one guided by the text prompt (conditioned denoising) and the other unguided (unconditioned denoising); the two denoised latent tensors are then aggregated by computing a weighted sum. As conditioned and unconditioned denoising have no dependency, KATZ parallelizes them on two GPUs. This technique, which we call *latent parallelism*, is also applied to accelerate CFG computation in ControlNet serving. Latent parallelism, together with kernel-level optimizations, collectively accelerate base model execution by 1.8× (§9.5).

We have implemented KATZ on top of HuggingFace Diffusers [50] and evaluated its performance using text prompts from PartiPrompts [64]. Our evaluation encompasses SDXL [39], a popular UNet-based diffusion model [42] in production [12, 46], and two diffusion transformer (DiT) models [19, 32, 38]. Evaluation results demonstrate that KATZ outperforms the state-of-the-art T2I serving systems, including Diffusers, Nirvana [12], and DistriFusion [30], reducing the average serving latency by up to 7.8× and improving the throughput by up to 1.7× (§9.2). To comprehensively assess the quality of the generated images, we engaged 75 human users and confirmed that KATZ produces images of the same quality as Diffusers [50], a lossless baseline. Our contributions are summarized as follows:

- We present the first characterization study in a production T2I platform and identify new challenges of serving base diffusion models with ControlNet and LoRA adapters.

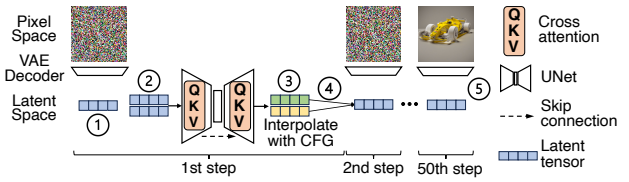


Figure 3: Image generation using a stable diffusion model. Time and token embeddings are ignored for simplicity.

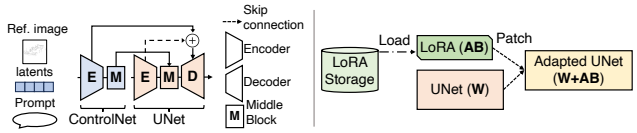


Figure 4: Serving a diffusion model with ControlNet (**Left**) and LoRA (**Right**).

- We propose a holistic approach with three key designs to systematically optimize the T2I serving workflow, including ControlNet-as-a-Service, bounded asynchronous LoRA loading, and latent parallelism for CFG computation.
- We develop KATZ, an optimized serving system for T2I applications that achieves significant speedup without compromising image quality (Fig. 15).

KATZ and the production trace have been open-sourced at https://github.com/modelscope/Katz.

2 Background

In this section, we give a primer to diffusion-based T2I generation and the use of two popular adapters, ControlNet and LoRA, for enhanced generation control.

Diffusion model. A typical stable diffusion model [15, 39, 41] consists of three main components: a text encoder [40], a convolutional UNet model [42], and a decoder-only variational autoencoder (VAE). The model generates images through an iterative denoising process illustrated in Fig. 3. Given a text prompt, the text encoder encodes the prompt into token embeddings. The image generation process then begins by initializing a *latent tensor* filled with *random noise* (①), which is progressively refined by the UNet over multiple denoising steps guided by the token embeddings. To steer the image generation towards the desired outcome, the UNet employs classifier-free guidance [27] (CFG). Specifically, at each denoising step, the UNet duplicates the latent tensor into two replicas (②); one replica is denoised *conditionally*, taking into account the token embeddings, whilst the other is denoised *unconditionally*. Intuitively, the unconditioned latent representation captures the general image distribution, whereas the conditioned representation incorporates specific context given by the text prompt (③). The two latent tensors are then combined by computing a weighted sum, yielding an interpolated latent representation for further refinement in the next step (④). Upon completion of the denoising process, the final interpolated latent tensor is sent to the VAE decoder to render the output image (⑤).

ControlNet. Users often find it challenging to control the details of output images because text prompts alone are insufficient to precisely specify complex layouts, compositions, and shapes. ControlNet addresses this issue by augmenting a base diffusion model with additional input conditions, such as edge maps and depth maps, which specify the desired spatial composition of the generated images. As illustrated in Fig. 1-Center, ControlNet enables users to provide a reference edge map, allowing the base model to generate a Ferrari that adheres to the specified spatial composition.

Fig. 4-Left illustrates a simplified workflow of applying a ControlNet in the image generation process. ControlNet has a similar architecture to the UNet encoder blocks and middle block, with additional zero convolution operators [69]. It is applied to each encoder level of the UNet backbone. In each denoising step (Fig. 3), ControlNet takes as input the text prompt, the encoded reference image, and the latent tensor. The outputs, which contain the processed features of the reference image, are then incorporated into the skip-connections and middle block of the UNet backbone, guiding the image generation process to conform to the reference image. In practice, users can apply multiple ControlNets to a single base model, in which the outputs of these ControlNets are simply summed up and applied to the corresponding backbone UNet blocks [69].

Low-Rank Adaptation (LoRA) is another popular adapter that allows for generating images in a customized style (Fig. 1-Right). LoRA is a parameter-efficient approach to adapting the base model for domain-specific tasks [28]. Specifically, given a pre-trained base model with weight matrix $\mathbf{W} \in \mathrm{R}^{H_1 \times H_2}$, LoRA introduces two low-rank matrices $\mathbf{A} \in \mathrm{R}^{H_1 \times r}$ and $\mathbf{B} \in \mathrm{R}^{r \times H_2}$, where $r \ll \min\{H_1, H_2\}$ is the LoRA rank. One can simply patch the LoRA weights to the base matrix, i.e., $\mathbf{W}' = \mathbf{W} + \mathbf{AB}$, and use the adapted diffusion model $\mathbf{W}'$ to generate stylized output with the desired visual characteristics, as illustrated in Fig. 4-Right.

Other adapters. Our work primarily focuses on ControlNet [69] and LoRA [28], the two most popular adapters in production environments (§3). Meanwhile, there are many emerging adapters [29, 52, 62, 68, 70] introduced by the research community to enhance image generation. These adapters can be broadly categorized into two classes. The first comprises adapters that operate in tandem with the base model during inference, akin to ControlNet. Examples include IP-Adapter [62] and BrushNet [29]. The second class consists of adapters that augment the base model by incorporating parameter-efficient patches, similar to LoRA [28], including Fooocus Inpaint [68] and IC-Light [70]. Because of the similarity, the techniques developed for expedited ControlNet and

Adapters	Number	Service A	Service B
ControlNet	0	0	1.9%
	1	30.5%	25.1%
	2	69.5%	69.9%
	3	0	3.1%
LoRA	0	0.2%	7.2%
	1	8.8%	73.6%
	2	91%	19.2%

Table 1: The distribution of the number of ControlNets and LoRAs used by each request in two production services.

LoRA serving can be easily extended to the two classes of emerging adapters, which we discuss in §10.

Limitations of current T2I serving systems. Diffusion-based T2I workflow serving should provide low latency to allow real-time user interaction to better support multi-round prompt editing and image fine-tuning [9, 12]. DIFFUSERS is the state-of-the-art system that supports T2I serving workflow with ControlNets and LoRAs [50]. It generates images of original quality but incurs significant latency overhead, especially as more adapters are being used (Fig. 2). Recently proposed T2I serving systems, such as NIRVANA [12] and DISTRIFUSION [30], focus solely on optimizing base diffusion model inference. NIRVANA [12] proposes to skip the first κ denoising steps by using a pre-cached image generated from a similar prompt to replace the randomly initialized noise latent (Fig. 3). DISTRIFUSION [30] uses multiple GPUs to a diffusion model to accelerate image generation. It splits the latent (② in Fig. 3) into small patches and parallelizes the computation on multiple GPUs, outperforming tensor parallelism [44] with improved communication efficiency [30]. We will show in §9 that these systems, though efficient in base model inference, fall short in workflow serving with many adapters, resulting in extended latency and quality loss.

3 Characterization Study

In this section, we present a characterization study on a 20-day workload trace collected in May and June 2024 on a production platform[2]. The trace contains more than 500k inference requests to two core T2I services for online retailing applications. Our characterization not only reflects the deployment scenarios of diffusion models in production, but also reveals the inefficiency of current T2I serving systems.

3.1 ControlNet Characterization

Prevalence. Table 1 shows the distribution of the number of ControlNets utilized by each request in two services. ControlNet is used by almost all requests for image generation control; approximately 70% of these requests utilize two or more ControlNets simultaneously.

[2] https://modelscope.cn/datasets/mental2008/T2I-Model-Serving-Request-Trace

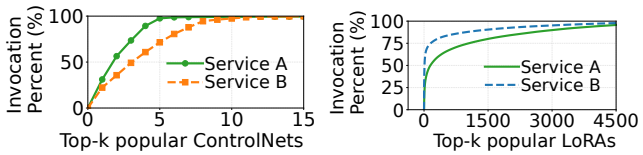


Figure 5: **Left**: ControlNet has a small population and exhibits a skewed popularity; the long tail of the graph is truncated for a better presentation. **Right**: LoRA has a large quantity and exhibits a long-tailed distribution in popularity.

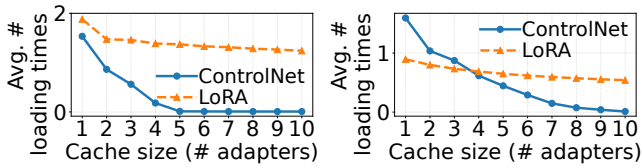


Figure 6: Configuring a larger LRU cache effectively eliminates the adapter loading overhead for ControlNets, but not for LoRAs. **Left**: Service A; **Right**: Service B.

Skewed popularity. Compared to a large quantity of requests, only 141 ControlNets are used in two services, where Service A offers 47 distinct ControlNets and Service B provides 94. These ControlNets exhibit a severe skewness in access frequency. As shown in Fig. 5-Left, the top-5 most popular ControlNets (11% in population) account for 98% of total invocations in Service A. When it comes to Service B, the top-8 most popular ControlNets (9% in population) contribute to 95% of total invocations.

The need for ControlNet caching. ControlNets are large in size (3 GiB each) and usually maintained in remote storage, introducing significant loading overhead. Given that ControlNets have a limited quantity and skewed popularity, caching a small number of top popular ControlNets in GPU memory effectively eliminates the loading overhead for most requests. To illustrate this, we configure an LRU cache of varying size for ControlNet caching. We replay the trace and measure the average number of times that the desired ControlNets are not resident on GPU and must be fetched from storage (i.e., cache miss) when serving two consecutive requests that desire different ControlNets. As illustrated in Fig. 6 (blue curves), caching only a handful of top popular ControlNets is sufficient to eliminate the loading overhead for both services (top-5 for Service A and top-8 for Service B). Similar results are observed if we replace the LRU cached with an LFU (Least Frequently Used) cache.

Computational overhead. ControlNet is compute-heavy as it shares a similar architecture to the UNet encoder and middle block (§2). As illustrated in Fig. 2 (blue bars), augmenting the base diffusion model with one ControlNet increases the serving latency by 1.6× (up from 2.9 seconds to 4.5 seconds). As more ControlNets are utilized, their computational overhead accumulates because current T2I serving systems [50] *se-*

quentially compute the outputs of the requested ControlNets before executing the base model at each denoising step.

3.2 LoRA Characterization

Prevalence. Similar to ControlNet, the majority of T2I requests utilize one or two LoRAs to stylize the generated image, as summarized in Table 1. Specifically, over 90% of requests in Service A desire two LoRAs, while nearly 93% of requests in Service B demand at least one LoRA.

Long-tailed popularity distribution. Compared to ControlNets, LoRAs have a significantly larger population but smaller sizes. Our trace reports 6,980 distinct LoRAs for Service A and 7,463 LoRAs for Service B. Each LoRA is a few hundreds of MiB. Unlike ControlNets, the popularity of LoRAs follows a *long-tailed distribution*; that is, a significant portion of LoRA invocations are contributed by a large number of *less popular* adapters, as illustrated in Fig. 5-Right. In Service A, 6,968 LoRAs (99.8% of the total 6,980) each account for less than 1% of invocations. Similarly, in Service B, 7,449 LoRAs (99.8% of the total 7,463) each contribute to less than 1% of invocations.

Ineffective LoRA caching. Given the long-tailed popularity distribution of LoRAs, caching the top popular LoRA adapters offers *limited benefits*. To demonstrate this, we configure an LRU cache of varying sizes for LoRA caching. We replay the trace and measure the average number of times that the desired LoRAs are not available on GPU and must be brought from storage (i.e., cache miss) when serving two consecutive requests that demand different LoRAs. As illustrated in Fig. 6 (orange curves), configuring a larger LRU cache only slightly reduces the loading overhead caused by cache misses. We also measure the cache misses with an LFU cache and observe similar results. For Service B with a cache size of 10, LRU and LFU achieve comparable average loading times due to cache misses, at 0.54 and 0.57, respectively. Intuitively, LFU favors LoRAs with high long-term popularity and LRU prioritizes LoRAs with high short-term popularity. However, due to the dynamic nature of LoRA invocation patterns and their long-tailed distribution, predicting invocation traffic remain challenging, making approaches fall short.

As another evidence, Fig. 7 depicts a scatter plot illustrating the number of requests served on each node in the trace and the number of *unique* LoRAs utilized by those requests on that node. We observe the linear correlation between the two numbers, invalidating the benefits of LoRA caching. Our production system hence chooses not to cache LoRAs but loads them from storage in an on-demand fashion.

LoRA loading and patching overhead. Compared to ControlNets, LoRAs are compute-light and LoRA serving is bottlenecked by the loading and patching overhead (Fig. 4-Right). Our measurements show that fetching two LoRAs (total size of approximately 800 MiB) from a remote distributed cache

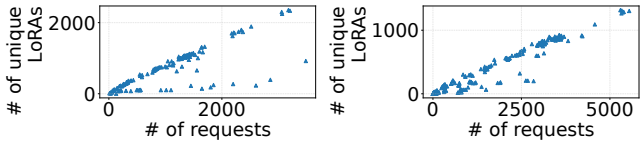


Figure 7: Scatter plots illustrating the number of requests received on each worker node (X-axis) against the number of unique LoRAs required by those requests (Y-axis). **Left**: Service A; **Right**: Service B.

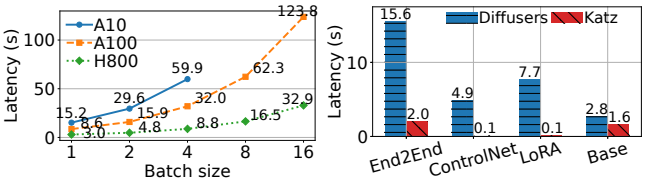


Figure 8: **Left**: Ineffective Batching for SDXL inference. **Right**: End-to-end latency and component breakdowns for a 3**C**/2**L** request, using Diffusers [50] and KATZ on H800.

takes more than one second, delaying the base model serving by 34% (up from 2.9 seconds to 3.9 seconds). In addition, simply patching LoRA weights to the base model, as implemented in existing systems [37, 50], incurs high overhead, which we elaborate in §6.

3.3 Characterizing Base Model Serving

Currently, UNet-based diffusion models, such as SDXL [39], are predominately deployed to handle the majority of requests in production platforms. These models are supported by a plethora of well-trained ControlNets [69] and LoRAs [28]. In the meantime, there is an emerging trend of deploying transformer-based diffusion models (DiT) [19, 32], but the development of corresponding adapters remains lagging behind at the moment. In this paper, we primarily focus on UNet-based models; our observations and optimization designs also apply to the transformer backbone.

Ineffectiveness of batching. Diffusion model serving is computationally intensive, as evidenced by our experiments with varying batch sizes for a standard SDXL model [39] on NVIDIA A10, A100, and H800 GPUs (Fig. 8-Left). Across all three GPUs, doubling the batch size results in an approximately 2× in serving latency, indicating minimal benefits from batching. In fact, generating a single image already saturates the computational resources of a high-end GPU. According to [30], the computational cost of generating a 1024×1024 image with SDXL requires 676T FLOPS, where the token length involved in SDXL's transformer computations can reach up to 4096, resulting in a high computational load. Consequently, production T2I services typically configure a constant batch size of 1 to minimize serving latency. That being said, we expect more performance gains from

Notation	Description
S	Number of denoising steps
M	Number of ControlNets used
N	Number of LoRAs used
$T_{\mathrm{C}_i}^{\mathtt{Load}}$, $T_{\mathrm{C}_i}^{\mathtt{Comp}}$	Time to load and compute **C**ontrolNet C_i
$T_{\mathrm{L}_j}^{\mathtt{Load}}$, $T_{\mathrm{L}_j}^{\mathtt{Patch}}$	Time to load and patch **L**oRA L_j
$T_{\mathrm{Enc}}^{\mathtt{Comp}}$	Time to compute the text encoder inference
$T_{\mathrm{VAE}}^{\mathtt{Comp}}$	Time to compute the VAE decoder inference
$T_{\mathrm{B}}^{\mathtt{Comp}}$	Time to compute the base model inference
$T_{\mathrm{B}^*}^{\mathtt{Comp}}$	Optimized base model inference time

Table 2: Notations used to model T2I inference latency.

batching when using more powerful GPUs.

Dominated CFG computation. To understand the computation of the base diffusion model, we break down its execution and find that over 90% of the inference time is spent on CFG computation (§2). Current CFG implementation employs *latent batching*. That is, at each denoising step, the latent tensor is duplicated and the two replicas are fed into the base model to perform conditional and unconditional denoising in *one batch* on a GPU. However, as the two denoising operations are compute-heavy, batch-executing them yields minimal benefits. In fact, latent batching results in up to 1.7× slowdown in base model serving compared to our optimized design (§9.5).

3.4 Latency Overhead of Current Systems

To sum up, current T2I systems serve the base model and the associated adapters in a *sequential execution pipeline*. Specifically, assume a request utilizing m ControlNets and n LoRAs. Upon request arrival, the system loads all the desired ControlNets and LoRAs into GPU memory, followed by patching the n LoRAs to the base model. The system then encodes the text prompt and proceeds to the denoising process in S steps. At each step, it sequentially executes the m ControlNets and the LoRA-patched base model to generate a latent representation. The final latent representation is then sent to the VAE decoder to generate the output image. The end-to-end workflow serving latency is given by Eq. (1), where the notations are defined in Table 2:

$$T = \underbrace{\sum_{i=1}^{M} T_{\mathrm{C}_i}^{\mathtt{Load}} + \sum_{j=1}^{N} (T_{\mathrm{L}_j}^{\mathtt{Load}} + T_{\mathrm{L}_j}^{\mathtt{Patch}})}_{\text{time to load and patch adapters}} + \underbrace{T_{\mathrm{Enc}}^{\mathtt{Comp}} + (\sum_{i=1}^{M} T_{\mathrm{C}_i}^{\mathtt{Comp}} + T_{\mathrm{B}}^{\mathtt{Comp}}) \times S + T_{\mathrm{VAE}}^{\mathtt{Comp}}}_{\text{computation time for multi-step image generation}}. \quad (1)$$

Our characterization identifies efficiency issues concerning ControlNet loading ($\sum_{i=1}^{M} T_{\mathrm{C}_i}^{\mathtt{Load}}$), sequential ControlNet execution ($\sum_{i=1}^{M} T_{\mathrm{C}_i}^{\mathtt{Comp}}$), slow LoRA loading ($\sum_{j=1}^{N} T_{\mathrm{L}_j}^{\mathtt{Load}}$) and patching ($\sum_{j=1}^{N} T_{\mathrm{L}_j}^{\mathtt{Patch}}$), and inefficient latent batching in base model execution ($T_{\mathrm{B}}^{\mathtt{Comp}}$), collectively accounting for up to 99% of end-to-end serving latency (Fig. 8-Right).

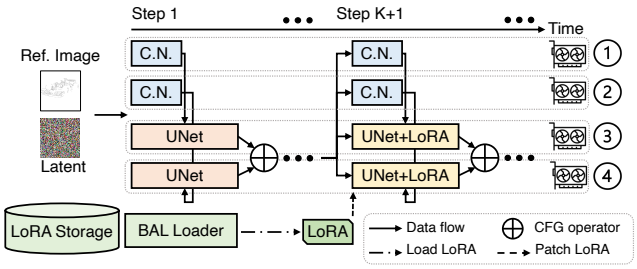


Figure 9: KATZ utilizes 4 GPUs to serve a base model with 1C/1L, two for the base and two for the **C**ontrol**N**et (**C.N.**).

4 Design Overview

To address the efficiency issues identified in §3, we propose KATZ, a system that efficiently serves a T2I workflow with many adapters. KATZ employs three key designs. **First**, to reduce the overhead of ControlNet loading and computation, KATZ introduces ControlNet-as-a-Service, enabling ControlNet caching and parallelization in a unified design (§5). **Second**, to mitigate LoRA loading and patching overheads, KATZ overlaps LoRA fetching with base model execution for a bounded steps (bounded asynchronous LoRA loading or BAL), and uses an efficient method to quickly patch LoRA adapters (§6). **Third**, KATZ accelerates CFG computation by concurrently executing conditional and unconditional denoising operations on two GPUs, a technique called *latent parallelism*, together with kernel-level optimizations (§7). Fig. 9 illustrates an example of serving a base diffusion model with one ControlNet and one LoRA (1C/1L). Latent parallelism is applied to accelerate both the base model and the ControlNet, each using two GPUs; LoRA is asynchronously loaded, in parallel with initial base model execution for up to K steps before patching to hide the loading overhead.

In practice, the loading and computational overhead associated with ControlNets and LoRAs are virtually eliminated in KATZ (Fig. 8-Right), reducing the end-to-end workflow serving latency from Eq. (1) to

$$T = T_{\mathrm{Enc}}^{\mathtt{Comp}} + T_{\mathrm{B}^*}^{\mathtt{Comp}} \times S + T_{\mathrm{VAE}}^{\mathtt{Comp}}. \quad (2)$$

Collectively, our designs achieve up to 7.8× reduction in latency (Fig. 8-Right) and 1.7× throughput improvement, without compromising image quality (more in §9.2).

While we present KATZ primarily based on diffusion models with the UNet backbone, our designs also apply to the recently proposed diffusion transformers (DiTs) [19, 32, 38].

5 ControlNet-as-a-Service

For ControlNet serving, the main overhead comes from adapter loading and computation, which is tightly coupled with base model execution: as more ControlNets are utilized, their overheads accumulate (§3.1). We address this problem with a principled ControlNet-as-a-Service approach.

ControlNet-as-a-Service. Unlike existing systems, KATZ decouples ControlNet execution from the base diffusion model and deploys ControlNets as a *separate, independently scaled service* on *dedicated* GPUs. This design provides three benefits. **First**, by deploying ControlNets as an independent service, multiple ControlNets can execute in parallel with the base model on multiple GPUs and synchronize only at the end of each denoising step (details to come), effectively hiding their computational overhead. **Second**, the service can easily track the popularity of each ControlNet and cache a small number of top popular ones in GPU memory, which is sufficient to eliminate the loading overhead given their skewed popularity as identified in §3.1. **Third**, the separated ControlNet service can be shared among many T2I workflows in a multi-tenant system, enabling a single ControlNet to be multiplexed by many base models and scaled out according to the request load.

ControlNet parallelization. Without changing the data flow between ControlNets and the base model, KATZ partitions the entire compute graph of image generation into a *serial* part and a *parallel* part. For the UNet-based SDXL model [39], the *serial* part consists of the one-time computation of the text encoder, one-time computation of the VAE decoder, and UNet decoder computations of denoising steps (§2). The *parallel* part consists of the computations of UNet's encoder and ControlNet(s) at each denoising step (Fig. 4-Left). KATZ distributes the computation of the *parallel* part across multiple GPUs. As illustrated in Fig. 9, it deploys the base UNet model on one set of GPUs and each ControlNet on a different set, operated as a separate service. At each denoising step, the UNet encoder and ControlNet(s) initiate computation concurrently. Upon completion of the UNet middle block inference, the UNet decoder *synchronously* awaits the outputs from ControlNet(s) before its computation, thereby preserving the original data dependencies (Fig. 4-Left). The ControlNet(s) then becomes idle for next invocation.

Performance analysis and optimization. To achieve maximum speedup for parallel computing, it is important to (1) balance the load of concurrent computing tasks and (2) minimize their communications to reduce the synchronization overhead. The first requirement is naturally met in ControlNet parallelization: given that ControlNet shares the same model architecture as the UNet's encoder block and middle block, with the only difference being the additional zero convolution operators [69], their computation load is well balanced, leading to almost the same execution time when running on homogeneous GPUs. Regarding the second requirement, we measure a medium to low data transfer between the base model and the associated ControlNet, e.g., only 108 MiB for SDXL. With high-speed interconnect, such as NVLink [7] and InfiniBand [6], the communication overhead is negligible (e.g., less than 1 ms for SDXL over NVLink). In this scenario, ControlNet parallelization achieves 1.42× speedup

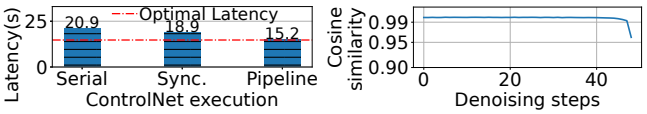


Figure 10: **Left**: Image generation latency using various ControlNet execution schemes on AWS `g5.2xlarge` instances with 10 Gbps network bandwidth (Serial: no parallelization; Sync: synchronous ControlNet parallelization; Pipeline: pipelined asynchronous parallelization). **Right:** Average cosine similarities of ControlNet outputs between two adjacent steps, which are calculated across the channel dimension.

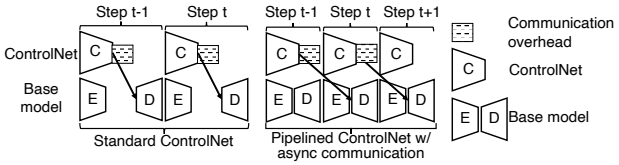


Figure 11: An illustration of ControlNet parallelization (**Left**) and its asynchronous pipeline implementation over slow links (**Right**). The middle blocks are omitted for simplicity.

over NVLink and 1.34× speedup over InfiniBand 400 Gbps link, either on A100 or H800 GPUs, closely matching the ideal speedup of 1.45× given by Gustafson's law [24].

However, when high-speed GPU interconnections are unavailable, communication may become a bottleneck (Fig. 10-Left). In this scenario, we propose *pipelined asynchronous parallelization* to hide the communication overhead. Our key observation is that the output tensor generated by a ControlNet in two adjacent denoising steps are *nearly identical*, with cosine similarity over 0.99 almost all the time, as illustrated in Fig. 10-Right. Based on this observation, we can relax the synchronization requirement between the base model and ControlNets to establish an *asynchronous pipeline*, while still achieving the same image quality. That is, at each denoising step t, the base model performs computation based on the *stale* ControlNet output generated at step $t-1$, which have already been transferred to the base model's GPU during the previous step. Fig. 11 illustrates ControlNet parallelization and its asynchronous pipeline implementation over slow links, where the latter achieves close-to-ideal speedup in our experiments (Fig. 10-Left). We will show in §9.3 that slightly relaxing the synchronization requirement for ControlNet parallelization leads to no quality loss.

Applicability to DiT. ControlNet parallelization is a generic technique that also applies to the emerging DiT-based models, such as SD3 [19] and Hunyuan-DiT [32]. This is because ControlNets for the DiT backbone share the same model architecture as the base model and their data dependencies are analogous to that of their UNet counterparts. One can hence expect the same efficacy for DiT-based models (§9.7).

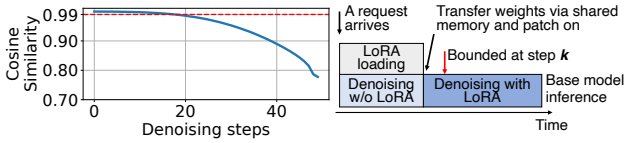


Figure 12: **Left**: Cosine similarities between the latents generated with LoRA and those without LoRA at each denoising step. **Right**: Bounded asynchronous LoRA loading.

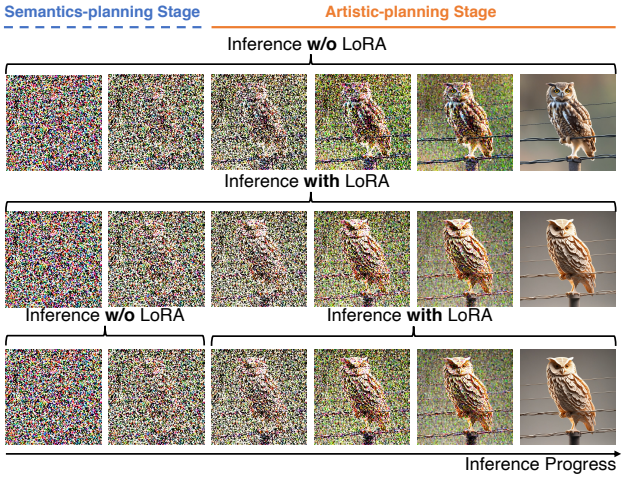


Figure 13: Images generated every 10 steps from the first step using a papercut LoRA [48]. **Prompt**: an owl standing on a wire. **Top**: Inference w/o LoRA. **Middle**: Inference w/ LoRA. **Bottom**: Inference w/o LoRA during semantics-planning stage and w/ LoRA during artistic-planning stage.

6 Efficient LoRA Serving

Motivation. As discussed in §3.2, LoRAs are stored in an external storage (e.g., local disk or remote cache). To apply a LoRA for stylizing the image generation, the system first fetches the adapter from storage and loads it into host memory. After that, it patches the adapter to the base diffusion model by merging its weights with the parameters of the base model. Both LoRA loading and patching incur significant overhead, which accumulates as more LoRAs are utilized (Fig. 2). We address these two problems in this section.

Bounded asynchronous LoRA loading (BAL). We analyze image generation with LoRA and observe a general trend: the effect of LoRA computation is initially imperceptible but, after certain steps, becomes increasingly significant. We empirically validate this by executing the image generation process twice, with and without LoRA. We calculate the cosine similarity between the output tensors generated with and without LoRA at each denoising step and depict the result in Fig. 12-Left. The cosine similarity consistently exceeds 0.99 in initial steps, indicating that LoRA exerts minimal effects during this stage. However, after a certain step (20 steps in this case), the similarity starts to plunge, a turning point at which LoRA effects kick in and become increasingly significant moving forward. Fig. 13 further visualizes the denoising process of image generation with and without LoRA. The initial denoising steps constitute a *semantics-planning stage* [12, 34, 45, 72], wherein the model determines the image composition and layout, generating visual semantics aligned with the text conditions [12, 72]. LoRA computation is less relevant and can be safely excluded during this stage (Fig. 13-Bottom). The remaining generation steps constitute an *artistic-planning stage* with image details gradually emerging, e.g., color, texture, and artistic style [12, 72]. LoRA plays a crucial role in this stage and must be included (Fig. 13-Bottom).

Driven by this observation, KATZ proposes overlapping LoRA loading with base model execution in the initial semantics-planning stage, as shown in Figures 12-Right and 13-Bottom. When a request arrives, KATZ asynchronously loads the requested LoRA(s). In the meantime, it *early-starts* the base diffusion model inference to perform image generation without LoRA. To ensure no quality loss, KATZ imposes an asynchrony bound K for overlapping LoRA loading with base model inference: in the worst case, if the requested LoRAs have not been loaded at the $(K+1)^{\text{th}}$ denoising step, the base model waits until the loading completes. The loaded LoRAs are then patched to the base model to continue the remaining generation steps.

Determining the asynchrony bound. Ideally, we should choose a large asynchrony bound K to maximally overlap LoRA loading with base model inference, subject to no quality loss. KATZ employs a profiling method to optimally determine the asynchrony bound K. Given a base model and a LoRA, KATZ offline calculates the cosine similarity between the latent tensors generated with and without LoRA at each denoising step, akin to those in Fig. 12-Left. It then sets K to the step at which the similarity starts to drop below a predefined threshold, e.g., 0.99. In our evaluation, we profile and set $K = 10$. Our experiment on A100 and H800 GPUs shows that LoRA loading completes mostly before the predefined step K, effectively hiding the adapter loading overhead (c.f. 0C/0L and 0C/1L in Fig. 2).

Efficient LoRA patching. Existing systems [50] use the PEFT [37] framework to merge LoRA weights with base model parameters. For a layer in the base stable diffusion model that will be patched with LoRA, PEFT *creates* a new LoRA layer to *replace* the original layer in the base model. The newly created LoRA layer augments the corresponding base model layer by incorporating LoRA weights and configurations. However, this *create-and-replace* operation incurs high overhead, taking 2 seconds for a LoRA of 341 MiB and occupying additional GPU memory. Although maintaining a separate copy of LoRA weights in the new augmented layer facilitates convenient LoRA training and efficiently patching off LoRA weights after image generation, we find it unnecessary. As a serving system, KATZ does not require the capability to

support LoRA training. Besides, our characterization study reveals that the time interval between two consecutive requests is long enough (over 1 second) to patch off LoRAs. Therefore, KATZ chooses to merge LoRA weights with base model parameters *in place*. This design eliminates the latency overhead caused by the *create-and-replace* operation and saves GPU memory without storing separate LoRA weights.

Applicability to DiT. Our design for efficient LoRA loading and patching is agnostic to the architecture of the base model, and can hence be applied directly to accelerate image generation in DiTs [19, 32].

7 Optimized Base Model Execution

With the overhead of adapters effectively addressed in §5 and §6, we now turn to accelerating base model inference, the last bottleneck in image generation. Given that diffusion model serving is computationally intensive that saturates a high-end GPU even with a small batch size of 1 (§3.3), we explore parallelization opportunities to accelerate CFG computation on multiple GPUs. We also incorporate kernel-level optimizations tailored for diffusion model computation and its interaction with adapters to further enhance performance.

Latent parallelism for diffusion model. As explained in §2, diffusion model uses the CFG technique to better align image generation with textual descriptions at each denoising step, where an input latent tensor is duplicated and undergoes two denoising operations, one conditioned on the texts and the other unconditionally. The two latents are then combined by computing a weighted sum, yielding an interpolated latent as the output. As conditioned and unconditioned denoising have no dependency, they can be performed in parallel on two GPUs, which we call *latent parallelism*. As illustrated in Fig. 9, KATZ maintains two instances of the base diffusion model on two homogeneous GPUs. At each denoising step, KATZ duplicates the input latent tensor (② in Fig. 3) and feeds the two replicas into the two base model instances to perform conditioned and unconditioned denoising in parallel. As the two computations have balanced load, they complete at the same time, and their outputs are then interpolated as a weighted sum through a synchronous communication.

The simple yet effective latent parallelism strategy can accelerate the base diffusion model inference of a request by 1.4–1.9×, depending on the GPU capability and model size. The performance gains are more pronounced with larger base models and lower-end GPUs (details in §9.5). Latent parallelism incurs little overhead because 1) computations on different GPUs are uniform and finish at almost the same time, and 2) the communication overhead is minimal, mainly comprising the transfer of a small latent ($<$ 1 MiB). However, the speedup achieved by latent parallelism may come at the expense of per-GPU throughput when the denoising computation of a single latent does not saturate a GPU. Latent parallelism also applies to the emerging DiT models as they employ the same CFG technique in image generation.

Compatibility with adapter optimizations. Latent parallelism can be naturally applied to accelerate CFG computation in ControlNet serving, as ControlNets are architecturally similar to the base model (§5). Fig. 9 illustrates an example where KATZ utilizes 4 GPUs to serve a base diffusion model and one ControlNet, both with latent parallelism. Meanwhile, the base model also utilizes BAL to overlap LoRA loading with base model computation at each denoising step.

Kernel-level optimizations. KATZ includes several kernel-level optimizations to further enhance performance, including a customized CUDA Graph [21] implementation and specialized CUDA kernels tailored to diffusion models. CUDA Graph is particularly suitable for T2I inference, as it uses a constant batch size of 1 (§3.3). Given the nearly homogeneous requested image resolutions in our production platform, we only need to maintain a small number of CUDA Graphs resident in GPU memory. Furthermore, we adapt the original CUDA graph to accommodate ControlNet parallelization. Specifically, we tailor the base model and segment it as distinct CUDA Graphs according to its data dependencies with ControlNets. Beyond existing optimized attention kernels [1], KATZ provides kernel optimizations specific to UNet-based diffusion models, including an optimized `GEGLU` activation operator by fusing `GELU` and matrix multiplication operations, and a fused operator that combines `GroupNorm` and `SiLU` operators to mask the latter's overhead.

8 Implementation

We have implemented KATZ on top of Diffusers [50], a PyTorch-based diffusion model inference framework that integrates state-of-the-art model optimization techniques. KATZ is written in 5.5k lines of Python and 2.4k lines of C++/CUDA code, where ControlNet-as-a-Service, BAL, and latent parallelism are implemented in Python, whilst customized CUDA operators are written in C++/CUDA. KATZ performs LoRA loading in separate processes and utilizes shared memory to transfer LoRA weights from the loading processes to the base model serving process for efficient (parallel) data transfer.

9 Evaluation

We evaluate KATZ's performance in terms of serving latency and image quality. Evaluation highlights include:

- KATZ achieves efficient serving performance without degrading image quality, accelerating T2I generation by up to 7.8× compared with state-of-the-art baselines (§9.2).
- KATZ's ControlNet-as-a-Service design decouples ControlNets from the critical path of a T2I workflow, achieving a *close-to-ideal* speedup (§9.3).

- KATZ effectively eliminates the LoRA loading and patching overheads, leading to consistent serving latency that matches LoRA-free execution (§9.4).
- With latent parallelism and kernel optimizations, KATZ achieves 1.7× speedup in base model inference (§9.5).
- KATZ increases the per GPU-minute throughput by up to 1.7× (§9.6) and generalizes to DiT-based models (§9.7).

9.1 Experimental Setup

Model and serving configurations. Unless otherwise specified, we use SDXL [39] as the base diffusion model. SDXL and its variants are widely deployed in production systems [12], including ours, and have comprehensive support for adapters. We use the default settings to generate images, where the number of denoising steps is set to 50 and the image resolution is 1024×1024. The ControlNets [3] and LoRAs [47–49] used are provided by HuggingFace. By default, we run experiments on NVIDIA H800 GPUs and configure synchronous ControlNet parallelization for KATZ (§5).

Baselines. We consider the following three baselines:

- DIFFUSERS is a standard baseline for T2I workflow serving with support of ControlNets and LoRAs [50].
- NIRVANA [12] is a strong baseline that skips the first κ denoising steps by using a pre-cached image given by a similar prompt. We use two configurations, $\kappa = 10$ (NIRVANA-10) and more aggressively $\kappa = 20$ (NIRVANA-20).
- DISTRIFUSION [30] is a strong baseline that utilizes multiple GPUs to accelerate base diffusion model inference. We extend it to support ControlNets on multiple GPUs. For a fair comparison, we configure DISTRIFUSION to use no fewer GPUs than KATZ.

Serving metrics. Our evaluation mainly concerns two metrics, *serving latency* and *image quality*. For serving latency, we measure the end-to-end latency of generating an image based on a given text prompt. For image quality, we use the following quantitative metrics, which are considered essential and widely used in measuring image quality [12,39,64,71,73].

- CLIP [25,40] score evaluates the alignment between generated images and their corresponding text prompts. A higher CLIP score indicates better alignment (↑).
- Fréchet Inception Distance (FID) score [26] calculates the difference between two image sets, which correlates with human visual quality perception [12]. A low FID score means that two image sets are similar (↓).
- Structural Similarity Index Measure (SSIM) score [55] measures the similarity between two images, with a focus on the structural information in images. A higher SSIM score suggests a greater similarity between the images (↑).

Like [12,39], we conducted a user study with 75 participants to evaluate the image quality based on their visual perception.

Workloads. KATZ is designed to reduce the serving latency of text-to-image requests associated with many adapters. To evaluate this, we measure per-request serving latency across different adapter configurations on provisioned instances, eliminating the impact of request queuing and model scaling due to load spikes. We configure the batch size to 1 due to the limited batching effect (§3.3). The requests use text prompts in Google's PartiPrompts (P2) [64], a popular benchmark for image generation tasks [36,39,64]. P2 provides a rich set of prompts, including simple and complex prompts across various categories (e.g., Animals, Scenes, and World Knowledge) and challenging aspects (e.g., Detail, Style, and Imagination). We serve each request with several adapters, following our production trace (Table 1).

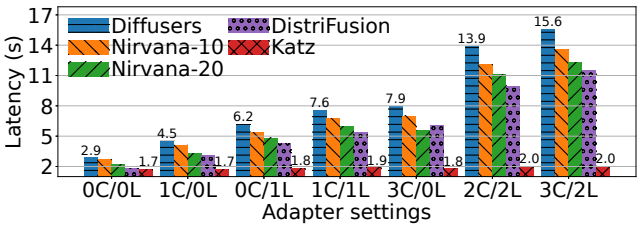


Figure 14: End-to-end serving latency with m ControlNets and n LoRAs (mC/nL). GPU allocation for DISTRIFUSION and KATZ is as follows: 0C/0L and 0C/1L configurations use two GPUs, 1C/0L and 1C/1L use four GPUs, and 3C/0L and 3C/2L use eight GPUs. For the 2C/2L configuration, KATZ uses six GPUs, while DISTRIFUSION requires eight GPUs. All other baselines use a single GPU across all configurations.

9.2 End-to-End Performance

Serving latency. We vary the number of adapters in a workflow and compare the average serving latency of KATZ and each baseline in Fig. 14. KATZ outperforms existing systems in all settings, achieving up to 7.8× speedup over DIFFUSERS and 5.7× speedup over DISTRIFUSION, the strongest baseline. Compared to the four baselines, KATZ exhibits a largely stable latency as more adapters are utilized in a workflow, the result of its design that decouples adapters from the critical path. In the absence of adapters (0C/0L), KATZ's latent parallelism and kernel optimizations (§7) accelerate base model inference by 1.7× and 1.3× compared to DIFFUSERS and the aggressive NIRVANA-20, respectively. Note that KATZ even outperforms DISTRIFUSION, a system that also parallelizes diffusion model inference on multiple GPUs, by 1.1× thanks to the kernel optimizations. We further compare DISTRIFUSION's parallelism with KATZ's ControlNet parallelism with the same number of GPUs in the settings of 1C/0L and 3C/0L. KATZ outperforms as the ControlNet parallelism design is based on the computation and communication characteristics of ControlNets while DISTRIFUSION is adapter agnostic.

Image quality. We compare the quality of images generated by KATZ and each baseline. Since our ControlNets-as-a-

Figure 15: Examples of images generated by each baseline.

LoRA Setting	System	CLIP(↑)	FID (↓)	SSIM (↑)
One LoRA: Papercut [48]	DIFFUSERS	34.1	-	-
	NOLORA	32.9	11.4	0.63
	NIRVANA-10	33.5	9.5	0.45
	NIRVANA-20	33.7	10.9	0.44
	DISTRIFUSION	34.0	1.7	0.86
	KATZ (ours)	34.1	2.1	0.83
Two LoRAs: Filmic [47] + Photography [49]	DIFFUSERS	34.2	-	-
	NOLORA	31.3	13.4	0.67
	NIRVANA-10	33.3	9.0	0.51
	NIRVANA-20	32.8	9.4	0.47
	DISTRIFUSION	34.1	2.9	0.86
	KATZ (ours)	34.1	3.1	0.82

Table 3: Quantitative evaluation on image quality.

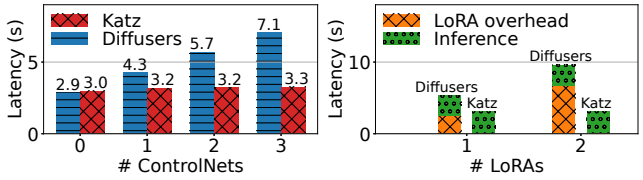


Figure 16: **Left**: Microbenchmark on ControlNets. **Right**: Microbenchmark on LoRAs.

Service design with synchronous parallelism makes no modification to image generation, we focus on evaluating the LoRA effects. Two settings are considered: the first uses a single LoRA to generate images in a papercut style [48], while the second employs two LoRAs to generate images in a combination of William Eggleston photography style and filmic style [47, 49]. We use the prompts in P2 [64] that emphasize vivid details in images.

***1)* Quantitative evaluation.** Table 3 shows the CLIP, FID, and SSIM scores achieved by each baseline. CLIP scores measure the alignment between generated images and their corresponding prompts [8]. The results indicate that DISTRIFUSION and KATZ exhibit good performance in terms of alignment, rivaling DIFFUSERS's standard lossless workflow.

FID and SSIM scores focus on comparing the generated images with the standard images ("ground truth"). Therefore, we use the images generated by DIFFUSERS as the ground truth, as it represents the original T2I serving workflow. We include the NOLORA baseline for reference, which utilizes no LoRA in image generation. In Table 3, DISTRIFUSION and KATZ achieve comparable performance and outperform others, indicating that they generate images highly similar to those generated by DIFFUSERS. NIRVANA-10 and NIRVANA-20 fall short because they generate an image based on the contents of a cached image, which is selected only based on the prompt similarity. Even with the same prompt, the visual contents in cached images can be drastically different (see Fig. 1-Left) and may not align with the style of LoRAs. Fig. 15 presents real examples generated by each baseline, where images generated by DIFFUSERS, KATZ, and DISTRIFUSION are almost visually indistinguishable, whereas NIRVANA-10 fails to match the quality of DIFFUSERS.

***2)* Qualitative evaluation.** We conducted a user study involving 75 participants to compare the quality of images generated based on human visual perception. The participants are mainly university students. We consider DIFFUSERS, NIRVANA-10, and KATZ in this part since quantitative evaluation shows image quality of NIRVANA-20 is not nearly as good as that of other baselines and DISTRIFUSION can match the quality of DIFFUSERS. Inspired by Chatbot Arena [74], we constructed an online arena that *randomly* presents a pair of two images to users, offering four options: both images are acceptable, neither is acceptable, image 1 is acceptable, or image 2 is acceptable. Participants made their selections based on both the degree of image alignment with the prompt and their subjective aesthetic preferences. We collected over 1.2k data points. The findings indicate that KATZ is capable of producing images of the same quality as DIFFUSERS, both with 70% acceptance rate. In contrast, NIRVANA-10's acceptance rate is below 50% due to its skipped denoising steps and not considering the impact of adapters during its prompt match process.

9.3 ControlNet-as-a-Service

We next evaluate the performance of KATZ's ControlNets-as-a-service design, with all other optimizations disabled. We compare KATZ with DIFFUSERS, as NIRVANA and DISTRIFUSION lack specialized designs for ControlNets. Fig. 16-Left illustrates the serving latency achieved by DIFFUSERS and KATZ, where KATZ achieves up to 2.2× speedup by distributing ControlNets computation across multiple GPUs. Note that in this case, KATZ employs synchronous ControlNet parallelization, generating identical images as DIFFUSERS.

Figure 17: Sample images generated using sync. and async. ControlNet scheme, along with images without ControlNets.

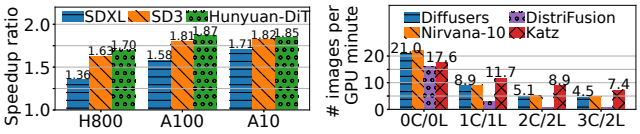


Figure 18: **Left**: Speedup ratio of latent parallelism on different GPU types and base models. **Right**: Serving throughput with m ControlNets and n LoRAs (mC/nL).

To further analyze KATZ's speedup, we apply Gustafson's law [24], which quantifies the theoretical speedup in execution time for a task that benefits from parallel computing. Let N denote the number of processors, and let s and p represent the time fractions spent executing the serial and parallel parts of the program ($s+p=1$). The theoretical speedup S from parallel computing is $S=s+pN$ [24]. In the context of T2I generation with ControlNets, the serial parts comprise the computation of decoder blocks in UNet, while the parallel parts include the computation of UNet's encoder blocks together with middle block and ControlNets (§5). When using three ControlNets, the serial parts account for $s=0.55$ and the parallel parts take $p=0.45$, leading to a theoretical speedup of 2.35×. KATZ achieves 2.2× speedup and is near optimal.

Pipelined asynchronous parallelization. We next evaluate the performance of pipelined asynchronous ControlNet parallelization in the absence of high-speed GPU interconnect. We deploy the base model and one ControlNet on two AWS `g5.2xlarge` instances, each with one A10G GPU and a 10 Gbps network [2]. Fig. 10-Left illustrates the serving latency, where the asynchronous scheme achieves *close-to-ideal* performance by pipelining communication and computation, outperforming the synchronous scheme by 1.25×. Besides, images generated by the two schemes are visually indistinguishable regarding composition and quality (see Fig. 17). The synchronous and asynchronous scheme achieve CLIP scores of 33.5 and 33.7, respectively. Compared to the sync scheme, the async has an FID of 2.7 and an SSIM of 0.77.

9.4 Optimizations for LoRA Serving

We now evaluate KATZ's design for efficient LoRA serving, excluding other optimizations. We only include DIFFUSERS because other baselines have no specialized design for LoRAs. As described in §6, DIFFUSERS requires two steps to patch on a LoRA: loading the adapter from storage and then merging its weights to the base model via a create-and-replace operation [37]. This approach is inefficient. As shown in Fig. 16-Right, with a single LoRA (341 MiB), it increases the latency by 80%; with two LoRAs, one 341 MiB and the other 456 MiB, it results in 2.1× slowdown. KATZ addresses this performance issue with BAL and efficient LoRA patching (§6), collectively reducing the LoRA loading and patching overhead to 230 ms, negligible in practice.

Figure 19: Sample images generated by SD3 (left) and Hunyuan-DiT (right) using LoRAs [4, 10].

9.5 Optimizations for Base Model Inference

We also evaluate the proposed optimizations for accelerating base diffusion model inference without adapters. Fig. 18-Left compares KATZ with DIFFUSERS and shows the speedup ratios achieved by latent parallelism on different GPUs, excluding kernel-level optimizations. The gains of latent parallelism are more pronounced on less capable GPUs, achieving speedup ratios of 1.36×, 1.58×, and 1.71× on H800, A100, and A10 GPU, respectively. Although the design of latent parallelism results in imperfect scaling—achieving less than a 2× speedup with 2× GPUs—when combined with other optimizations, KATZ can deliver up to a 1.7× throughput improvement for LoRA-dependent requests (Fig. 18-Right). On the basis of latent parallelism, enabling kernel-level optimizations (§7) additionally yields 1.24× speedup. Collectively, these two optimizations enable KATZ to outperform all baselines regarding base model inference (0C/0L in Fig. 14).

9.6 Serving Throughput

KATZ is primarily designed to minimize serving latency, but it also achieves superior serving throughput for workflows with many adapters. Fig. 18-Right illustrates the request serving throughput of each baseline, measured as the number of images generated per minute of GPU time. When the workflow involves many adapters, KATZ achieves higher throughput (up to 1.7×) compared to other baselines, benefiting from its efficient design of LoRA loading and patching (§6). However, when there is no adapter (0C/0L), KATZ and DISTRIFUSION fall short in throughput as parallelizing base model inference on two GPUs does not saturate their compute capabilities; NIRVANA-20 sometimes achieves good throughput due to its aggressive design, albeit at the cost of image quality loss.

9.7 Generalization to DiT-based Models

KATZ's three designs naturally extend to the DiT backbone, which we evaluate with SD3 [19] and Hunyuan-DiT [32]. First, ControlNet-as-a-Service leads to close-to-ideal speedup. Compared to DIFFUSERS, KATZ achieves speedup ratios of 1.23× and 1.46× for SD3 and Hunyuan-DiT, each with one ControlNet, closely matching the theoretical upper bounds of 1.27× and 1.50× given by the Gustafson's law [24]. Second, BAL effectively overlaps adapter loading with base DiT inference, eliminating the LoRA overhead. User study confirms that images generated by KATZ and DIFFUSERS are visually indistinguishable regarding quality with sample images presented in Fig. 19. Finally, latent parallelism accelerates CFG computation of DiTs, as shown in Fig. 18-Left, achieving even more significant speedup than that of SDXL (UNet).

10 Discussion and Related Works

Generalization to other adapters. As discussed in §2, KATZ can support adapters beyond ControlNet and LoRA. For adapters [29, 62] that share architectural similarities with the base diffusion model, ControlNet-as-a-Service (§5) decouples them from the base and deploys them on dedicated GPUs, enabling caching, parallelization, and sharing. For adapters [68, 70] that incorporate parameter-efficient patches, BAL (§6) can effectively mitigate the overhead associated with model loading and patching. Latent parallelism (§7) applies to the base diffusion model and is adapter-agnostic.

T2I diffusion model inference. In §2, we have presented existing serving systems for T2I workloads [12, 30, 50]. Besides, several works expedite image generation by reducing redundant computations in the denoising process [35, 36, 57]. Yet, these works only optimize the base model inference and overlook the significant latency overhead introduced by adapters. Our work is the first to analyze this problem and addresses the system inefficiencies caused by the use of adapters in T2I serving. It is hence orthogonal to the existing optimization solutions for base diffusion models.

Serving systems with adapters. In the domain of LLMs, pioneering research [16, 31, 43, 58] has designed efficient systems to serve base LLMs with adapters. Despite their effectiveness, these LLM-focused systems are inadequate for diffusion models due to their fundamental differences. First, these systems focus solely on LoRAs, without considering ControlNets which are specialized for diffusion models (§3.1). Second, the LoRAs associated with diffusion models are significantly larger in size and quantity than those for LLMs [16], necessitating external LoRA storage and incurring orders of magnitude higher loading overhead (§3.2). Third, these systems emphasize multiplexing a single base LLM to serve multiple LoRAs simultaneously. Yet, batching yields *minimal* benefits in diffusion model inference (§3.3). Consequently, their optimizations become ineffective in our scenario.

Other model serving systems. Existing research on model serving systems focuses on optimizing latency [17, 53], throughput [14, 61], performance predictability [22, 67], and resources efficiency [23, 51, 60, 65]. These studies apply to various workloads, including graph neural networks [54] and large language models [13, 18, 53, 63]. KATZ is orthogonal to the aforementioned efforts, as T2I models have drastically different computation intensity and workflow.

In the context of online model serving, there exists a series of works to address various challenges such as model placement [33, 67], request scheduling [22, 67], and dynamic scaling [20, 59, 66], which are designed to manage load spikes in model serving systems. While KATZ focuses on per-request latency optimization, these scheduling and scaling approaches are complementary to KATZ and can be seamlessly integrated to effectively handle diverse request arrival patterns.

11 Conclusion

We presented KATZ, the first system that efficiently serves a T2I workflow with many adapters, such as ControlNets and LoRAs. KATZ introduces three novel designs: (1) ControlNet-as-a-Service that deploys ControlNets as a separate service on dedicated GPUs to enable caching, parallelization, and sharing, (2) bounded asynchronous LoRA loading and efficient patching, and (3) latent parallelism that accelerates CFG computation on multiple GPUs. Collectively, these designs decouple adapters from the critical path of a T2I workflow, while accelerating base model inference. Compared to existing systems, KATZ achieves up to 7.8× speedup and 1.7× improvement in throughput, while maintaining image quality.

Acknowledgment

We thank our shepherd Yungang Bao and the anonymous reviewers for their valuable comments that help improve the quality of this work. We also thank colleagues from Alibaba Huiwa AI Team for their assistance. This work was supported in part by the Alibaba Innovative Research (AIR) Grant, RGC CRF Grant (Ref. #C6015-23G), RGC GRF Grants (Ref. #16217124 and #16210822), and NSFC/RGC CRS Grants (Ref. #CRS_PolyU501/23 and #CRS_HKUST601/24).

References

[1] Accelerate inference of text-to-image diffusion models. https://huggingface.co/docs/diffusers/en/tutorials/fast_diffusion, 2025.

[2] Amazon EC2 G5 Instances. https://aws.amazon.com/ec2/instance-types/g5/, 2025.

[3] HuggingFace Diffusers SDXL ControlNets. https://huggingface.co/diffusers/controlnet-depth-sdxl-1.0, 2025.

[4] HunyuanDiT LoRA. https://huggingface.co/Tencent-Hunyuan/HYDiT-LoRA, 2025.

[5] Midjourney AI. https://www.midjourney.com/explore, 2025.

[6] NVIDIA InfiniBand Switch Systems User Manual. https://docs.nvidia.com/networking/display/qm97x0pub/interfaces, 2025.

[7] NVIDIA NVLink: High-speed GPU interconnect. https://www.nvidia.com/en-us/design-visualization/nvlink-bridges/, 2025.

[8] OpenAI CLIP. https://huggingface.co/openai/clip-vit-base-patch16, 2025.

[9] OpenAI DALL·E 2. https://openai.com/index/dall-e-2/, 2025.

[10] SD3 DreamBooth LoRA. https://huggingface.co/BeQuiet94/trained-sd3-lora, 2025.

[11] Adobe. Create with Adobe Firefly generative AI. https://www.adobe.com/products/firefly.html, 2025.

[12] Shubham Agarwal, Subrata Mitra, Sarthak Chakraborty, Srikrishna Karanam, Koyel Mukherjee, and Shiv Kumar Saini. Approximate caching for efficiently serving text-to-image diffusion models. In *Proc. USENIX NSDI*, 2024.

[13] Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav Gulavani, Alexey Tumanov, and Ramachandran Ramjee. Taming throughput-latency tradeoff in LLM inference with Sarathi-Serve. In *Proc. USENIX OSDI*, 2024.

[14] Sohaib Ahmad, Hui Guan, Brian D. Friedman, Thomas Williams, Ramesh K. Sitaraman, and Thomas Woo. Proteus: A high-throughput inference-serving system with accuracy scaling. In *Proc. ACM ASPLOS*, 2024.

[15] Fengxiang Bie, Yibo Yang, Zhongzhu Zhou, Adam Ghanem, Minjia Zhang, Zhewei Yao, Xiaoxia Wu, Connor Holmes, Pareesa Golnari, David A. Clifton, Yuxiong He, Dacheng Tao, and Shuaiwen Leon Song. RenAIssance: A survey into AI text-to-image generation in the era of large model. *IEEE Trans. Pattern Anal. Mach. Intell.*, 2024.

[16] Lequn Chen, Zihao Ye, Yongji Wu, Danyang Zhuo, Luis Ceze, and Arvind Krishnamurthy. Punica: Multi-tenant LoRA serving. In *Proc. MLSys*, 2024.

[17] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J. Franklin, Joseph E. Gonzalez, and Ion Stoica. Clipper: A low-latency online prediction serving system. In *Proc. USENIX NSDI*, 2017.

[18] Jiangfei Duan, Runyu Lu, Haojie Duanmu, Xiuhong Li, Xingcheng Zhang, Dahua Lin, Ion Stoica, and Hao Zhang. MuxServe: Flexible spatial-temporal multiplexing for multiple LLM serving. In *Proc. ICML*, 2024.

[19] Patrick Esser, Sumith Kulal, Andreas Blattmann, et al. Scaling rectified flow transformers for high-resolution image synthesis. In *Proc. ICML*, 2024.

[20] Yao Fu, Leyang Xue, Yeqi Huang, Andrei-Octavian Brabete, Dmitrii Ustiugov, Yuvraj Patel, and Luo Mai. ServerlessLLM: Low-Latency serverless inference for large language models. In *Proc. OSDI*, 2024.

[21] Alan Gray. Getting Started with CUDA Graphs. https://developer.nvidia.com/blog/cuda-graphs/, 2019.

[22] Arpan Gujarati, Reza Karimi, Safya Alzayat, Wei Hao, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace. Serving DNNs like Clockwork: Performance predictability from the bottom up. In *Proc. USENIX OSDI*, 2020.

[23] Jashwant Raj Gunasekaran, Cyan Subhra Mishra, Prashanth Thinakaran, Bikash Sharma, Mahmut Taylan Kandemir, and Chita R. Das. Cocktail: A multidimensional optimization for model serving in cloud. In *Proc. USENIX NSDI*, 2022.

[24] John L. Gustafson. Reevaluating Amdahl's law. *Commun. ACM*, 1988.

[25] Jack Hessel, Ari Holtzman, Maxwell Forbes, Ronan Le Bras, and Yejin Choi. CLIPScore: A reference-free evaluation metric for image captioning. In Marie-Francine Moens, Xuanjing Huang, Lucia Specia, and Scott Wen-tau Yih, editors, *Proc. EMNLP*, 2021.

[26] Martin Heusel, Hubert Ramsauer, Thomas Unterthiner, Bernhard Nessler, and Sepp Hochreiter. GANs trained by a two time-scale update rule converge to a local Nash equilibrium. In *Proc. NIPS*, 2017.

[27] Jonathan Ho and Tim Salimans. Classifier-free diffusion guidance. In *Proc. NeurIPS 2021 Workshop on Deep Generative Models and Downstream Applications*, 2021.

[28] Edward J Hu, yelong shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. LoRA: Low-rank adaptation of large language models. In *Proc. ICLR*, 2022.

[29] Xuan Ju, Xian Liu, Xintao Wang, Yuxuan Bian, Ying Shan, and Qiang Xu. BrushNet: A plug-and-play image inpainting model with decomposed dual-branch diffusion. In *Proc. ECCV*, 2024.

[30] Muyang Li, Tianle Cai, Jiaxin Cao, Qinsheng Zhang, Han Cai, Junjie Bai, Yangqing Jia, Ming-Yu Liu, Kai Li, and Song Han. DistriFusion: Distributed parallel inference for high-resolution diffusion models. In *Proc. IEEE/CVF CVPR*, 2024.

[31] Suyi Li, Hanfeng Lu, Tianyuan Wu, Minchen Yu, Qizhen Weng, Xusheng Chen, Yizhou Shan, Binhang Yuan, and Wei Wang. Toppings: CPU-assisted, rank-aware adapter serving for LLM inference. In *Proc. USENIX ATC*, 2025.

[32] Zhimin Li, Jianwei Zhang, Qin Lin, et al. Hunyuan-DiT: A powerful multi-resolution diffusion transformer with fine-grained chinese understanding. *arXiv preprint arXiv:2405.08748*, 2024.

[33] Zhuohan Li, Lianmin Zheng, Yinmin Zhong, Vincent Liu, Ying Sheng, Xin Jin, Yanping Huang, Zhifeng Chen, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. AlpaServe: Statistical multiplexing with model parallelism for deep learning serving. In *Proc. OSDI*, 2023.

[34] Haozhe Liu, Wentian Zhang, Jinheng Xie, Francesco Faccio, Mengmeng Xu, Tao Xiang, Mike Zheng Shou, Juan-Manuel Perez-Rua, and Jürgen Schmidhuber. Faster diffusion via temporal attention decomposition. *arXiv preprint arXiv:2404.02747*, 2024.

[35] Xinyin Ma, Gongfan Fang, Michael Bi Mi, and Xinchao Wang. Learning-to-cache: Accelerating diffusion transformer via layer caching. In *Proc. NeurIPS*, 2024.

[36] Xinyin Ma, Gongfan Fang, and Xinchao Wang. DeepCache: Accelerating diffusion models for free. In *Proc. IEEE/CVF CVPR*, 2024.

[37] Sourab Mangrulkar, Sylvain Gugger, Lysandre Debut, Younes Belkada, Sayak Paul, and Benjamin Bossan. PEFT: State-of-the-art parameter-efficient fine-tuning methods. https://github.com/huggingface/peft, 2022.

[38] William Peebles and Saining Xie. Scalable diffusion models with transformers. In *Proc. IEEE/CVF ICCV*, 2023.

[39] Dustin Podell, Zion English, Kyle Lacey, Andreas Blattmann, Tim Dockhorn, Jonas Müller, Joe Penna, and Robin Rombach. SDXL: Improving latent diffusion models for high-resolution image synthesis. In *Proc. ICLR*, 2024.

[40] Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, Gretchen Krueger, and Ilya Sutskever. Learning transferable visual models from natural language supervision. In *Proc. ICML*, 2021.

[41] Robin Rombach, Andreas Blattmann, Dominik Lorenz, Patrick Esser, and Björn Ommer. High-resolution image synthesis with latent diffusion models. In *Proc. IEEE/CVF CVPR*, 2022.

[42] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-Net: Convolutional networks for biomedical image segmentation. In *Proc. MICCAI*, 2015.

[43] Ying Sheng, Shiyi Cao, Dacheng Li, Coleman Hooper, Nicholas Lee, Shuo Yang, Christopher Chou, Banghua Zhu, Lianmin Zheng, Kurt Keutzer, Joseph E. Gonzalez, and Ion Stoica. S-LoRA: Serving thousands of concurrent LoRA adapters. In *Proc. MLSys*, 2023.

[44] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-LM: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

[45] Chenyang Si, Ziqi Huang, Yuming Jiang, and Ziwei Liu. FreeU: Free lunch in diffusion U-Net. In *Proc. IEEE/CVF CVPR*, 2024.

[46] Kolors Team. Kolors: Effective training of diffusion model for photorealistic text-to-image synthesis. https://github.com/Kwai-Kolors/Kolors/commits/master/imgs/Kolors_paper.pdf, 2024.

[47] TheLastBen. Filmic Style, SDXL LoRA. https://huggingface.co/TheLastBen/Filmic, 2025.

[48] TheLastBen. Papercut Style, SDXL LoRA. https://huggingface.co/TheLastBen/Papercut_SDXL, 2025.

[49] TheLastBen. William Eggleston Photography Style, SDXL LoRA. https://huggingface.co/TheLastBen/William_Eggleston_Style_SDXL, 2025.

[50] Patrick von Platen, Suraj Patil, Anton Lozhkov, Pedro Cuenca, Nathan Lambert, Kashif Rasul, Mishig Davaadorj, Dhruv Nair, Sayak Paul, William Berman, Yiyi Xu, Steven Liu, and Thomas Wolf. Diffusers: State-of-the-art diffusion models. https://github.com/huggingface/diffusers, 2022.

[51] Luping Wang, Lingyun Yang, Yinghao Yu, Wei Wang, Bo Li, Xianchao Sun, Jian He, and Liping Zhang. Morphling: Fast, near-optimal auto-configuration for cloud-native model serving. In *Proc. ACM SoCC*, 2021.

[52] Qixun Wang, Xu Bai, Haofan Wang, Zekui Qin, Anthony Chen, Huaxia Li, Xu Tang, and Yao Hu. InstantID: Zero-shot identity-preserving generation in seconds. *arXiv preprint arXiv:2401.07519*, 2024.

[53] Yiding Wang, Kai Chen, Haisheng Tan, and Kun Guo. Tabi: An efficient multi-level inference system for large language models. In *Proc. ACM EuroSys*, 2023.

[54] Yuke Wang, Boyuan Feng, Zheng Wang, Tong Geng, Kevin Barker, Ang Li, and Yufei Ding. MGG: Accelerating graph neural networks with fine-grained intra-kernel communication-computation pipelining on multi-GPU platforms. In *Proc. USENIX OSDI*, 2023.

[55] Zhou Wang, Alan C Bovik, Hamid R Sheikh, and Eero P Simoncelli. Image quality assessment: From error visibility to structural similarity. *IEEE Trans. Image Process.*, 2004.

[56] Zijie J. Wang, Evan Montoya, David Munechika, Haoyang Yang, Benjamin Hoover, and Duen Horng Chau. DiffusionDB: A large-scale prompt gallery dataset for text-to-image generative models. In *Proc. ACL*, 2023.

[57] Felix Wimbauer, Bichen Wu, Edgar Schoenfeld, Xiaoliang Dai, Ji Hou, Zijian He, Artsiom Sanakoyeu, Peizhao Zhang, Sam Tsai, Jonas Kohler, et al. Cache me if you can: Accelerating diffusion models through block caching. In *Proc. IEEE/CVF CVPR*, 2024.

[58] Bingyang Wu, Ruidong Zhu, Zili Zhang, Peng Sun, Xuanzhe Liu, and Xin Jin. dLoRA: Dynamically orchestrating requests and adapters for LoRA LLM serving. In *Proc. USENIX OSDI*, 2024.

[59] Hao Wu, Yue Yu, Junxiao Deng, Shadi Ibrahim, Song Wu, Hao Fan, Ziyue Cheng, and Hai Jin. StreamBox: A lightweight GPU SandBox for serverless inference workflow. In *Proc. ATC*, 2024.

[60] Lingyun Yang, Yongchen Wang, Yinghao Yu, Qizhen Weng, Jianbo Dong, Kan Liu, Chi Zhang, Yanyi Zi, Hao Li, Zechao Zhang, Nan Wang, Yu Dong, Menglei Zheng, Lanlan Xi, Xiaowei Lu, Liang Ye, Guodong Yang, Binzhang Fu, Tao Lan, Liping Zhang, Lin Qu, and Wei Wang. GPU-disaggregated serving for deep learning recommendation models at scale. In *Proc. USENIX NSDI*, 2025.

[61] Yanan Yang, Laiping Zhao, Yiming Li, Huanyu Zhang, Jie Li, Mingyang Zhao, Xingzhen Chen, and Keqiu Li. INFless: A native serverless system for low-latency, high-throughput inference. In *Proc. ACM ASPLOS*, 2022.

[62] Hu Ye, Jun Zhang, Sibo Liu, Xiao Han, and Wei Yang. IP-Adapter: Text compatible image prompt adapter for text-to-image diffusion models. *arXiv preprint arXiv:2308.06721*, 2023.

[63] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. Orca: A distributed serving system for transformer-based generative models. In *Proc. USENIX OSDI*, 2022.

[64] Jiahui Yu, Yuanzhong Xu, Jing Yu Koh, et al. Scaling autoregressive models for content-rich text-to-image generation. *Transactions on Machine Learning Research*, 2022.

[65] Chengliang Zhang, Minchen Yu, Wei Wang, and Feng Yan. MArk: Exploiting cloud services for cost-effective, SLO-aware machine learning inference serving. In *Proc. USENIX ATC*, 2019.

[66] Dingyan Zhang, Haotian Wang, Yang Liu, Xingda Wei, Yizhou Shan, Rong Chen, and Haibo Chen. Fast and live model auto scaling without caching. In *Proc. OSDI*, 2025.

[67] Hong Zhang, Yupeng Tang, Anurag Khandelwal, and Ion Stoica. Shepherd: Serving DNNs in the wild. In *Proc. USENIX NSDI*, 2023.

[68] Lvmin Zhang. Fooocus. https://github.com/lllyasviel/Fooocus, 2025.

[69] Lvmin Zhang, Anyi Rao, and Maneesh Agrawala. Adding conditional control to text-to-image diffusion models. In *Proc. IEEE/CVF ICCV*, 2023.

[70] Lvmin Zhang, Anyi Rao, and Maneesh Agrawala. IC-Light. https://github.com/lllyasviel/IC-Light, 2025.

[71] Richard Zhang, Phillip Isola, Alexei A Efros, Eli Shechtman, and Oliver Wang. The unreasonable effectiveness of deep features as a perceptual metric. In *Proc. IEEE/CVF CVPR*, 2018.

[72] Yuxin Zhang, Weiming Dong, Fan Tang, et al. Prospect: Prompt spectrum for attribute-aware personalization of diffusion models. *ACM Trans. Graph.*, 2023.

[73] Haozhe Zhao, Xiaojian Ma, Liang Chen, Shuzheng Si, Rujie Wu, Kaikai An, Peiyu Yu, Minjia Zhang, Qing Li, and Baobao Chang. UltraEdit: Instruction-based fine-grained image editing at scale. In *Proc. NeurIPS Datasets and Benchmarks Track*, 2024.

[74] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, et al. Judging LLM-as-a-judge with MT-Bench and Chatbot Arena. In *Proc. NeurIPS Datasets and Benchmarks Track*, 2023.

PopFetcher: Towards Accelerated Mixture-of-Experts Training Via Popularity Based Expert-Wise Prefetch

Junyi Zhang[1], Chuanhu Ma[1], Xiong Wang[1]*, Yuntao Nie[1], Yuqing Li[2], Yuedong Xu[3], Xiaofei Liao[1], Bo Li[4], and Hai Jin[1]

[1]*National Engineering Research Center for Big Data Technology and System, Service Computing Technology and System Lab/Cluster and Grid Computing Lab, School of Computer Science and Technology, Huazhong University of Science and Technology*

[2]*School of Cyber Science and Engineering, Wuhan University*

[3]*School of Information Science and Engineering, Fudan University*

[4]*Department of Computer Science and Engineering, Hong Kong University of Science and Technology*

Abstract

Scaling laws indicate that increasing model size enhances performance. The *Mixture-of-Experts (MoE)* architecture enables scaling model parameters to trillions while requiring only a sub-linear increase in training computations. However, the sparse activation of experts within MoE leads to substantial *All-to-All communications* and *imbalanced* computation workloads, which in turn can severely degrade training efficiency. In this paper, we develop PopFetcher, a scalable MoE training system with *popularity*-aided *expert-wise prefetching*, to address these communication and computation bottlenecks. Specifically, PopFetcher uncovers *skewed* and *correlated* patterns in expert selection, and implements a lightweight sliding-window technique to accurately predict the popularity of experts. As a result, PopFetcher facilitates dynamic identification of high-demand experts and *prefetches* them in the next layer during the execution of current *non-MoE computations*, thereby exploiting the *idle* network links to reduce dispatched tokens in upcoming All-to-All communications. PopFetcher rigorously formulates the end-to-end training latency and develops a tailored *pruning* strategy to derive the globally optimal prefetching scheme, which can restore both communication and computation balances based on the underlying network infrastructure. By *prioritizing* All-to-All data stream during the backward pass, PopFetcher significantly alleviates the communication blockage. Extensive experiments conducted on GPU clusters demonstrate that PopFetcher outperforms existing state-of-the-art systems, reducing training time by *15%-94.5%*.

1 Introduction

In recent years, Transformer-based *pre-trained language models (PLMs)*, such as BERT [8], GPT [5,29], and Llama [36,37], have achieved enormous success across various domains, including natural language processing [12], computer vision [9, 25], and machine translation [3]. The scaling law for PLMs demonstrates an empirical relationship between model size and performance [20], proliferating gigantic PLMs with up to trillions of parameters to enhance service quality. However, large-scale PLMs come at the cost of *sharply increasing* computational demands, often hitting the trainability wall [12]. In response to these challenges, the *Mixture-of-Experts (MoE)* architecture has garnered more and more interest [31]. MoE model, like DeepSeek [7], Qwen [35] and Phi [2], expands numerous sub-models or experts within Transformer framework, which are *sparsely activated* by a gate network, thus allowing for scaling model capacity while ensuring only a sub-linear increase in training computations. Substantial evidence also corroborates the potential of MoE in delivering performance on par with its dense counterparts [4].

Despite the benefits of MoE, its sparsely activated structure also presents unique challenges in model training [30]. As the number of experts grows, the memory requirement for MoE model also scales, often demanding hundreds of GB or even TB, far beyond the capacity of a single GPU (typically tens of GB). As such, *expert parallelism (EP)* has been proposed to train large-scale MoE models by placing various experts to different GPU workers for parallel processing [21]. Within each MoE layer, input tokens are dynamically routed to certain experts, who may reside on remote workers, incurring frequent cross-machine data transfers. Specifically, this process necessitates two All-to-All communication primitives on the critical path: first All-to-All operation dispatches input tokens to selected experts and the second collects the computation results back, as depicted in Figure 1. Since All-to-All communication is *time-consuming*, the MoE training, especially for large models, is often severely prolonged [40].

Considerable efforts have been devoted to addressing the communication bottleneck and enhancing MoE training efficiency [17, 22]. Common strategies include designing a balanced gate network to alleviate uneven expert workloads [44] or pipelining the All-to-All operation through token subdivision of communication tasks [32]. While these approaches can improve training performance, the volume of communication for token dispatching and collection *remains substantial*.

*Corresponding author: Xiong Wang (xiongwang@hust.edu.cn)

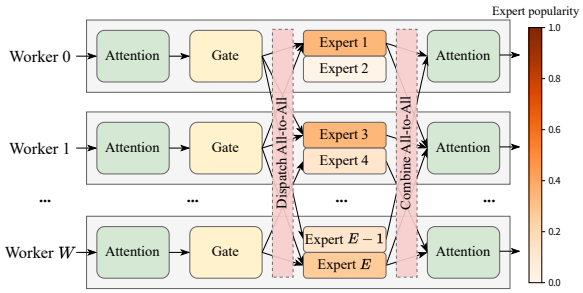


Figure 1: All-to-All communications during MoE training.

To minimize transferred tokens, FasterMoE [16] introduces a strategy to "shadow" partial experts to all GPU workers, which allows specific tokens to be processed locally, alleviating the need for external transmission. However, the periodic *broadcasting* of expert parameters can undermine the benefits of decreased token transmission due to potential communication cost and interference. Alternatively, Janus [24] attempts to optimize whether to fetch experts for tokens or send tokens to experts before the training starts, conditioned on their relative overheads. Yet, it struggles when the size of expert parameters or token data consistently exceeds the other. Currently, expert scheduling is mostly conducted concurrently with All-to-All communication, which may mitigate token transfers but also introduces *additional* communication overheads for expert scheduling. This raises a critical question: *Can we stagger expert scheduling with token dispatching to eliminate the interruption of expert scheduling and more effectively address the All-to-All communication bottleneck*? To answer the question, we face the following challenges.

First, to minimize the interference of expert scheduling on token dispatching, it is essential to retrieve the necessary experts before the training progresses to the current MoE layer. This requires *advanced* knowledge of expert statistics to facilitate timely expert retrieval. However, acquiring such information is challenging due to the dynamic training nature. Second, experts within the same layer often receive tokens that *vary greatly* in volume, which leads to considerable differences in performance gains when fetching different experts. Optimally deciding which experts to fetch from which worker involves modeling the training latency, yet the net effect of expert scheduling and token transmission is difficult to quantify. Third, All-Reduce operations among gradient aggregation can *compete for* limited network bandwidth with All-to-All communications, consequently delaying the backward computations. To enhance training efficiency, it is essential to schedule different network streams independently.

In this paper, we propose PopFetcher, an efficient MoE training system featured with *popularity* based *expert-wise prefetching*, to address the communication bottleneck and ensure balanced workloads. Specifically, PopFetcher employs a lightweight prediction to acquire the popularity of experts in each MoE layer, which effectively assists the expert prefetching design and remains uninterrupted to normal training. Building on this statistics, PopFetcher enables each worker to *preemptively* pull popular experts from neighboring workers in idle periods or to push local tokens to remote workers when fetching corresponding experts is costlier. This *hybrid pull-push* mechanism unleashes the potentials of processing tokens both locally and remotely. PopFetcher rigorously formulates the end-to-end training latency, incorporating both computation and communication costs, to optimize the prefetching of experts for each worker. As a result, PopFetcher can substantially reduce the volume of token dispatching in EP, while incurring *negligible interference* with All-to-All communication since experts are retrieved in advance. Moreover, PopFetcher *prioritizes* the All-to-All stream during the backward pass, which further alleviates the communication blockage. In summary, this paper makes the following contributions.

- We discover the *correlation* between expert selections across consecutive MoE layers, and further develop a *lightweight* sliding-window based prediction scheme to precisely access the popularity of experts. With these insights, we design PopFetcher, an efficient MoE training system with popularity-aided expert-wise prefetching, which enables a *preliminary* expert scheduling to address the All-to-All communication bottleneck.
- PopFetcher devises a hybrid paradigm of *expert pull and token push* for EP-based MoE training. Through a thorough analysis of end-to-end training latency, PopFetcher enables the identification of the optimal experts to be prefetched for each worker while also respecting their memory constraints. By coupling with the underlying network architecture, PopFetcher fully utilizes high-speed links for expert retrieval, which greatly mitigates prefetching time.
- PopFetcher *prioritizes* All-to-All communication stream over All-Reduce operation among prefetched experts. By effectively pipelining non-MoE All-Reduce tasks with MoE backward computations, PopFetcher significantly mitigates network contention and reduces communication blockages.
- Extensive experiments on real-world datasets show the superiority of PopFetcher in training throughput and latency. In particular, PopFetcher reduces training time by *15%-94.5%* than state-of-the-art systems.

2 Background and Motivation

We first introduce the preliminary of distributed training of MoE models, then present our motivation, as well as the opportunities and challenges of PopFetcher.

2.1 Background

2.1.1 Transformer-Based PLM

Transformer model [38] has been recognized as the fundamental architecture in sequence processing. A typical Transformer block includes an Attention layer and a *multi-layer percep-*

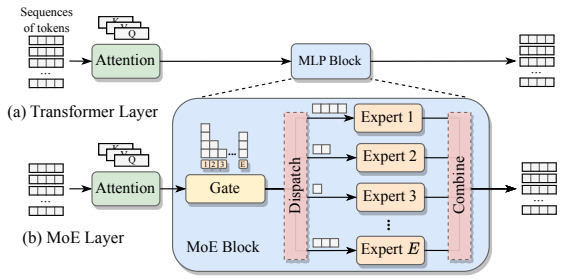


Figure 2: Structure of MoE block.

tron (MLP), generally comprising two *fully-connected (FC)* layers with a ReLU activation in between. Attention network transforms input tokens into query, key and value matrices, which then undergo processing via scaled dot-product attention, with outcomes being further channeled into the MLP. Output from MLP will be added to the block's input and normalized through a layer normalization. This standard configuration allows for stacking multiple identical Transformers to build powerful PLMs, like BERT [8] and GPT [5, 29].

2.1.2 MoE Architecture and Model Training

Despite the success of Transformer architecture, scaling PLM capacity remains challenging due to increasing computation demands for training. MoE model [31], illustrated in Figure 2, tackles this by dividing the dense MLP into multiple smaller, sparse experts, thereby exploiting the inherent sparsity of Transformer to scale model size with only a sub-linear increase in computation cost. During training, each input token is routed by a gate to one or more experts, typically employing top-k expert selection where k is often 1 or 2. Token dispatching to experts and the subsequent return of results are managed via All-to-All communication primitives.

Large-scale MoE models often far exceed the capacity of a single GPU. To facilitate the training of MoE, EP is adopted to distribute various experts from each MoE layer across different GPU workers [21]. Typically, the model layers that reside between MoE layers (e.g., Attention layer), referred to as non-MoE layers, are replicated on each worker, following a principle similar to that of data parallelism [19].

2.2 Motivation

Skewed expert activation leads to hot expert. In MoE architecture, the gate module's variability in routing input tokens often leads to an imbalanced workload among experts. As in Figure 3, which shows the expert selection distribution within a layer of MoE-GPT model [41], the pattern of token dispatching evolves considerably early in training but soon exhibits a trend towards *localized activity*, with minor variations between adjacent iterations. Despite attempts at balanced gating strategies, such as imposing capacity limits on each expert [21], certain experts consistently receive more tokens, placing heavier computations on GPUs hosting these *hot experts*. Such load imbalance is particularly pronounced in the frontier layers of the MoE architecture, whereas in deep layers, the load tends to stabilize and changes less dramatically. Emergence of hot experts requires an effective management of load distribution to avoid overburdening GPU workers.

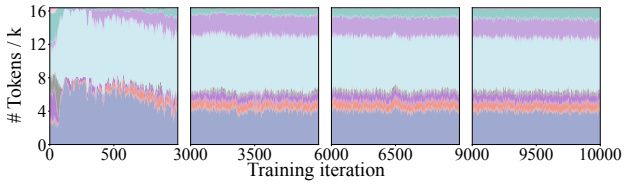


Figure 3: Skewed expert selection within a MoE layer (k on the y-axis represents units of 1000).

Table 1: Time consumption within a MoE layer.

# Layers	Batch size	# Experts	A2A Ratio
12	16	16	56.44 %
24	16	16	57.57 %
24	32	16	57.20 %
12	16	32	56.13 %
12	16	64	56.34 %

All-to-All communication bottlenecks in MoE training. During the training of MoE models, each token undergoes All-to-All communication twice per MoE layer, including token dispatching and subsequent output collection, which often leads to significant delays. Table 1 indicates that All-to-All communication can account for 50%-60% of the total time when training MoE-GPT across eight workers. The primary communication bottleneck arises from the *mandatory data synchronizations* in each MoE layer, where the commencement of expert execution and subsequent computations is delayed until all requisite tokens and their processed outputs are received. This synchronization is particularly problematic when hot experts are present, as their increased load further intensifies the network contentions. Therefore, a meticulously designed EP is critical to address synchronous execution challenge and reduce communication overhead.

Coarse-grained expert scheduling is insufficient. To enhance MoE training efficiency, replicating partial experts across multiple GPUs is adopted to alleviate the skewed computations on overloaded workers, allowing specific tokens to be processed locally by shadowed experts rather than being dispatched remotely [16]. While effective, broadcasting the expert parameters to all GPUs can introduce inadvertent delays due to data synchronizations. An alternative approach attempts to pull necessary expert parameters to the local GPU, which can decrease the need for pushing tokens to remote experts [24]. However, this method encounters significant limitations when expert parameters are substantially larger than input tokens, making pulling experts even more communication-intensive. Despite these efforts, existing *coarse-grained* MoE schemes still struggle with imbalanced computation loads and persistent All-to-All communication bottlenecks.

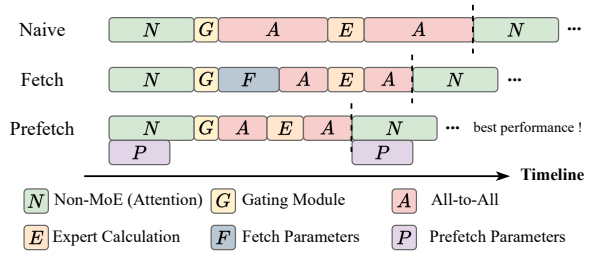


Figure 4: Execution timeline of different expert schedules.

2.3 Opportunities and Challenges

Addressing communication bottleneck while restoring computation balance is crucial for improving the efficiency of MoE model training. We next explore the opportunity to achieve this objective and highlight the potential challenges.

2.3.1 Opportunities

Efforts to promote MoE efficiency have typically pursued two orthogonal directions.

Model design. Literature reveals attempts to refine the MoE model architecture itself. For instance, GShard [21] incorporates a load balance loss to evenly distribute workloads across different GPU workers. Similarly, expert routing strategies [42] have been redefined to allow the gate module to select the top-k best-matching tokens, enhancing load balancing. While these methods aid in achieving more equitable workload distribution, they also risk *compromising model accuracy*, and the communication bottleneck *persists*.

System design. System-oriented approaches like FasterMoE [16] and FlexMoE [28] alleviate the adverse impact of load imbalance by shadowing or replicating popular experts. However, they can introduce *additional* overhead for parameter synchronization or system adjustment, potentially offsetting the benefits of improved workload distribution.

Despite these challenges, non-MoE components between MoE layers open up optimization opportunities. Figure 4 shows the data flow within a MoE layer, implying that during the computation in *non-MoE sections*, like the Attention layer, a worker utilizes only local input data, leaving network links underutilized. As training progresses, expert load variations tend to stabilize, offering a chance to use historical token distribution for expert prefetching. That is, we can proactively pull popular experts in next MoE layer from remote GPUs to a local worker in anticipation of their forthcoming selections, thus diminishing future All-to-All communication demands by using *currently idle* links.

2.3.2 Challenges

Prefetching hot experts during non-MoE computation phases is promising to minimize communication overhead and balance GPU workloads. However, it also presents unique challenges that need to be addressed.

How to accurately predict popular experts in next MoE layer? As training progresses, the load distribution across experts *changes dynamically*. For effective expert scheduling, it is crucial to develop a lightweight method that can accurately predict which experts will be in high demand in subsequent layers, yet challenging for their dynamic nature.

How to prefetch appropriate experts subject to worker capacity? Given the *limited memory* capacity of GPU workers, and the presence of multiple experts already on the local GPU, deciding which and how many experts to prefetch presents a dilemma. This decision must balance the benefits of expert prefetching against available memory resources.

How to manage additional cost associated with expert prefetching? While prefetching can be scheduled to overlap with non-MoE computations, it must adhere to the *data dependencies* of the MoE workflow. Ensuring that prefetched experts are prepared before the computation of the next layer begins can introduce additional synchronization overhead. Furthermore, the All-Reduce operation among prefetched experts may compete with All-to-All communication, potentially obstructing gradient computations in backward pass.

3 PopFetcher Overview

In this paper, we present PopFetcher, an advanced MoE training system with popularity based expert-wise prefetching, to decrease All-to-All communication overhead and ensure balanced GPU workload. PopFetcher maintains an expert pool on each worker to store both local expert parameters and those prefetched from remote workers. Figure 5 exhibits the system architecture of PopFetcher, which is primarily composed of three key modules: routing information collector, prefetching decision-maker, and asynchronous scheduling executor.

3.1 Routing Information Collector

Each worker node in PopFetcher is outfitted with a routing information collector to track and monitor the *runtime popularity* of experts. Specifically, this module records the gate selection details for each token across every MoE layer and updates the distribution of tokens routed to each expert. Based on this data, the collector renews the expert popularity within each MoE layer, as well as relaying to the prefetching decision-maker for more informed expert retrieval. Expert popularity compiled by each worker is promptly synchronized via torch.distributed.all_reduce to maintain consistency across all workers. Since the popularity vector is small, the synchronization overhead is negligible.

3.2 Prefetching Decision-Maker

The prefetching module periodically aggregates data from the routing information collector across all workers to discern the popularity of each expert. Considering that each GPU

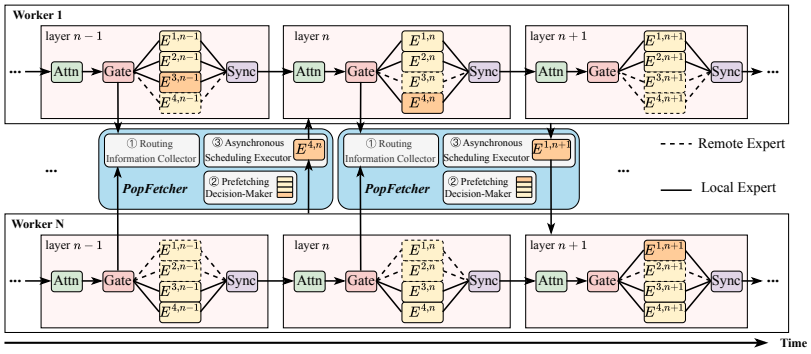


Figure 5: System architecture of PopFetcher.

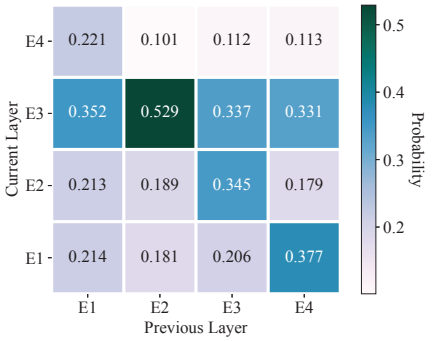


Figure 6: Expert correlation between MoE layers within an iteration (training MoE-GPT with four experts per layer).

worker already hosts local experts and corresponding activations from the forward pass, fetching all remaining experts is impractical due to memory constraints. Informed by the updated popularity statistics, the decision-maker crafts a *tailored* prefetching strategy for each worker within the confines of limited GPU memory. PopFetcher carefully orchestrates the decision-making process to avoid disrupting regular training activities, as routing information can be analyzed on the CPU concurrently with ongoing GPU-based training operations. Furthermore, PopFetcher decides on expert prefetching for the upcoming MoE layer to minimize the end-to-end training latency as the current layer progresses.

3.3 Asynchronous Scheduling Executor

The scheduling executor aims to utilize idle network links during MoE training by preemptively and asynchronously fetching experts from the next layer that are originally located on remote workers. This strategy reduces the volume of transferred tokens during All-to-All communications, thereby effectively addressing the communication bottleneck. Ideally, expert scheduling should coincide with the computations of current Transformer's non-MoE sections, i.e., the Attention layer, which ensures minimal overhead and *avoids interference* with the execution of the current MoE layer.

In general, the executor intelligently schedules experts for each worker by prioritizing hot experts from the global pool based on directives from the prefetching decision-maker. Consequently, PopFetcher not only alleviates the All-to-All communication volume but also ensures a balanced GPU workload. Considering that multiple experts can exist across different workers due to expert prefetching, PopFetcher concurrently performs All-Reduce operations for non-MoE layers and gradient computations for experts during the backward pass to reduce communication overhead. To further minimize the impact on backward All-to-All communication and avoid the computation blockage, we also prioritize the All-to-All data flows over the All-Reduce stream.

4 Lightweight Popularity Prediction

In this section, we will delve into a dynamic prediction for expert popularity, which is deemed to be both lightweight and accurate to enhance expert prefetching. Basically, expert activation is dynamically predetermined by the gate network in standard scenarios, which may limit the opportunities for system-level optimization.

4.1 Expert Correlation between MoE Layers

As mentioned earlier, the selection of experts by input tokens exhibits discernible regularity. Within a MoE layer, certain experts consistently attract more tokens, indicating their popularity, while others are chosen less frequently. Observations from Figure 3 also imply that experts who are popular at one point tend to maintain their popularity temporally.

Beyond individual MoE layers, there is also a correlation in expert selection across consecutive layers. Let us consider an expert index selected by a token in the i-th MoE layer as x, and in the subsequent $(i+1)$-th layer as x'. The tuple (x, x') then represents a set of correlated selections. Figure 6 illustrates the conditional probability of selecting expert x' in the current layer, given the selection of expert x in the previous layer. Outcomes reveal a distinct trend of expert selection between adjacent MoE layers, with *certain correlations* more prevalent than others. In essence, hot experts of the upcoming layer can often be predicted based on expert preferences in the preceding layer.

Although the popularity of experts and their correlations continue to evolve as training progresses, these changes tend to be gradual over time, which supports the effectiveness of early expert prefetching during non-MoE computations.

4.2 Expert Popularity Prediction

Achieving optimal computation and communication balance among workers hinges on accurately predicting expert popularity, a task complicated by its dynamic nature. To address

Table 2: Main notations.

Notation	Definition
N	# workers
K	# experts per layer on each worker
E_w^i	i-th expert on worker w
P_w	Computation throughput of worker w
$W_{n,w}$	Network bandwidth of link between workers n, w
I_w	# input tokens of worker w
B_w	# tokens received by worker w
B_w^i	# tokens sent to E_w^i
p_w^i	Popularity of E_w^i
H	Hidden size (length of embedding vectors)
α	Fraction of intermediate embedding in MLP
$\delta_{n,w}^i$	1 if E_n^i is prefetched by worker w, else 0

this while minimally interfering with normal training, we propose a sliding-window based prediction to pinpoint hot experts [6]. Specifically, the routing information collector routinely gathers runtime selection data from the gate network during the forward pass, which is then analyzed to update our current understanding of expert popularity.

Initially, we analyze the distribution of tokens over recent iterations, up to the *current MoE* layer:

$$p_{\text{seq}} = \frac{\text{\# tokens allocated in the past } s \text{ iterations}}{\text{\# tokens processed in the past } s \text{ iterations}} \quad (1)$$

Here s represents the size of the sliding window, which, based on our testing, is optimally set at 10 iterations. Our goal is to accurately predict expert popularity for the *upcoming MoE* layer, which involves incorporating the expert correlations, as illustrated in Figure 6. Let M signify the total number of tokens, with each token expressed by $T_m, m \in \{1, 2, \cdots, M\}$. Also, we denote the i-th expert on the j-th layer (i.e., Transformer block) as $E^{i,j}$, where each layer contains K' experts in total. Then, the correlation between any experts $E^{i,j}$ and $E^{h,j+1}$ can be evaluated by their conditional probability:

$$\Pr\left(E^{h,j+1} | E^{i,j}\right) = \frac{1}{M} \sum_{m=1}^{M} \Pr\left(E^{h,j+1} | E^{i,j}, T_m\right) \quad (2)$$

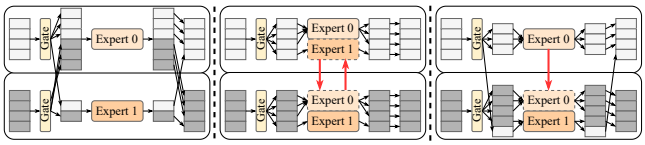
The expert popularity based on the correlation is given by:

$$p\left(E^{h,j+1}\right) = \sum_{i=1}^{K'} \Pr\left(E^{h,j+1} | E^{i,j}\right) p_{\text{seq}}^{i,j} \quad (3)$$

Eq. (3) enables an accurate, lightweight popularity prediction of experts in the subsequent layer by leveraging timely updated selection data from the preceding layer. As a result, our PopFetcher can dynamically adapt to changes in expert demand without overburdening the system.

5 Expert-Wise Prefetching and Scheduling

This section details our strategy for expert prefetching and scheduling given limited GPU memory. To aid our later presentation, we summarize the main notations in Table 2.



(a) Expert-centric (b) Data-centric (c) Hybrid-paradigm

Figure 7: Hybrid push-pull paradigm.

5.1 Hybrid Push-Pull Paradigm

Imbalances in communication and computation due to skewed expert selection substantially prolong MoE training. Previous attempts address this by either sending (pushing) tokens to experts (expert-centric) [12, 16, 21] or pulling experts to local GPUs (data-centric) [24], which, however, proves insufficient in managing the uneven token dispatching inherent in MoE architecture, as illustrated in Figure 7. Since these methods exclusively send tokens or pull experts, they fail to handle blockages during large data transmissions, as all data chunks are treated equally and scheduled simultaneously. To clarify, each expert module typically comprises a FeedForward Network with two Linear layers, featuring matrix dimensions of $H \times \alpha H$ and $\alpha H \times H$, respectively, with α commonly set to 2. Then, the parameter count for a single expert layer is $4H^2$. Using a float32 data type with $H = 1024$, the parameter reaches 16MB. Besides, cross-machine data transfer per token during two All-to-All communications is $2H$. Hence, we deduce that pulling an expert becomes more viable when the token transfer exceeds 2048 tokens. Otherwise, the conventional approach of sending tokens offers lower overhead.

We introduce a new hybrid push-pull strategy for tokens and experts, which combines the strengths of both expert-centric and data-centric frameworks. According to Figure 7, after tokens pass through the gate module, their dispatching to different workers varies significantly. By adopting a hybrid paradigm that *concurrently* pushes tokens and pulls experts, we can optimize data transfer based on the current token distribution among experts. Specifically, when the volume of tokens exceeds that of expert parameters, we opt to fetch experts towards the data; otherwise, we default to send tokens as expert parameters are less burdensome.

Our hybrid approach also includes expert prefetching to enable an advanced retrieval of remote experts. For this purpose, we need to accommodate two MoE computation patterns: local computation and prefetching computation. In local computation, tokens are sent to their specific experts for processing, with results returned to the originating node. Prefetching computation involves experts that are prefetched and available locally, eliminating the need to send out tokens over the network. However, prefetching computation introduces an extra step to ensure the logical integrity of model updates. That is, during backward pass, gradients of replicated experts must be sent back to the worker hosting the primary expert for global reduction. Gradient synchronization, while essential,

inevitably consumes network bandwidth and must be *factored into* prefetching decisions.

5.2 Problem Formulation

Our objective is to devise an optimal scheduling strategy that includes expert prefetching to minimize the training latency of *each MoE layer*. In this context, both communication and computation delays are predominantly determined by the performance of the slowest worker.

5.2.1 Training Latency without Prefetching

First, we consider the conventional case where expert prefetching is excluded, which can guide later analysis of our PopFetcher. Let B_w denote the number of tokens received by worker w, then we have $B_w = \sum_{n=1}^{N} T_{n,w}$, where $T_{n,w} = I_n \sum_{i=1}^{K} p_w^i$ accounts for I_n tokens from worker n based on expert popularity predictions $p_w^i, i \in \{1,2,\cdots,K\}$. For each MoE layer *within a Transformer*, token processing by an expert entails one GeMM operation during forward pass, and two in backward as gradients need to be computed for both the inputs and expert parameters. Moreover, the size of intermediate embedding vector between the FC layers is αH with H being the embedding size of each token. Thus, the total operations for two FC layers in MLP are $4B_w\alpha H^2$. Assuming data type in float32, each GeMM requires $\frac{4B_w\alpha H^2}{P_w}$, leading to a computation latency of $3 \times \frac{4B_w\alpha H^2}{P_w}$. Furthermore, forward and backward pass encompass a total of four rounds of All-to-All communications, with the communication latency given by $4H \sum_{n=1}^{N} \sum_{i=1}^{K} \frac{B_{n,w}^i}{W_{n,w}}$, where $B_{n,w}^i$ denotes the number of tokens sent by worker n to expert E_w^i.

Therefore, the training latency of worker w is attained:

$$Lat_w^{\text{origin}} = 3 \times \frac{4B_w\alpha H^2}{P_w} + 4H \sum_{n=1}^{N} \sum_{i=1}^{K} \frac{B_{n,w}^i}{W_{n,w}} \tag{4}$$

Since the overall latency depends on the slowest worker, the end-to-end training time of a MoE layer becomes

$$Lat^{\text{origin}} = \max_w \left\{ Lat_w^{\text{origin}} \right\}, w \in \{1,2,\cdots,N\} \tag{5}$$

5.2.2 Training Latency with Prefetching

Building on the derivation of Eq. (5), we continue to explore the impact of prefetching hot experts on training latency. Specifically, expert prefetching incurs two types of delays.

- Prefetched expert computation: this delay arises from fetching expert parameters from remote worker to local node for computation, which is influenced by the expert popularity and how many local tokens supposed to be routed to it.
- Model gradient reduction: the time required to reduce gradients of prefetched experts with primary experts.

We derive the training latency when experts are prefetched.

- **Forward pass**. (1) Computation time: the computation time for local tokens is $Comp_{\text{l}}^{\text{f}} = \frac{4B_w\alpha H^2}{P_w}$, while that for prefetched expert is given by $Comp_{\text{p}}^{\text{f}} = \frac{4\alpha H^2}{P_w} \sum_{n=1}^{N} \sum_{i=1}^{K} B_{n,w}^i \delta_{n,w}^i$, where the indicator $\delta_{n,w}^i$ signifies whether expert E_n^i is prefetched by worker w. (2) Communication time: the token transfer time amounts to $Comm_{\text{t}}^{\text{f}} = 2H \sum_{n=1}^{N} \sum_{i=1}^{K} \frac{B_{n,w}^i (1-\delta_{n,w}^i)}{W_{n,w}}$.
- **Backward pass**. (1) Computation time: in the backward pass, local token computation time doubles to $Comp_{\text{l}}^{\text{b}} = 2 \times \frac{4B_w\alpha H^2}{P_w}$, and prefetched expert computation needs $Comp_{\text{p}}^{\text{b}} = 2 \times \frac{4\alpha H^2}{P_w} \sum_{n=1}^{N} \sum_{i=1}^{K} B_{n,w}^i \delta_{n,w}^i$. (2) Communication time: the token transfer $Comm_{\text{t}}^{\text{b}}$ mirrors that of the forward pass $Comm_{\text{t}}^{\text{f}}$, and the time for gradient reduction is $Comm_{\text{r}}^{\text{b}} = 2\alpha H^2 \sum_{n=1}^{N} \sum_{i=1}^{K} \frac{\delta_{n,w}^i}{W_{n,w}}$.

Including the times $Comp_{\text{l}}^{\text{f}}, Comp_{\text{p}}^{\text{f}}, Comm_{\text{t}}^{\text{f}}$ in forward pass and those $Comp_{\text{l}}^{\text{b}}, Comp_{\text{p}}^{\text{b}}, Comm_{\text{t}}^{\text{b}}, Comm_{\text{r}}^{\text{b}}$ in backward pass, we characterize the latency for worker w:

$$\begin{aligned} Lat_w^{\text{prefetch}} = {} & 3 \times \frac{4B_w\alpha H^2}{P_w} + 3 \times \frac{4\alpha H^2}{P_w} \sum_{n=1}^{N} \sum_{i=1}^{K} B_{n,w}^i \delta_{n,w}^i \\ & + 4H \sum_{n=1}^{N} \sum_{i=1}^{K} \frac{B_{n,w}^i (1-\delta_{n,w}^i)}{W_{n,w}} + 2\alpha H^2 \sum_{n=1}^{N} \sum_{i=1}^{K} \frac{\delta_{n,w}^i}{W_{n,w}} \end{aligned} \tag{6}$$

Recall our goal is to minimize the end-to-end latency, i.e.,

$$Lat^{\text{prefetch}} = \min \max_w \left\{ Lat_w^{\text{prefetch}} \right\}, w \in \{1,2,\cdots,N\} \tag{7}$$

Given the extensive number of global experts, such as 256 GPUs on Azure with 128 experts on each worker [30], the overhead of deciding whether to prefetch an expert is substantial, which needs further optimization.

5.3 Expert Prefetching Decision

Our objective now turns to solving the problem outlined in Eq. (7), focusing on optimizing the assignment $\{\delta_{n,w}^i\}$ of prefetched experts to GPU workers in an efficient manner.

5.3.1 Expert Prefetch Pruning

The primary challenge in designing expert prefetching is the vast search space associated with deciding experts for the next layer. To overcome the problem, we narrow down the search space through *expert pruning*, which effectively reduces the number of experts that need to be considered to at most $k \times N$, with k denoting the top-k expert selection of gate module.

- **GPU memory limitation.** Since each GPU has already hosted local expert parameters and intermediate activations, the aggregate size of experts that can be prefetched must not exceed its available memory. Then,

$$2\alpha H^2 \sum_{n=1}^{N} \sum_{i=1}^{K} \delta_{n,w}^i \leq Mem_w^{\text{free}} \tag{8}$$

- **Transfer time constraint.** Prefetching should ideally overlap with non-MoE layer computations, implying that the time taken to fetch expert needs to fulfill

$$2\alpha H^2 \sum_{n=1}^{N}\sum_{i=1}^{K}\frac{\delta_{n,w}^i}{W_{n,w}} \leq Time^{\text{non-MoE}} \tag{9}$$

These constraints establish an upper bound r_w for the number of experts that can be feasibly prefetched:

$$\sum_{n=1}^{N}\sum_{i=1}^{K}\delta_{n,w}^i \leq r_w \tag{10}$$

Moreover, expert prefetching for any worker is supposed to reduce end-to-end training latency. Consistent with Eqs. (4) and (6), it implies the following difference is greater than 0:

$$\begin{aligned} Lat_w^{\text{origin}} - Lat_w^{\text{prefetch}} &= 4H\sum_{n=1}^{N}\sum_{i=1}^{K}\frac{B_{n,w}^i}{W_{n,w}} \\ -2\alpha H^2\sum_{n=1}^{N}\sum_{i=1}^{K}\frac{\delta_{n,w}^i}{W_{n,w}} &- \frac{12\alpha H^2}{P_w}\sum_{n=1}^{N}\sum_{i=1}^{K}B_{n,w}^i\delta_{n,w}^i \end{aligned} \tag{11}$$

Specifically, $Lat_w^{\text{origin}} > Lat_w^{\text{prefetch}}$ when the condition below holds true:

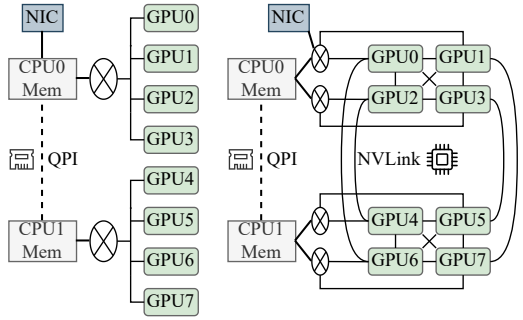
$$2(\varepsilon - 3\alpha H)B_{n,w}^i > \varepsilon\alpha H \tag{12}$$

Here, we use ε as shorthand for the compute-to-bandwidth ratio $\frac{P_w}{W_{n,w}}$ for conciseness. Prefetching is observed to enhance training efficiency only when ε exceeds $3\alpha H$, indicating a scenario with limited inter-worker bandwidth but potent GPU computing capability. For instance, NVIDIA DGX B200 is equipped with computing capacity of 72 petaFLOPS and NVLink connection with 400 Gb/s. Given a typical embedding size $H \in \{768, 1024, 2048, 4096\}$, it is clear that $\varepsilon > 3\alpha H$, namely expert prefetching is viable. In such cases, the tokens received by expert E_n^i, if prefetched, must satisfy

$$B_{n,w}^i > \frac{\varepsilon\alpha H}{2(\varepsilon - 3\alpha H)} \tag{13}$$

By combining Eq. (10) and Eq. (13), we prioritize prefetching experts based on their popularity until the GPU memory is fully utilized. The net time reduction achieved by prefetching a single expert is calculated as $4H\sum_{n=1}^{N}\frac{B_{n,w}^i}{W_{n,w}} - 2\alpha H^2\sum_{n=1}^{N}\frac{\delta_{n,w}^i}{W_{n,w}} - \frac{12\alpha H^2}{P_w}\sum_{n=1}^{N}B_{n,w}^i\delta_{n,w}^i$. As aforementioned in Figure 3, expert popularity tends to stabilize as training progresses. Therefore, in the mid-to-late stages of MoE training, it may not be necessary to assess each expert's prefetching needs in every iteration. Instead, a fixed prefetching strategy could be adopted, or the *frequency* of expert replanning is reduced, to further cut down on operational overhead.

Overview of PopFetcher mechanics. In summary, we illustrate the operational mechanics of PopFetcher. Conditioned on the end-to-end latency in Eq. (6), we aim to minimize the latency by searching the optimal expert prefetching solution. Given the exponentially growing complexity of the search space with increases in the number of machines and GPUs, we implement two key constraints to manage this complexity: GPU memory limitations as defined in Eq. (10) and constraints on expert transfer time as specified in Eq. (13). These constraints effectively prune the search space, allowing us to formulate a comprehensive global expert prefetching strategy. Specifically, for each set $\delta_{n,w}^i$ identified in our strategy, we execute prefetching of the i-th expert E_n^i on worker n to worker w during non-MoE layer computations.



(a) PCIe-only topology **(b)** NVLink-based topology

Figure 8: Different types of GPU machine topology.

5.3.2 Internal Expert Sharing among Local GPUs

In GPU clusters, links connecting various components often occupy *heterogeneous bandwidths*, as exemplified in Figure 8 which depicts the topology of a typical machine. Generally, GPUs within the same machine communicate through NVLink, with bandwidths reaching up to 1800GB/s, while those interconnected with the machine's main CPU via PCIe share a bandwidth of 64GB/s. For inter-server communication, machines are often linked through GDR NICs, with their bandwidths reaching 400Gb/s, where experts must be first pulled to local CPU memory via the NICs and then moved to the GPU memory through PCIe. Given this diverse network fabric, when a worker prefers to fetching experts from others, coupling with the heterogeneous connections can apparently enhance bandwidth utilization. Specifically, workers will prioritize sources connected by higher bandwidth links for retrieving their needed expert parameters.

Additionally, internal experts can be shared across all local workers using CPU memory. Once an expert is available on the same machine, PopFetcher bypasses the need for external fetching, i.e., workers directly access the expert from local CPU memory, to mitigate the overhead of expert prefetching. To achieve this, we maintain a cache manager for each server, which is responsible for sharing remote expert parameters among local workers and utilizes CPU memory as an intermediary to expedite the prefetching process.

In general, our scheduling scheme is designed to optimally

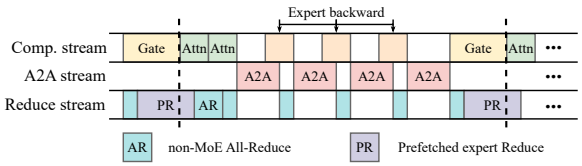


Figure 9: Pipeline scheduling of streams in backward pass.

retrieve experts through the fastest available links, regardless of the specific intra and inter-machine bandwidths. By leveraging the CPU memory, PopFetcher effectively circumvents the redundant transmission of expert parameters, thus enhancing the overall efficiency of MoE training system.

6 Stream Scheduling in Backward Pass

Due to expert prefetching, multiple copies of an expert may exist across different workers, requiring an additional All-Reduce operation during the backward pass. Typically, All-to-All communication for EP, All-Reduce gradient aggregation for non-MoE layers, and All-Reduce among prefetched experts are handled in *separate* process groups, which will initiate three distinct CUDA streams running concurrently. Without proper scheduling, these streams may compete for limited network bandwidth and GPU resources, potentially disrupting the normal flow of backward computation.

Ideally, strict priority should be assigned to All-to-All communication to minimize bandwidth contention. This strategy involves launching All-Reduce tasks as soon as they are ready to prevent delays, while also allowing All-to-All tasks to preemptively claim GPU resources upon initiation. However, implementing such precise stream scheduling is not feasible with highly optimized multi-GPU communication libraries like NCCL, which lock communication primitives into CUDA streams at the point of invocation. As a result, transmissions must be determined upfront, *precluding* any real-time adjustments in the use of core and bandwidth resources in conjunction with other operations.

To overcome these challenges, we propose decomposing both All-Reduce and All-to-All communications into micro-operations, which are executed in a pipelined fashion. In practice, expert prefetching leads to a reduced volume of All-to-All token transfers during the backward pass, diminishing their overlap with the computation of expert gradients. By strategically *interleaving* the micro-operations of All-Reduce and All-to-All communications, as vividly illustrated in Figure 9, we can fully capitalize on network idle periods when GPU workers are otherwise engaged.

7 Implementation

We have developed PopFetcher atop PyTorch, using Python, C++ and CUDA to construct the MoE training framework, which encompasses over 8,000 lines of code. The routing information collector, implemented using Python, aggregates runtime data via the All-Gather operator. Besides, the expert scheduling executor module utilizes the communication capabilities of torch.distributed, with the core logic for expert prefetching crafted in C++ and CUDA. Additionally, the executor facilitates parallel processing of computation and communication tasks through torch.cuda.Stream, and it manages a dedicated prefetching stream within the training system's prefetching interface.

For user convenience, PopFetcher is implemented as a PyTorch plugin that can function independently or be integrated into the Megatron-LM framework [34]. PopFetcher employs the torch.autograd.Function class, which allows for the customization of both forward and backward behaviors of the MoE operator, to ensuring its seamless integration into the MoE computational graph. All computation, communication, and prefetching activities are encapsulated within this custom MoE operator. Moreover, the pipeline scheduling of computation and communication streams is implemented in C++ and CUDA to optimize performance.

Table 3: Configuration of different MoE models.

Model	α	Hidden	Layer	Batch	Expert	Worker	Cluster
MoE-GPT	2	1024	12	32	32	8	*A*
				64	64	32	*B*
MoE-BERT	2	1024	12	32	32	8	*A*
				64	64	32	*B*

8 Evaluation

In this section, we evaluate the performance of PopFetcher in terms of token throughput and per-iteration time for different MoE models on real-world datasets.

8.1 Evaluation Setup

8.1.1 Hardware Setup

We evaluate PopFetcher on two GPU clusters.

- Cluster A. This GPU cluster comprises two machines, each outfitted with four NVIDIA RTX 4090 GPUs that have 24GB of memory per GPU. The GPU nodes are connected through a 100Gbps Mellanox ConnectX-5 InfiniBand adaptor for communication.

- Cluster B. The cluster includes eight machines, each equipped with four NVIDIA A10 GPUs, totaling 32 GPUs, with each GPU also featuring 24GB of memory. The inter-node communication operates over link with a maximum speed of 32Gbps bandwidth.

Besides, PyTorch version is 2.3 and CUDA version is 12.4. The communication backend is based on NCCL.

8.1.2 Datasets and Models

We test PopFetcher on two typical MoE models: (1) MoE-GPT [41], (2) MoE-BERT [16], which serve as representatives of PLMs that utilize Transformer decoder or encoder architecture. More details about model configurations are listed in Table 3. Both MoE-GPT and MoE-BERT are initially trained on the OpenWebText dataset [15]. Additionally, MoE-BERT undergoes further training on the significantly larger PILE dataset [14], which comprises approximately 600GB of data. We also conduct comparative experiments using the OSCAR-2201 dataset [1] to further assess the performance of PopFetcher under varying data conditions.

8.1.3 Baselines

We compare our PopFetcher with the following frameworks.

- DeepSpeed [30]: a high-performance system that supports massive scale MoE models.
- FasterMoE [16]: an easy-to-use and efficient MoE training system which simplifies the management of experts by shadowing them across GPU workers.
- Megablocks [13]: a lightweight library for MoE training, featuring a block-sparse GPU kernel that handles dynamic token routing and avoids token dropping without sacrificing the hardware efficiency.
- Tutel [17]: an optimized MoE system which achieves adaptive expert execution and adaptive pipelining for dispatch and combine operations in MoE layers.
- Janus [24]: a data-centric MoE training system which enables pulling experts to tokens. We replicated its functionality by prefetching all experts in our experiments, as the source code is not publicly available.

All baselines can be integrated into Megatron-LM [27, 34], and we conduct our comparisons using the same hardware and software environments to ensure fairness.

8.1.4 Metrics

For a comprehensive evaluation of performance, we employ two key metrics, i.e., token throughput and per-iteration time. In particular, token throughput characterizes the number of tokens processed by the training system in each iteration, reflecting the overall training efficiency. Per-iteration time quantifies the total time required to complete one forward and backward pass, with shorter iteration time indicating faster training speed and higher resource utilization.

8.2 Statistical Training Equivalence

PopFetcher significantly reduces the per-iteration time required for MoE training by prefetching popular experts, thereby accelerating the overall training process. To maintain training equivalence with traditional MoE methods, we conduct an additional reduction operation that synchronizes the gradient updates between the prefetched experts and the original experts. To substantiate this approach, we perform training sessions for MoE-GPT on Cluster A using both the conventional FasterMoE and our PopFetcher enhancement.

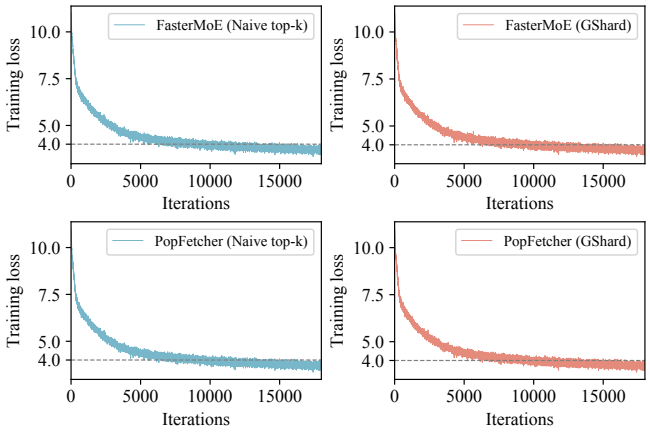


Figure 10: Loss value over training iterations.

The loss curves for both the naive top-k and GShard expert routing mechanisms are illustrated in Figure 10. The upper section of the figure depicts the original FasterMoE training scheme, whereas the lower section represents the enhancements made by PopFetcher. It is evident that regardless of the gating mechanism employed, be it naive or GShard, PopFetcher maintains the integrity of the expert selection process and does not introduce any additional steps in the convergence process. Consequently, all loss trajectories align perfectly across training iterations, demonstrating that PopFetcher achieves statistical equivalence with the standard training framework without compromising model accuracy.

8.3 Overall End-to-End Performance

Initially, we evaluate the end-to-end performance, focusing on the training time per iteration of our PopFetcher and various baselines. All results are obtained using GShard gating, given its broader application.

Figure 11 shows PopFetcher's overall speedup relative to baselines across different GPU clusters, calculated against the slowest system in each setup. We observe that PopFetcher boosts training speed by 1.28-2.4× and 1.18-18.3× in the two clusters, respectively. These gains stem from its expert prefetching design and network stream prioritization, which concurrently fetches popular expert parameters during non-MoE layer computations without blocking the backward pass. Since Janus employs a solution that pulls all expert parameters, it imposes extremely high demands on GPU memory capacity. Given the settings described, Janus often results in OOM errors and is therefore excluded from Figure 11.

PopFetcher consistently achieves robust training speeds, even in low inter-machine network bandwidth environments like Cluster B. In contrast, frameworks such as Megablocks,

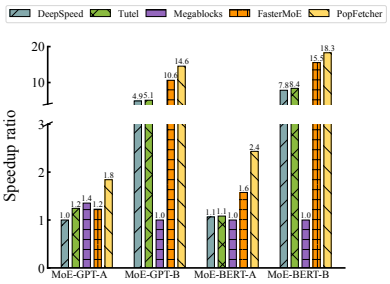


Figure 11: Speedups over baselines.

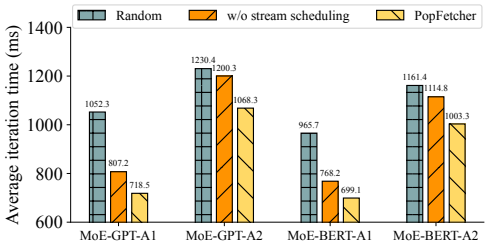


Figure 12: Expert and stream scheduling.

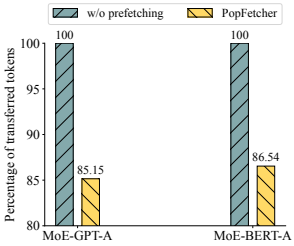


Figure 13: Token transfers.

heavily reliant on the inter-machine communication, can experience significant efficiency losses under similar conditions, resulting in slower training speeds. This contrast highlights PopFetcher's suitability for bandwidth-constrained hardware environments, such as those using consumer-grade GPUs. In these scenarios, with a high compute-to-bandwidth ratio ε, PopFetcher is able to effectively utilize the idle links during non-MoE layer computations.

8.4 Ablation Study

8.4.1 Expert Popularity and Stream Scheduling

We continue evaluating the benefits of expert popularity prediction and backward stream scheduling. We compare PopFetcher with baseline MoE training approaches that rely on random expert prefetching or built-in stream strategy, which ignore variations in expert selection or stream priorities, respectively. Our evaluations employ MoE-GPT and MoE-BERT models in Cluster A, varying the number of experts per layer for a comprehensive analysis: MoE-GPT-A1 includes 16 experts, MoE-GPT-A2 has 8, with similar setups for the MoE-BERT model.

Figure 12 shows the average per-iteration time. In particular, our popularity-based expert scheduling yields significant speedup enhancements, achieving 1.30× and 1.26× faster training for MoE-GPT and MoE-BERT models, respectively, compared to random prefetching. Additionally, PopFetcher cuts the average iteration time by 10.9% for MoE-GPT and 10% for MoE-BERT, highlighting substantial efficiency gains from incorporating pipelined stream scheduling during the backward pass, thus avoiding network contention and removing computation blockage.

8.4.2 Token Transfer within Expert Prefetching

Expert prefetching in PopFetcher reduces token transfers in All-to-All communications, thus cutting the communication time. As shown in Figure 13, implementing our prefetching strategy globally decreases tokens transmitted by 14.85% for MoE-GPT and 13.46% for MoE-BERT in Cluster A.

Furthermore, we assess the prefetching status of experts in MoE-GPT and MoE-BERT models (16 experts per layer), as depicted in Figure 14. The vertical axis shows the global index of experts, and the horizontal axis represents iterations. Darker cell colors indicate more frequent prefetching. Results illustrate that strategic prefetching of certain experts can speed up training, with these experts consistently requested by multiple workers across consecutive training iterations. That is, expert popularity exhibits "locality" (discussed in Section 2.2), which is seen as long, dark bands in the figure. These popular experts vary per layer, but consistently, at least one such hot expert appears in every layer, highlighting a focused prefetching strategy to boost model performance.

8.5 Hybrid Push-Pull Analysis

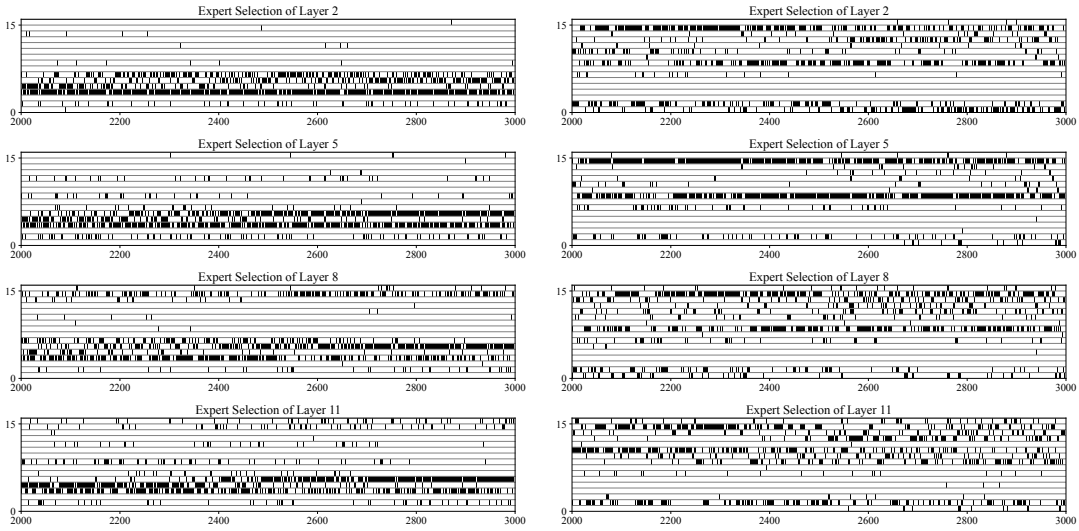
PopFetcher integrates the advantages of both traditional data-centric and expert-centric MoE training by adopting a hybrid approach that involves sending tokens and pulling experts. To demonstrate the superiority of PopFetcher, we compare its training throughput and per-iteration time with those of data-centric (Janus) and expert-centric systems (w/o prefetching).

As depicted in Figure 15, in experiments conducted on Cluster A using MoE-GPT and MoE-BERT models, PopFetcher surpasses both the expert-centric and data-centric paradigms in training throughput and per-iteration time. This outcome underscores the efficacy of our hybrid communication method which combines sending tokens and pulling experts, optimizing data transmission volumes in All-to-All communications. Our strategy fine-tunes the balance of load and addresses the granularity issues associated with load imbalances in All-to-All communications, thereby enhancing overall MoE training efficiency.

"Bad prefetching" arises when prefetched experts receive fewer tokens than alternatives, reducing All-to-All communication savings. However, PopFetcher effectively mitigates this issue by accurately prefetching popular experts using a sliding window and inter-layer correlation. Should "bad prefetching" occur, PopFetcher simply reverts to the conventional training mode without incurring additional overhead, as prefetch operations overlap with non-MoE computations.

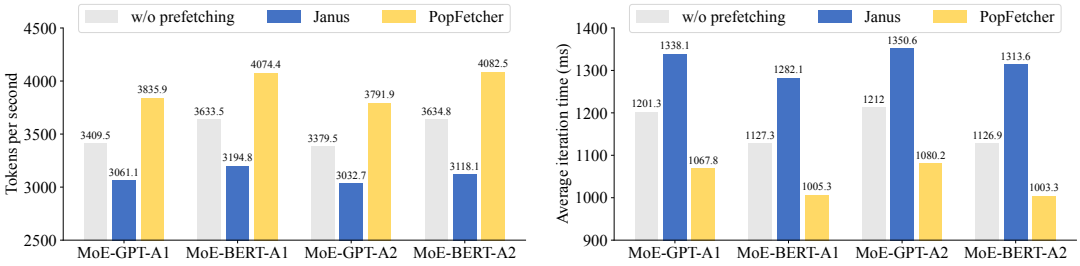
8.6 Parameter Sensitivity Analysis

The performance of PopFetcher is closely linked to the prediction accuracy of expert popularity, which is influenced by the sliding window size s. A larger s does not necessarily lead to



(a) MoE-GPT-A (b) MoE-BERT-A

Figure 14: Demonstrations of prefetched experts in PopFetcher over training iterations.



(a) Training throughput (b) Iteration time

Figure 15: Effect of PopFetcher's hybrid token send and expert pull.

Table 4: Prediction accuracy over sliding window size.

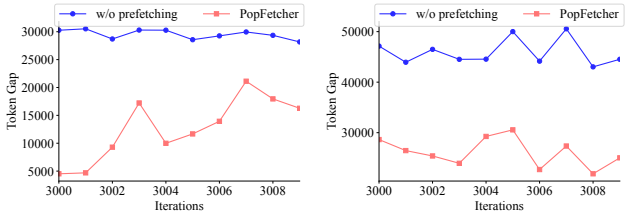
Acc. / Value / Gate	5	10	15	20	25	50	100
Naive top-k	77.46%	77.04%	76.77%	76.37%	76.25%	75.69%	75.31%
GShard	68.13%	69.62%	57.26%	53.66%	51.19%	46.49%	45.30%

better accuracy, but it does require maintaining an extensive history of past records, resulting in inefficient resource utilization without significant improvements in prediction accuracy.

Given the number of experts prefetched per layer is typically no more than four due to GPU memory constraint, we focus on measuring the accuracy of predicted top-5 popular experts against the actual top-5 hot experts. This evaluation is further averaged across all MoE layers. As depicted in Table 4, after assessing the prediction accuracy under various configurations, we select a sliding window size of 10 for runtime operations to achieve an optimal balance between prediction performance and memory consumption.

8.7 Balanced GPU Workload

By prefetching popular experts, PopFetcher ensures a more balanced workload distribution across workers, since the loads of hot experts are shared among multiple workers. As illustrated in Figure 16, it significantly reduces the disparity in token distribution between the least and most loaded workers: 43.1% for MoE-GPT and 57.1% for MoE-BERT. This uniform workload distribution during training prevents the communication blockages common in traditional MoE systems, where biased expert selections often cause workload imbalances and network congestion.



(a) MoE-GPT-A (b) MoE-BERT-A

Figure 16: Difference in received number of tokens between lightest and heaviest loaded workers (16 experts per layer).

8.8 GPU Memory Consumption

Through hybrid token dispatch and expert retrieval, PopFetcher achieves more efficient utilization of limited GPU memory. To illustrate, we compare the accommodated

Table 5: Maximum accommodated model size.

Size / Model / Method	MoE-GPT	MoE-BERT
FasterMoE	1.844B	1.884B
Janus	1.390B	1.430B
PopFetcher	**2.071B**	**2.262B**

parameter size of PopFetcher with those of FasterMoE and Janus by adjusting the number of experts in each MoE layer. Table 5 presents the maximum configurations and parameter scales that the MoE-GPT and MoE-BERT models can train on Cluster A. Compared to FasterMoE, PopFetcher demonstrates improvements of 12.3% and 20.1% in model size on the MoE-GPT and MoE-BERT models, respectively. Similarly, compared to Janus, PopFetcher achieves enhancements of 49.0% and 58.2% on the MoE-GPT and MoE-BERT models, respectively. These results underscore that the hybrid push-pull paradigm can effectively exploit GPU memory.

8.9 Runtime Overhead

PopFetcher introduces minimal runtime overhead, largely because the expert popularity prediction and the search for prefetching solutions are performed asynchronously on the CPU. These tasks run in the background without disrupting the main training process. Moreover, our pruning strategy significantly reduces the search space, rendering the computational cost negligible. In practical tests, the overhead for popularity prediction is less than 100ms, which is readily absorbed within the total training time. Given that the benefits of prefetching popular experts far outweigh this slight overhead, PopFetcher efficiently accelerates the training process while minimizing its negative impact.

9 Related Work

Sparse MoE model. Large-scale models have excelled in computer vision [39] and natural language processing [11], thanks to extensive parameters. MoE architectures, introduced in a 1991 paper [18], further boost parameter scales and capabilities while maintaining computational efficiency [26]. Shazeer *et al.* expands MoE to LSTMs with 137 billion parameters and 131,072 experts [31]. GShard adapts MoE for transformer models, achieving 12.5 to 600 billion parameters [21]. Switch Transformers enhances the T5 model by replacing Feed-Forward layers with MoE layers and optimizing routing strategy [12]. ST-MoE further develops an encoder-decoder MoE architecture with 269 billion parameters, 32 billion active at any time [43]. GLaM introduces a decoder-only model with a record 1.2 trillion size [10].

MoE model training. Switch Transformers pioneer EP by distributing individual experts across GPUs within the same MoE layer [12]. Tutel on the other hand emphasizes MoE's dynamic nature by optimizing token-expert allocation for data, model, and expert parallelism [17]. To enhance MoE training, Janus modifies the traditional All-to-All communication in EP by routing experts to where tokens are located [24]. Besides, Lina prioritizes All-to-All communication over typical All-Reduce operations while adoting data parallelism [23]. FasterMoE introduces dynamic load balancing through expert replication and a flexible pipeline scheduling strategy [16]. ScheMoE further develops a scheduling framework that manages communication and computation tasks, integrating an advanced All-to-All communication primitive to maximize bandwidth use [33]. These works significantly enhance the efficiency, scalability, and performance of MoE training systems through optimizing expert-token interactions and improving load balancing in large-scale distributed setups.

10 Conclusion

In this paper, we propose PopFetcher, an effective and scalable MoE training framework that incorporates popularity based expert-wise prefetching, to mitigate the All-to-All communication bottleneck in MoE training. PopFetcher identifies skewed and correlated patterns in expert selection and employs a sliding-window based prediction to determine the popularity of experts in upcoming layers. Building on this statistics, PopFetcher meticulously analyzes the end-to-end training latency of each MoE layer, and establishes an optimal mapping of prefetched experts to GPU workers. By doing so, PopFetcher not only minimizes communication time but also ensures a balanced workload distribution among workers. Additionally, PopFetcher exploits network idle periods to pipeline All-Reduce streams and non-MoE computation, thereby expediting the backward pass process. Experimental evaluations on real-world datasets show that PopFetcher outperforms state-of-the-art MoE training systems, as we can mitigate the training time by 15%-94.5%.

Acknowledgement

This work was supported in part by NSFC-RGC under Grant 62461160333, Wuhan Natural Science Foundation Exploratory Program (Chenguang Program) under Grant 2024040801020210, National Natural Science Foundation of China under Grants 62202185, an RGC RIF grant under contract R6021-20, and an RGC TRS grant under contract T43-513/23N-2.

References

[1] Julien Abadji, Pedro Javier Ortiz Suárez, Laurent Romary, and Benoît Sagot. Towards a cleaner document-oriented multilingual crawled corpus. In *Proceedings of the Thirteenth Language Resources and Evaluation Conference*, pages 4344–4355, 2022.

[2] Marah I Abdin, Sam Ade Jacobs, Ammar Ahmad Awan, Jyoti Aneja, Ahmed Awadallah, Hany Awadalla, Nguyen Bach, Amit Bahree, Arash Bakhtiari, Harkirat S. Behl, Alon Benhaim, Misha Bilenko, Johan Bjorck, Sébastien Bubeck, Martin Cai, Caio César Teodoro Mendes, Weizhu Chen, Vishrav Chaudhary, Parul Chopra, Allie Del Giorno, Gustavo de Rosa, Matthew Dixon, Ronen Eldan, Dan Iter, Amit Garg, Abhishek Goswami, Suriya Gunasekar, Emman Haider, Junheng Hao, Russell J. Hewett, Jamie Huynh, Mojan Javaheripi, Xin Jin, Piero Kauffmann, Nikos Karampatziakis, Dongwoo Kim, Mahoud Khademi, Lev Kurilenko, James R. Lee, Yin Tat Lee, Yuanzhi Li, Chen Liang, Weishung Liu, Eric Lin, Zeqi Lin, Piyush Madan, Arindam Mitra, Hardik Modi, Anh Nguyen, Brandon Norick, Barun Patra, Daniel Perez-Becker, Thomas Portet, Reid Pryzant, Heyang Qin, Marko Radmilac, Corby Rosset, Sambudha Roy, Olatunji Ruwase, Olli Saarikivi, Amin Saied, Adil Salim, Michael Santacroce, Shital Shah, Ning Shang, Hiteshi Sharma, Xia Song, Masahiro Tanaka, Xin Wang, Rachel Ward, Guanhua Wang, Philipp A. Witte, Michael Wyatt, Can Xu, Jiahang Xu, Sonali Yadav, Fan Yang, Ziyi Yang, Donghan Yu, Chengruidong Zhang, Cyril Zhang, Jianwen Zhang, Li Lyna Zhang, Yi Zhang, Yue Zhang, Yunan Zhang, and Xiren Zhou. Phi-3 technical report: A highly capable language model locally on your phone. *CoRR*, abs/2404.14219, 2024.

[3] Roee Aharoni, Melvin Johnson, and Orhan Firat. Massively multilingual neural machine translation. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Long and Short Papers)*, volume 1, pages 3874–3884, 2019.

[4] Mikel Artetxe, Shruti Bhosale, Naman Goyal, Todor Mihaylov, Myle Ott, Sam Shleifer, Xi Victoria Lin, Jingfei Du, Srinivasan Iyer, Ramakanth Pasunuru, Giridharan Anantharaman, Xian Li, Shuohui Chen, Halil Akin, Mandeep Baines, Louis Martin, Xing Zhou, Punit Singh Koura, Brian O'Horo, Jeffrey Wang, Luke Zettlemoyer, Mona T. Diab, Zornitsa Kozareva, and Veselin Stoyanov. Efficient large scale language modeling with mixtures of experts. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pages 11699–11732, 2022.

[5] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901, 2020.

[6] Peizhuang Cong, Aomufei Yuan, Shimao Chen, Yuxuan Tian, Bowen Ye, and Tong Yang. Prediction is all moe needs: Expert load distribution goes from fluctuating to stabilizing. *CoRR*, abs/2404.16914, 2024.

[7] DeepSeek-AI, Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, Damai Dai, Daya Guo, Dejian Yang, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Han Bao, Hanwei Xu, Haocheng Wang, Haowei Zhang, Honghui Ding, Huajian Xin, Huazuo Gao, Hui Li, Hui Qu, J. L. Cai, Jian Liang, Jianzhong Guo, Jiaqi Ni, Jiashi Li, Jiawei Wang, Jin Chen, Jingchang Chen, Jingyang Yuan, Junjie Qiu, Junlong Li, Junxiao Song, Kai Dong, Kai Hu, Kaige Gao, Kang Guan, Kexin Huang, Kuai Yu, Lean Wang, Lecong Zhang, Lei Xu, Leyi Xia, Liang Zhao, Litong Wang, Liyue Zhang, Meng Li, Miaojun Wang, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Mingming Li, Ning Tian, Panpan Huang, Peiyi Wang, Peng Zhang, Qiancheng Wang, Qihao Zhu, Qinyu Chen, Qiushi Du, R. J. Chen, R. L. Jin, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, Runxin Xu, Ruoyu Zhang, Ruyi Chen, S. S. Li, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shaoqing Wu, Shengfeng Ye, Shengfeng Ye, Shirong Ma, Shiyu Wang, Shuang Zhou, Shuiping Yu, Shunfeng Zhou, Shuting Pan, T. Wang, Tao Yun, Tian Pei, Tianyu Sun, W. L. Xiao, and Wangding Zeng. Deepseek-v3 technical report. *CoRR*, abs/2412.19437, 2024.

[8] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Long and Short Papers)*, volume 1, pages 4171–4186, 2019.

[9] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and

Neil Houlsby. An image is worth 16x16 words: Transformers for image recognition at scale. In *9th International Conference on Learning Representations*, 2021.

[10] Nan Du, Yanping Huang, Andrew M. Dai, Simon Tong, Dmitry Lepikhin, Yuanzhong Xu, Maxim Krikun, Yanqi Zhou, Adams Wei Yu, Orhan Firat, Barret Zoph, Liam Fedus, Maarten P. Bosma, Zongwei Zhou, Tao Wang, Yu Emma Wang, Kellie Webster, Marie Pellat, Kevin Robinson, Kathleen S. Meier-Hellstern, Toju Duke, Lucas Dixon, Kun Zhang, Quoc V. Le, Yonghui Wu, Zhifeng Chen, and Claire Cui. Glam: Efficient scaling of language models with mixture-of-experts. In *International Conference on Machine Learning*, volume 162, pages 5547–5569, 2022.

[11] Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, Anirudh Goyal, Anthony Hartshorn, Aobo Yang, Archi Mitra, Archie Sravankumar, Artem Korenev, Arthur Hinsvark, Arun Rao, Aston Zhang, Aurélien Rodriguez, Austen Gregerson, Ava Spataru, Baptiste Rozière, Bethany Biron, Binh Tang, Bobbie Chern, Charlotte Caucheteux, Chaya Nayak, Chloe Bi, Chris Marra, Chris McConnell, Christian Keller, Christophe Touret, Chunyang Wu, Corinne Wong, Cristian Canton Ferrer, Cyrus Nikolaidis, Damien Allonsius, Daniel Song, Danielle Pintz, Danny Livshits, David Esiobu, Dhruv Choudhary, Dhruv Mahajan, Diego Garcia-Olano, Diego Perino, Dieuwke Hupkes, Egor Lakomkin, Ehab AlBadawy, Elina Lobanova, Emily Dinan, Eric Michael Smith, Filip Radenovic, Frank Zhang, Gabriel Synnaeve, Gabrielle Lee, Georgia Lewis Anderson, Graeme Nail, Grégoire Mialon, Guan Pang, Guillem Cucurell, Hailey Nguyen, Hannah Korevaar, Hu Xu, Hugo Touvron, Iliyan Zarov, Imanol Arrieta Ibarra, Isabel M. Kloumann, Ishan Misra, Ivan Evtimov, Jade Copet, Jaewon Lee, Jan Geffert, Jana Vranes, Jason Park, Jay Mahadeokar, Jeet Shah, Jelmer van der Linde, Jennifer Billock, Jenny Hong, Jenya Lee, Jeremy Fu, Jianfeng Chi, Jianyu Huang, Jiawen Liu, Jie Wang, Jiecao Yu, Joanna Bitton, Joe Spisak, Jongsoo Park, Joseph Rocca, Joshua Johnstun, Joshua Saxe, Junteng Jia, Kalyan Vasuden Alwala, Kartikeya Upasani, Kate Plawiak, Ke Li, Kenneth Heafield, Kevin Stone, and et al. The llama 3 herd of models. *CoRR*, abs/2407.21783, 2024.

[12] William Fedus, Barret Zoph, and Noam Shazeer. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *Journal of Machine Learning Research*, 23(120):1–39, 2022.

[13] Trevor Gale, Deepak Narayanan, Cliff Young, and Matei Zaharia. Megablocks: Efficient sparse training with mixture-of-experts. In *Proceedings of the Sixth Conference on Machine Learning and Systems*, volume 5, pages 288–304, 2023.

[14] Leo Gao, Stella Biderman, Sid Black, Laurence Golding, Travis Hoppe, Charles Foster, Jason Phang, Horace He, Anish Thite, Noa Nabeshima, Shawn Presser, and Connor Leahy. The pile: An 800gb dataset of diverse text for language modeling. *CoRR*, abs/2101.00027, 2021.

[15] Aaron Gokaslan and Vanya Cohen. Openwebtext corpus. http://Skylion007.github.io/OpenWebTextCorpus, 2019.

[16] Jiaao He, Jidong Zhai, Tiago Antunes, Haojie Wang, Fuwen Luo, Shangfeng Shi, and Qin Li. Fastermoe: modeling and optimizing training of large-scale dynamic pre-trained models. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 120–134, 2022.

[17] Changho Hwang, Wei Cui, Yifan Xiong, Ziyue Yang, Ze Liu, Han Hu, Zilong Wang, Rafael Salas, Jithin Jose, Prabhat Ram, HoYuen Chau, Peng Cheng, Fan Yang, Mao Yang, and Yongqiang Xiong. Tutel: Adaptive mixture-of-experts at scale. In *Proceedings of the Sixth Conference on Machine Learning and Systems*, volume 5, pages 269–287, 2023.

[18] Robert A. Jacobs, Michael I. Jordan, Steven J. Nowlan, and Geoffrey E. Hinton. Adaptive mixtures of local experts. *Neural computation*, 3(1):79–87, 1991.

[19] Yimin Jiang, Yibo Zhu, Chang Lan, Bairen Yi, Yong Cui, and Chuanxiong Guo. A unified architecture for accelerating distributed DNN training in heterogeneous GPU/CPU clusters. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 463–479, 2020.

[20] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models. *CoRR*, abs/2001.08361, 2020.

[21] Dmitry Lepikhin, HyoukJoong Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Yanping Huang, Maxim Krikun, Noam Shazeer, and Zhifeng Chen. Gshard: Scaling giant models with conditional computation and automatic sharding. In *9th International Conference on Learning Representations*, 2021.

[22] Mike Lewis, Shruti Bhosale, Tim Dettmers, Naman Goyal, and Luke Zettlemoyer. BASE layers: Simplifying training of large, sparse models. In *Proceedings of the 38th International Conference on Machine Learning*, volume 139, pages 6265–6274, 2021.

[23] Jiamin Li, Yimin Jiang, Yibo Zhu, Cong Wang, and Hong Xu. Accelerating distributed moe training and inference with lina. In *Proceedings of the 2023 USENIX Annual Technical Conference*, pages 945–959, 2023.

[24] Juncai Liu, Jessie Hui Wang, and Yimin Jiang. Janus: A unified distributed training framework for sparse mixture-of-experts models. In *Proceedings of the ACM SIGCOMM 2023 Conference*, pages 486–498, 2023.

[25] Ze Liu, Yutong Lin, Yue Cao, Han Hu, Yixuan Wei, Zheng Zhang, Stephen Lin, and Baining Guo. Swin transformer: Hierarchical vision transformer using shifted windows. In *Proceedings of the IEEE/CVF international conference on computer vision*, pages 10012–10022, 2021.

[26] Saeed Masoudnia and Reza Ebrahimpour. Mixture of experts: a literature survey. *Artificial Intelligence Review*, 42(2):275–293, 2014.

[27] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, Amar Phanishayee, and Matei Zaharia. Efficient large-scale language model training on GPU clusters using megatron-lm. In *International Conference for High Performance Computing, Networking, Storage and Analysis*, page 58, 2021.

[28] Xiaonan Nie, Xupeng Miao, Zilong Wang, Zichao Yang, Jilong Xue, Lingxiao Ma, Gang Cao, and Bin Cui. Flexmoe: Scaling large-scale sparse pre-trained model training via dynamic device placement. *Proceedings of the ACM on Management of Data*, 1(1):1–19, 2023.

[29] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

[30] Samyam Rajbhandari, Conglong Li, Zhewei Yao, Minjia Zhang, Reza Yazdani Aminabadi, Ammar Ahmad Awan, Jeff Rasley, and Yuxiong He. Deepspeed-moe: Advancing mixture-of-experts inference and training to power next-generation AI scale. In *International Conference on Machine Learning*, volume 162, pages 18332–18346, 2022.

[31] Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc V. Le, Geoffrey E. Hinton, and Jeff Dean. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. In *5th International Conference on Learning Representations*, 2017.

[32] Shaohuai Shi, Xinglin Pan, Xiaowen Chu, and Bo Li. Pipemoe: Accelerating mixture-of-experts through adaptive pipelining. In *IEEE INFOCOM 2023 - IEEE Conference on Computer Communications*, pages 1–10, 2023.

[33] Shaohuai Shi, Xinglin Pan, Qiang Wang, Chengjian Liu, Xiaozhe Ren, Zhongzhe Hu, Yu Yang, Bo Li, and Xiaowen Chu. Schemoe: An extensible mixture-of-experts distributed training system with tasks scheduling. In *Proceedings of the Nineteenth European Conference on Computer Systems*, pages 236–249, 2024.

[34] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *CoRR*, abs/1909.08053, 2019.

[35] Qwen Team. Qwen1.5-moe: Matching 7b model performance with 1/3 activated parameters", February 2024.

[36] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurélien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. Llama: Open and efficient foundation language models. *CoRR*, abs/2302.13971, 2023.

[37] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton-Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurélien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. Llama 2: Open foundation and fine-tuned chat models. *CoRR*, abs/2307.09288, 2023.

[38] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems*, volume 30, pages 5998–6008, 2017.

[39] Wenhai Wang, Zhe Chen, Xiaokang Chen, Jiannan Wu, Xizhou Zhu, Gang Zeng, Ping Luo, Tong Lu, Jie Zhou, Yu Qiao, and Jifeng Dai. Visionllm: Large language model is also an open-ended decoder for vision-centric tasks. In *Advances in Neural Information Processing Systems*, volume 36, pages 61501–61513, 2023.

[40] Zhiyuan Zeng and Deyi Xiong. Scomoe: Efficient mixtures of experts with structured communication. In *The Eleventh International Conference on Learning Representations*, 2023.

[41] Zheng Zhang, Yaqi Xia, Hulin Wang, Donglin Yang, Chuang Hu, Xiaobo Zhou, and Dazhao Cheng. Mpmoe: Memory efficient moe for pre-trained models with adaptive pipeline parallelism. *IEEE Trans. Parallel Distributed Syst.*, 35(6):843–856, 2024.

[42] Yanqi Zhou, Tao Lei, Hanxiao Liu, Nan Du, Yanping Huang, Vincent Y. Zhao, Andrew M. Dai, Zhifeng Chen, Quoc V. Le, and James Laudon. Mixture-of-experts with expert choice routing. In *Advances in Neural Information Processing Systems*, volume 35, pages 7103–7114, 2022.

[43] Barret Zoph, Irwan Bello, Sameer Kumar, Nan Du, Yanping Huang, Jeff Dean, Noam Shazeer, and William Fedus. St-moe: Designing stable and transferable sparse expert models. *CoRR*, abs/2202.08906, 2022.

[44] Simiao Zuo, Xiaodong Liu, Jian Jiao, Young Jin Kim, Hany Hassan, Ruofei Zhang, Jianfeng Gao, and Tuo Zhao. Taming sparsely activated transformer with stochastic experts. In *The Tenth International Conference on Learning Representations*, 2022.

HYPERECA: Distributed Heterogeneous In-Memory Embedding Database for Training Recommender Models

Jiaao He Shengqi Chen Kezhao Huang Jidong Zhai

Tsinghua University

Abstract

Making high-quality recommendations is important in online applications. To improve user satisfaction and effectiveness of advertising, deep learning-based recommender models (DLRM) are widely studied and deployed. Training these models on massive data demands increasing computation power, commonly provided by a cluster of numerous GPUs. Meanwhile, the embedding tables of the models are huge, posing challenges on the memory. Existing systems exploit host memory and hashing techniques to accommodate them. However, the simple offloading design is hard to scale up to multiple nodes. The sparse access to the distributed embedding tables introduces high data management and all-to-all communication overhead.

We find that a distributed in-memory key-value database is the best abstraction to serve and maintain embedding vectors in DLRM training. To achieve high scalability, our system, *HypeReca*, utilizes both GPU and CPU memory. We improve the throughput of data management according to the batching pattern of DNN training, using a pipeline over decentralized indexing tables and a contention-avoiding schedule for data exchange. A two-fold parallel strategy is used to guarantee consistency of all embedding vectors. The communication overhead is reduced by replicating a few frequently accessed embedding vectors, exploiting the sparse pattern with a performance model. In our evaluation on 32 GPUs over real-world datasets, *HypeReca* achieves $2.16-16.8\times$ end-to-end speedup over HugeCTR, TorchRec and TFDE. The source code is available at https://github.com/thu-pacman/hypereca/.

1 Introduction

Making recommendations has always been one of the core components of the Internet. Users are served with blog posts and videos, products and advertisements as long as they are surfing online. Making more accurate recommendations is core of the business. In recent years, recommendation models based on deep learning (DLRMs) are showing superior model quality. They automatically learn from the enormous data generated by users' online behaviors. For instance, websites predict the click-through rate (CTR) of a video to decide whether to show it to a user. And then, actual click data is used to train the CTR-predicting model. This training cycle is processed every few minutes in real-world systems. To provide up-to-date recommendation, training performance of these models is critical. Training them demands a large amount of data processed by significant computational resources, e.g., up to hundreds of GPUs.

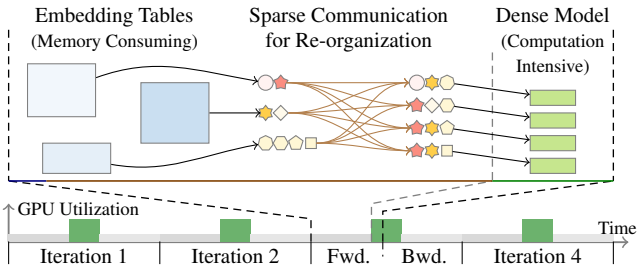


Figure 1: Sketch of a DLRM Training Process

As shown in Figure 1, the *sparse part* consists of embedding tables that contain the feature vectors of numerous items. To train the model, the related feature vectors of items in the training samples are gathered from the embedding tables via sparse inter-process communication. After receiving all the embedding vectors, the GPUs process them with the *dense part* of the model.

Typically, a hybrid parallel strategy is adopted to train the model. The memory-consuming embedding tables are partitioned across workers, and the host memory is commonly used to accommodate most of them. They may contain up to trillions of parameters [21, 25] in production. Rest layers of the model, the dense part, are replicated. They model is replicated on up to hundreds of GPUs, which provide the computation capability to train it.

Unfortunately, such parallel strategy fails to scale up, due

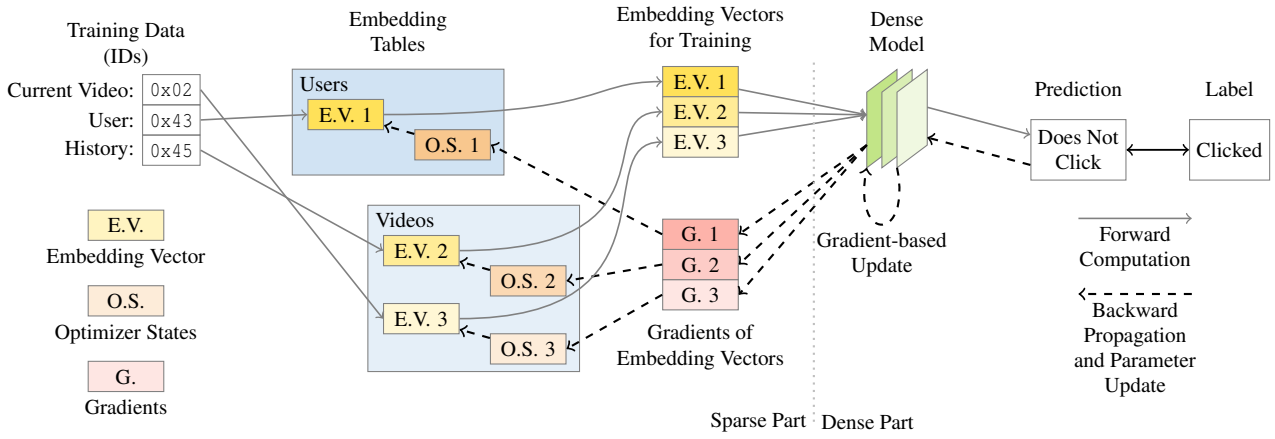


Figure 2: General Model Structure and Training Process of DLRMs

to the huge size and heterogeneity of the model parts. Every training sample is related to tens of embedding vectors that reside in several other workers, incurring significant overhead to exchange the embedding vectors among the hardware devices. From the ranklist of DLRM training benchmark [41] of MLPerf [23], we observe that the most advanced system from NVIDIA [39] only achieves 2.89× speed up using 13× more GPUs (from 8 to 112). We even find that using more GPUs may slow down the training in some cases.

A series of prior works introduce algorithms that change the training behavior to alleviate this problem, e.g., shuffling the training samples [2] or dropping certain embedding vectors [20]. Although higher throughput might be achieved, they pose uncertain influences on final model quality. Model developers thus usually appreciate systems that train the model without changing the algorithm to avoid any side effect. Therefore, we focus on improving the efficiency of the original synchronous training method.

We identify that existing model-transparent approaches fail to cope with the huge size and heterogeneity of the models due to the following two challenges.

High Data Management Cost Different from typical embedding tables in other models, embedding vectors of new items are constantly inserted to the tables, because there are always new items in the training data. Among the distributed embedding tables, overhead of the item-wise data management is high [43]. Current approaches maintain the embedding vectors differently at either table level [25, 33] or item level [3, 34] according to the access pattern. The system has to maintain locations of every embedding vector and guarantee the consistency. The GPUs consume up to millions of items every few milliseconds, posing high throughput requirement on the CPU-side embedding feeding and updating module.

High Communication Cost The communication overhead to gather the embedding vectors from remote nodes and sending the gradients back takes a significant amount of time, comparing with the neural computation on the GPUs. Uneven access pattern has been exploited by previous works [3, 25, 33, 34, 44] to accelerate the communication by replicating some frequently accessed items (e.g. items represented by stars in Figure 1) across all GPUs. However, this introduces additional overhead to synchronize the replicas, and the total commjjunication overhead can hardly be reduced. Selecting the items to replicate is critical but challenging.

We propose an in-memory embedding database, *HypeReca*, to address the challenges in the distributed DLRM training scenario. Behind a simple interface similar to a key-value database, it automatically utilizes both host and device memory in the cluster to store the embedding data, and serve it with different parallel strategies according to the specific pattern of DNN training and its data.

First, we design an efficient decentralized embedding vector management mechanism. The vectors are stored in chunks, and the locations are maintained by a distributed indexing table. As training data are fed to the model in batches during the training process, we utilize the multi-thread parallel processing capability for the throughput-critical indexing tasks. Furthermore, we introduce an intra-node pipeline to reduce the lock overhead over the tables.

Second, we adopt a two-fold parallel strategy for the embedding tables, namely ***2FP***. We apply communication-efficient data parallelism for a special chunk of frequently-accessed embedding vectors stored on GPUs. While slight redundancy is introduced, the synchronization operation is simplified, and thus better utilizes the network bandwidth. We build a performance model on the replication strategy that makes a trade-off when selecting the replicated items. The sparse embedding vector access pattern is exploited to reduce the overall communication latency.

We evaluate *HypeReca* on two different clusters with up to 32 GPUs. With identical model quality, our system achieves $2.16-16.8\times$ speedup over the baseline systems on different

models and training datasets with terabytes of data.

2 Background and Motivation

2.1 DLRM Model and Work Flow

DLRMs are commonly used to predict behaviors of users, e.g. click-through rate (CTR). It takes IDs of entities, including the user, the presented content, and any possibly related items, as input. Then, using a deep neural network, it conducts predictions to provide better recommendations. For example, when predicting CTR of a video shown to a user, the user's ID, together with the video's ID, and possibly IDs of past ten videos the user watched, are fed into the DLRM as one training sample. The fact that the user clicked it is used as a label, and the model is trained by supervised learning.

As shown in Figure 2, a DLRM model mainly consists of two parts. *Sparse Part* contains one or multiple embedding tables. *Dense Part* consists of various neural network models.

Sparse Part contains several embedding tables. They map discrete IDs of certain categories to embedding vectors, commonly consisting of tens or some hundreds of floating point numbers. Due to the numerous contents online, sparse parts of real DLRMs contain more than billions [20, 21, 25] of parameters. Despite consuming much memory, it is sparsely accessed, involving little computation. Optimizing the memory-intensive operators [28] is effective on a single GPU, but scaling it up requires special communication design.

Dense Part studies on the embedding vectors of input items and makes predictions. This part is being intensively studied by the model developing community. MLPs and CNNs, Transformers and MMoEs [5, 11, 27, 36], are introduced for better model quality. As recommendation tasks require low inference latency, these models are usually of moderate size, but much more computation-intensive.

To train the model, a batch of labeled training data, containing up to millions of IDs, is first fed into *Sparse Part*. The data is processed by the DLRM, and output of *Dense Part* is compared with the ground truth. Then, gradients produced by a loss function are propagated backward through both parts. Finally, a gradient-based optimizer, e.g. SGD [16] or Adam [18], is used to update the parameters.

It should be noticed that the embedding tables are part of the parameters of the model. They are updated every iteration when training. Maintaining the optimizer states of the embedding tables is a part of the parameter updating process. Although the gradients are sparse, the updated embedding vector may be immediately needed by the next training batch.

2.2 Existing Basic Parallel Strategy

Distributed training is the most promising way to increase training throughput. However, in DLRM, the dense part involves intensive computation, while the sparse part requires a lot of memory to store the embedding tables. The difference in workload leads to different selections of parallel strategy. A basic hybrid parallel strategy is commonly used in present DLRM training systems [15, 39, 43]. Figure 3 illustrates the hybrid parallelism that applies different parallel strategy to sparse part and dense part separately.

For *Sparse Part*, it is inevitable that the embedding tables are partitioned across multiple devices, as they may contain terabytes of parameters. For simplicity, we adopt *GSET* [43] technique that maps IDs of items in different categories into a unique global ID, which simplifies multiple embedding tables into one. To fit *Sparse Part* into distributed memory, embedding vectors of different items are placed on different workers. From the view of distributed deep learning, it adopts model parallelism, where rows of embedding vectors are stacked up to be a tall matrix, and split up horizontally. Host memory is commonly used to store these large embedding tables [25, 43]. Some GPU-centric systems put them on GPUs [12, 15, 39], but they still have to offload most embedding vectors elsewhere, due to the huge size of training datasets.

Rest parts of the model, including data loading and *Dense Part*, adopt data parallelism. A global batch of training data is split into multiple mini-batches by samples. Each worker is bound to one GPU. *Dense Part* is replicated across all GPUs, as it has fewer parameters and is more compute-intensive. To synchronize the replicas, `all-reduce` operation is performed over the gradients before updating the models. This approach can greatly scale up the throughput [19, 22, 31], but it is inefficient in utilizing memory.

Due to the different parallel strategies between the two parts, the embedding vectors are re-organized. For instance, a training sample on worker A includes items whose embedding vectors are on worker B and C. Then, B and C need to find them in local memory, and send them to A. Globally, for a batch of training samples, every worker gathers embedding vectors from local memory, and send them to other workers. This is commonly implemented by invoking `all-to-all`, a basic sparse collective communication operation [30]. It may be replaced by `all-reduce` over a tensor containing multiple rows of zeros, depending on the efficiency of the communication operation implementation [39].

The all-to-all communication pattern leads to significant scalability issues. Training time composition of several open-source distributed DLRMs [15, 39] is measured and presented in Figure 4. *Sparse Part*, which is dominated by the sparse all-to-all communication, makes up most of the training time, exceeding 90% in most occasions. Increasing network bandwidth is the only direct way to accelerate the inter-device communication. For example, previous works utilize high-bandwidth NV-Links among GPUs within a node [39, 47], or equip high-performance NICs for every GPUs [3, 25]. These hardware items are expensive and less scalable compared to clusters connected via commodity hardware (e.g. PCIe GPUs and Infiniband HCAs per node).

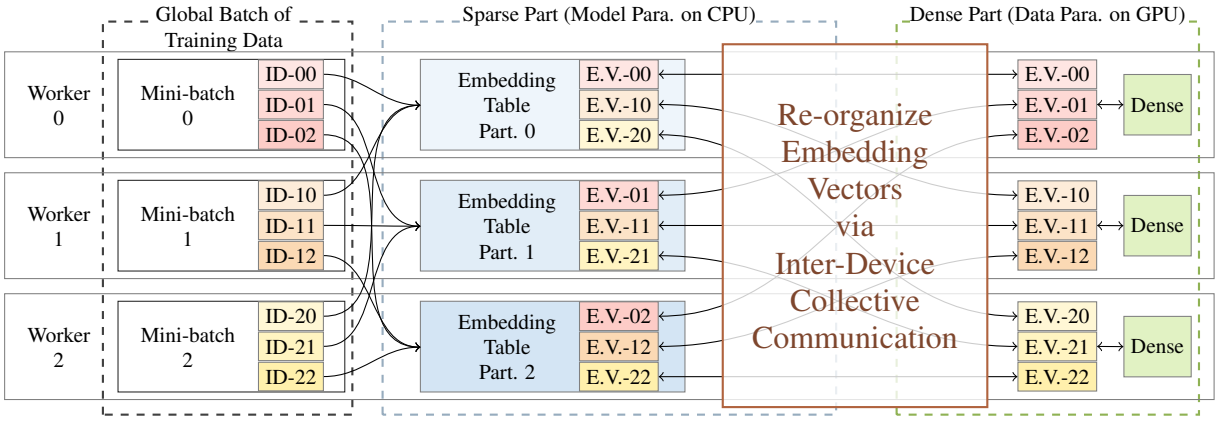


Figure 3: Existing Basic Hybrid Parallelism on Sparse Part and Dense Part

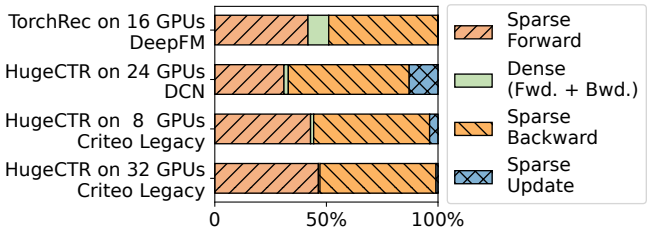


Figure 4: Iteration Time Break-down

In fact, the embedding vectors are created, read, and updated in the same way as a KV store system. Differently, the requests during DLRM training are served in large batches. The serving throughput is more important than the latency.

2.3 Throughput-Sensitive Data Management

While it is a widely known approach to handle hot and cold data separately, the presence of GPUs in DLRM training introduces unique challenges. Data management overhead of the training system is essential to overall performance. The system has to determine the items to replicate, indicate locations of all demanded items, and ensure synchronization of replicas within a short period of time.

When training a common DLRM, an NVIDIA V100 GPU consumes `159k` items in `10ms`. As there are commonly 8 GPUs in a node, the CPU has to provide the GPUs with embedding vectors of millions of items within a few milliseconds. The throughput of managing the embedding vectors should be as high as that of computation on GPUs, posing a strong efficiency demand.

The difficulty prevents prior works [2] from scaling up to multiple nodes. Expensive four-socket nodes [34] and even a dedicated CPU node equipped with Intel's most powerful Xeon Platium CPUs [3] are introduced to cope with the data management stress. As it is hard to change the hardware setup in many production cases, we focus on improving distributed DLRM training performance on the extensively deployed homogeneous clusters using commodity hardware.

2.4 Sparse Pattern of Training Data: Skewness

Intuitively, some items in DLRMs are much more frequently accessed than others, e.g., accounts of celebrities or spot news. Besides, embedding tables of some categories contain very few items, such as genders, which are always accessed. We refer to such uneven sparse access pattern as **skewness**.

It is easy to accelerate DLRM training by utilizing skewness across embedding tables. Applying different parallel strategies to different embedding tables [25, 33, 48] can reduce communication overhead with little modification of the training system. However, embedding tables are the smallest units of parallelism in these systems. Such granularity is so coarse that they miss many opportunities among items within embedding tables, and gain little improvement.

With a finer granularity, skew access pattern has been observed among embedding vectors within an embedding table [2,3,34]. We closely inspect such skewness in real datasets to find opportunities for improving the performance and scalability of distributed DLRM training.

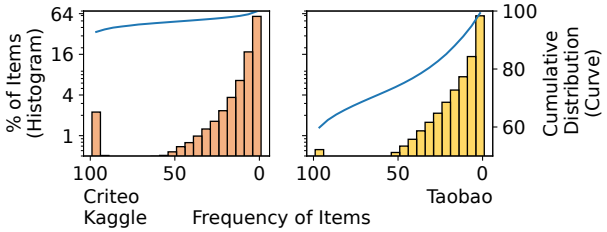


Figure 5: Observation on Skewed Distribution of Items

We inspect two public datasets, *Criteo Kaggle* [7] and *Taobao* [4]. We peek into 100 batches of training data in

each dataset. The histograms in Figure 5 aggregate the batch-frequency of items, i.e. the number of batches containing a certain item out of all 100 batches. In the *Criteo Kaggle* dataset, we find 2.2% items appear in more than 95 batches, indicated by the leftmost spiking bar of the histogram. We call them **frequent items** in our work. Similarly, we find that 0.6% items are frequent items in *Taobao* dataset. The rest items briefly follow an exponential distribution. Most of them appear in no more than 5 out of 100 batches.

We further inspect another batch to obtain the distribution of items over each bin, which indicates embedding vector access during training. Fewer than 1% items are not present in the previous 100 batches. As the cumulative curves in Figure 5 suggest, in *Criteo Kaggle* dataset, more than 90% accesses are directed to the 2.2% most frequently-accessed items. While in *Taobao*, 13% most frequent items can cover about 86% embedding vector accesses.

The skewness brings an opportunity for reducing communication. Suppose that we have a replica of the frequent items on each GPU, the `all-to-all` communication volume can be reduced by up to 90%. The end-to-end performance can be greatly improved since most time of distributed DLRM training is spent on the inter-device communication.

While the item-wise skewness pattern should be utilized, we keep in mind that the data management overhead should be minimized to achieve end-to-end speed up.

3 System Overview

HypeReca adopts the key-value store abstraction for the embedding vector database. As shown in Figure 6, the operations are integrated into the training process, while the stand-alone embedding database handles the requests.

Our system maintains the embedding vectors and the relevant optimizer states in chunks. The chunks may reside in host memory or device memory. As they occupy most of the storage space, *HypeReca* employs the following techniques to manage them. Decentralized Indexing Tables (**DIT**) are used to locate the embedding vectors in chunks on different processes, according to the input ID. It allows migrating the embedding vectors among chunks, and indexing the embedding vectors at high throughput. A special embedding chunk ($\mathcal{R}$ **chunk**) is placed in the device memory of GPUs to provide the dense part of the model with efficient direct device-local access to certain embedding vectors and utilize the skewness in the embedding vector access.

For a batch of IDs in the training dataset, the `Prefetch` operation first asynchronously locate or allocate the embedding vectors in the chunks. The `Pull` and `Push` operations work as the embedding computation in the forward and backward propagation. The `Update` operation commits the updates according to the gradients.

It takes negligible effort to use *HypeReca* with existing neural network training frameworks, such as PyTorch [29]

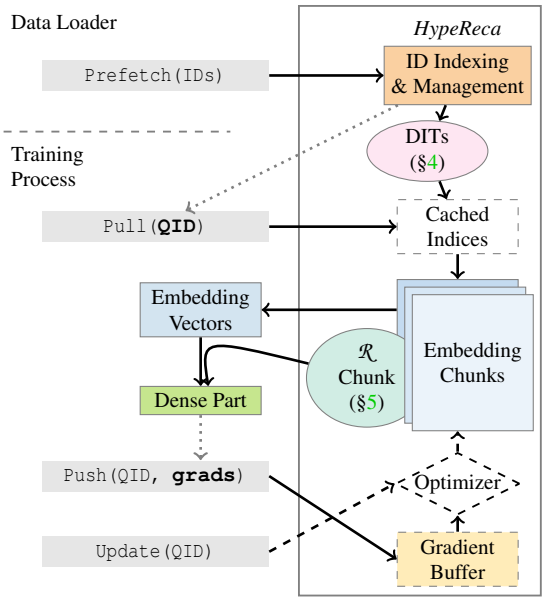


Figure 6: Overview of *HypeReca*

and HugeCTR [39]. While the `Prefetch` call may be inserted into the data loader or integrated with `Pull`, the operations can be simply used as a customized embedding layer.

4 Throughput-Oriented Data Management

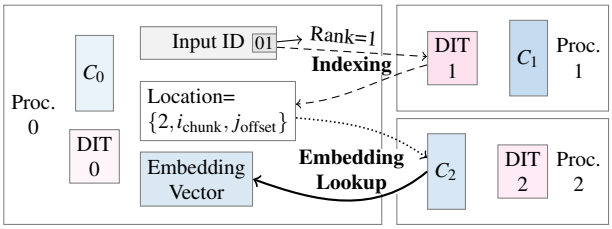
Performance challenges of the sparse communication for embedding vector accesses come from complicated fine-grained data management. On the one hand, millions of items are consumed by the GPUs in milliseconds. On the other hand, the embedding vectors locate in various places to utilize the memory. For a batch of training data, the system goes through the management routine for each item separately, which results in fragmented memory access and inter-process communication. We mitigate these drawbacks with throughput-oriented designs for competitive end-to-end training performance.

4.1 Dynamic Decentralized Indexing

Most embedding vectors are maintained in distributed memory space across multiple processes. In existing systems, the input IDs are commonly directly mapped to partitions of the embedding table by simple hashing, e.g. using a few bits in the ID to indicate the index of the partition and offset within it. However, besides hash collision that may hurt model quality [43], it is less scalable. To extend the capacity of a partition, a significant amount of data has to be moved around in memory, leading to unpredictable access latency.

To efficiently manage memory, *HypeReca* allocates spaces for embedding vectors by chunks. Every process maintain several embedding chunks in the host memory. Also, we

place a special replicated chunk on each GPU. For a newly seen ID, it is assigned to a certain chunk. If there is no space left in the last chunk, a new chunk is allocated. By this design, *HypeReca* is able to avoid any collision, and keep allocated memory utilized. Besides, there is little overhead to maintain the chunks, because they are orders of magnitudes fewer than the embedding vectors.



Example embedding vector fetching process for an item in a mini-batch on process 0 whose location is maintained on process 1, and the embedding vector resides in chunk C_2 on process 2.

Figure 7: Workflow to Fetch an Embedding Vector

As Figure 7 shows, to fetch an embedding vector, there are two steps: (1) ID of the item is converted to location information in the distributed memory space, namely **indexing**; (2) the actual **embedding lookup** is performed, where the embedding vector is fetched from an embedding chunk.

The location information consists of rank of the process that holds the embedding vector, index of the chunk, and offset of the embedding vector within the chunk. This information is compactly packed into a few bytes.

HypeReca employs decentralized indexing hash tables (**DIT**) to maintain the locations of items, which is more scalable compared with a centralized indexing module. To guarantee consistency, each ID is exactly maintained by one certain hash table on a process. The last few bits of IDs are used to indicate the rank of the process. On each process, the DIT only maintains location information, which is much lighter than the embedding vectors.

In the embedding lookup step, the location information is used to retrieve the embedding vector from memory of processes *i*. Also, during backward propagation, the gradients are accordingly sent backward, and the embedding vectors are updated by the end of every iteration.

A notable detail is that simply accessing remote embedding vectors through RDMA is poorly efficient. In an experimental version of our system, we adopt one-sided RDMA put and get operations provided by UCX to directly fetch the embedding vectors from remote host memory. Unfortunately, because the embedding vectors are barely larger than 500 bytes, the overhead to send one-sided requests dominates. The actual communication bandwidth can hardly achieve more than 10% the theoretical peak bandwidth.

Instead, in our system, for a batch of embedding vector fetching requests from process *A* to process *B*, process *B* locally gathers embedding vectors from local chunks (and maybe perform some local pooling according to the model) into a temporary contiguous memory space before sending them to process *A*. This approach better utilizes the bandwidth of the network, and the RDMA network is managed by the high-level communication libraries, such as MPI and NCCL.

The decentralized indexing design provides *HypeReca* with high flexibility. ID of the items can be in any format, without the need to be preprocessed into contiguous indices, which is very useful for streaming training data.

4.2 Asynchronous Parallel Indexing Pipeline

We find an extra performance benefit from the indexing design. Performing indexing is not related to the actual data of embedding vectors, which is updated in every iteration. Therefore, indexing can be decoupled with embedding lookup, and moved to the thread for data loading. After the data loader reads IDs in a training sample from storage, *HypeReca* immediately convert these IDs into locations in *HypeReca*. The training thread then asynchronously reads them from a buffer, so the latency on the critical path is reduced.

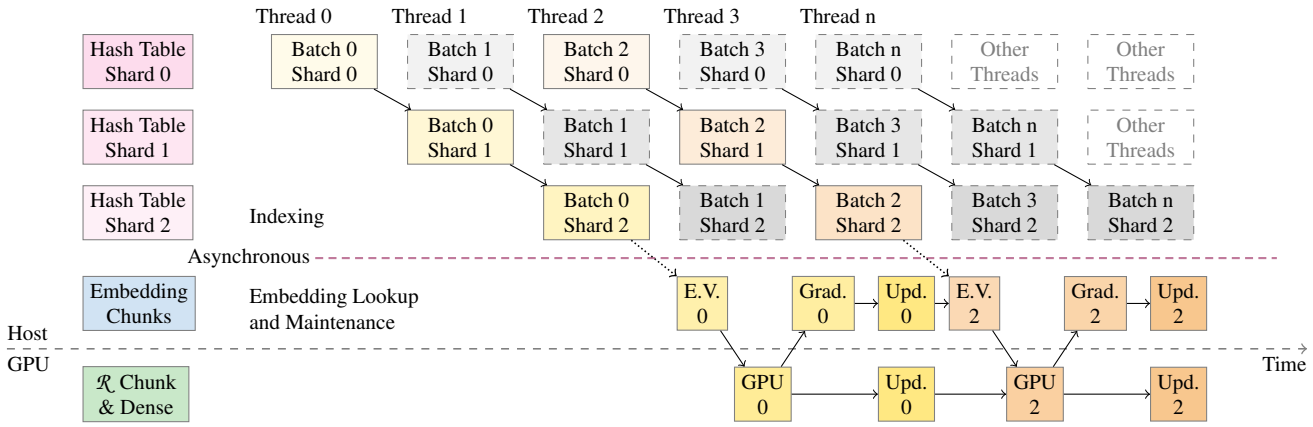
Specifically, inter-process communication is conducted to perform indexing over remote DITs. When a process performs indexing for a batch of input IDs, it sends requests to all other processes. For each process, one thread handles one request from a remote process. The thread goes through the local DIT to find locations for each ID in the batch. New locations are assigned to embedding vectors of newly-seen IDs. Finally, after the allocation is finished, the thread responds to the request with a batch of locations.

The latency of the request-handling thread can be high, because even executed in batches, the latency to access hash tables is not that predictable. Fortunately, multiple threads are used to load multiple batches of data simultaneously ahead of training in most training systems, so indexing can also be performed in parallel to increase throughput.

To guarantee consistency of the DITs, read-write locks are used to support concurrent access. We observe that the lock is a performance bottleneck. Besides the overhead in its implementation, many other threads are blocked when one thread is trying to write new items. A naïve implementation can barely meet the throughput requirement.

We invent a parallel pipeline for the hash table operations, exploiting opportunities brought by the batch of queries. For better concurrency, we first slice the hash table into a few tens of shards. Multiple threads handling different indexing tasks go through the shards as a pipeline, as shown in Figure 8. Every thread only needs to lock a shard once, and all the IDs in this shard are processed. Such design eliminates most contention, because the threads are mostly operating on different shards in a pipeline manner.

This indexing pipeline is inspired by the commonly used



Batch 1 and 3 are trained on some remote processes. Thread 1 and 3 respond to the indexing request, and their further steps are not presented in this figure.

Figure 8: Indexing Pipeline and Computation in a Process

pipeline parallelism in training large neural networks. However, different from the inter-layer and inter-device pipeline parallelism, this pipeline works within the embedding layer and within a CPU. It utilizes independent data structures simultaneously for higher throughput.

Although this pipeline does not directly reduce indexing latency of one batch, the overall throughput of multiple threads is increased. As long as it can match the throughput on GPUs, the overall training process is not throttled by indexing, because it is processed by the data loading thread. In our practice, throughput of this indexing pipeline is high enough to match the throughput of 8 GPUs using 32 CPU threads.

5 Pattern-Aware Parallel Strategy

The chunk-based in-memory embedding database provides us with the flexibility to utilize skew access pattern by placing chunks on the GPU. The synchronization overhead of the chunks varies according to the placement strategy of the chunks and the related synchronization algorithm. Inspired by hybrid training algorithms of typical neural networks, we propose a hybrid chunk placement strategy, and build a performance model to guide the selection of the strategy. Over the ever-existing all-to-all communication for the sparse embedding vector access, we employ scheduling techniques to reduce its contention.

5.1 2-Fold Parallel Strategy

We partition the embedding vectors into two parts according to the access frequency. An item-wise two-fold parallel strategy (*2FP*) is applied to both parts of the embedding vectors. More specifically, we apply either of the following two strategies to the embedding vector related to an item: (a) data parallelism is used on a small portion of frequent items; (b) model parallelism handles the rest items.

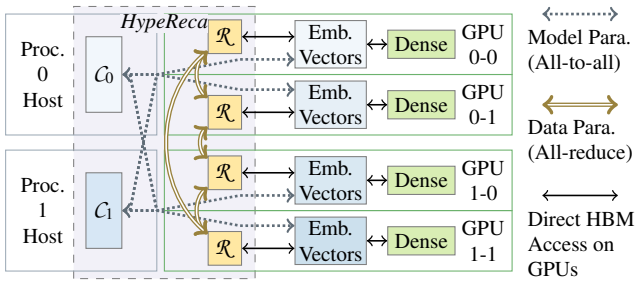


Figure 9: The Two-Fold Parallel Strategy (*2FP*)

Figure 9 illustrates the design of *2FP*. There are two exclusive types of partitions. Any embedding vector belongs to exactly one of them.

$\mathcal{R}$ is the replicated embedding chunk for high throughput GPU-local access of embedding vectors. There is a replica of $\mathcal{R}$ on each GPU, so accesses of embedding vectors in $\mathcal{R}$ are performed locally through the high-bandwidth device memory (HBM). Inter-device data movement is eliminated during forward and backward propagation, removing significant communication overhead. Synchronous data-parallel training algorithm is used over $\mathcal{R}$ to keep it consistent across GPUs. Gradients are synchronized in each iteration before updating the embedding vectors.

$\mathcal{C}_i (i \in [0, N_{\text{proc.}}))$ denotes the rest trivial chunks on process i. Host memory of the process is used to store embedding vectors in $\mathcal{C}_i$. This saves GPU memory to store $\mathcal{R}$ and the dense part. During training, processes fetch embedding vectors of samples in its local batch from all $\mathcal{C}_i$, excluding those in $\mathcal{R}$. Gradients are then pushed back, aggregated and used to update the embedding vectors. As the updates finish before the

pulling operation of the next iteration, the dense part always works on the latest version of embedding vectors. In other words, model parallelism is applied to $\mathcal{C}_i$. The embedding tables are row-wise partitioned.

HypeReca achieves model transparency via the two-fold strategy. The embedding vectors are kept in a fixed place throughout the training process. Without complicated strategies like prefetching or caching, we are able to guarantee strong consistency directly by parallel training algorithms.

Additionally, the *2FP* design removes the necessity to perform deduplication [46], which removes duplicated items before initiating all-to-all to reduce its volume, but introduces extra overhead. In *HypeReca*, the duplicated items are mostly frequent items in $\mathcal{R}$, which do not go through all-to-all.

In this section, we illustrate how to reduce the communication overhead with *2FP* by building a performance model. Speed up of *2FP* is achieved based on the efficient managing the data in end-to-end DLRM training processes.

5.2 Performance Modeling

To select proper items to put in $\mathcal{R}$ for optimal performance, there are two steps. First, we identify the frequency of items. Then, we pick a proper number of items.

For the first step, peeping at the dataset can provide adequate information to find out the most frequent items. As Figure 5 suggests, sampling of 100 training batches has already shown an obvious tendency of skewness. And 100 iterations are likely fewer than 1% of a training process that can be done in a few minutes. Due to the fact that more than $10k$ iterations are normally needed for training, and the data follows a similar skewed distribution, performing frequency counting of items in the first 100 iterations introduces only minor overhead. It is possible that the skewness changes during a training process that lasts much longer. We can always re-peep to keep $\mathcal{R}$ fresh, and this can be overlapped with other jobs that pause training, e.g. writing a checkpoint.

The second step is more challenging. The size of $\mathcal{R}$, denoted as $\mathbb{R}$, is significant to the performance. We find a trade-off between reducing the all-to-all volume and increasing the synchronization overhead of $\mathcal{R}$. Having more items in $\mathcal{R}$ can reduce the number of items that must be fetched across processes, i.e., all-to-all overhead. However, the whole $\mathcal{R}$ must be synchronized and updated as a dense parameter in every iteration, even if some items are not accessed at all, incurring overhead and losing the benefits of sparsity. Therefore, we need a proper $\mathbb{R}$ to minimize overall latency.

A performance model is built to help to determine $\mathbb{R}$. The overall communication latency L_{overall} consists of two parts:

$$L_{\text{overall}} = L_{\mathcal{R}} + L_{\mathcal{C}}$$

The latency to synchronize $\mathcal{R}$ is noted as $L_{\mathcal{R}}$ and the latency to fetch data in $\mathcal{C}_i$ is noted as $L_{\mathcal{C}}$. As both tasks are communication-intensive, there is little benefit from overlapping them, so we can add them up directly.

The first part, $L_{\mathcal{R}}$, is proportional to $\mathbb{R}$:

$$L_{\mathcal{R}} = \mathbb{R} C_{\text{ar}}$$

The latency comes with a coefficient C_{ar} that denotes the communication performance of all-reduce on given hardware and model setup, which can be calculated or measured by a micro-benchmark.

The second part, $L_{\mathcal{C}}$ is a little more complicated:

$$L_{\mathcal{C}} = N(1-\rho(\mathbb{R}))C_{\text{a2a}}$$

The latency of $\mathcal{C}_i$ is related to N, the number of items in a batch. Here, $1-\rho(\mathbb{R})$ denotes the portion of items that do not present in $\mathcal{R}$, and have to be fetched from $\mathcal{C}_i$. As communication for $\mathcal{C}_i$ is all-to-all, $L_{\mathcal{C}}$ is calculated with another coefficient, C_{a2a}, which can be obtained in a similar way to C_{ar} above.

Algorithm 1 Produce ρ from sampled training data

```
1:  function GetK(A sampled list of items D[N])
2:      C ← {0,0,...,0}   ▷ A map from item ID to counter
3:      for all Item ID i ∈ D do
4:          C_i ← C_i + 1
5:      end for
6:      U ← Unique IDs in C
7:      Descending sort U by C_{U_i}
8:      ρ_0 ← 0
9:      for i ← 1...|U| do
10:         ρ_i ← ρ_{i-1} + C_{U_i}/N
11:     end for
12:     return ρ
13: end function
```

$\rho(\mathbb{R})$ is a function that uses the sampled training data to estimate the portion of embedding vector accesses that are covered by $\mathcal{R}$. Algorithm 1 is used to determine $\rho(\mathbb{R})$ for any possible $\mathbb{R}$. Items are sorted according to their frequency, and the top $\mathbb{R}$ items are selected. Then, we calculate the expectations of items in $\mathcal{R}$ to appear in a training batch.

Finally, L_{overall} is expanded as follows:

$$L_{\text{overall}} = \mathbb{R} C_{\text{ar}} + N(1-\rho(\mathbb{R}))C_{\text{a2a}}$$

We observe that $L_{\mathcal{R}}$, the first part of L_{overall}, increases when $\mathbb{R}$ increases, while $L_{\mathcal{C}}$, the second part, decreases, corresponding to the aforementioned trade-off. Besides, $\rho(\mathbb{R})$ grows sharply with $\mathbb{R}$ when $\mathbb{R}$ is relatively small, e.g. a few thousands, and turns to a slow increase when $\mathbb{R}$ is larger. Therefore, L_{overall} is roughly a U-shaped curve over $\mathbb{R}$.

A ternary search algorithm can be used to quickly find an optimal range of $\mathbb{R}$, considering the memory limit as the maximum candidate value. Then, we employ bulk enumeration to get an optimal value of $\mathbb{R}$.

By further analyzing the equation, we can see that various batch sizes N lead to different optimal $\mathbb{R}$ values for minimum L_{overall}, as it only affects L_C. As N gets larger, $(1-\rho(\mathbb{R}))C_{\text{a2a}}$ becomes more significant in L_{overall}, and a larger $\mathbb{R}$ is more beneficial. This brings extra performance gain for *HypeReca* when scaling DLRM training up with larger batch size, if there is enough memory for a larger $\mathcal{R}$.

5.3 Analytical Effectiveness of *2FP*

Should the frequent items be removed from model parallelism, the communication volume of all-to-all can be significantly reduced. And as the they only make up a small portion of all items, they can be duplicated with little extra memory consumption, and synchronized with less communication overhead. This can be verified by applying the performance model to the real-world datasets that we observed.

First, we measure the performance of all-reduce for data parallelism and all-to-all for model parallelism on 32 GPUs across 4 nodes. Then, we calculate ρ from the datasets to estimate the reduction of communication.

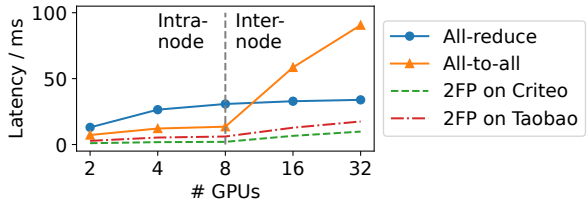


Figure 10: Communication Latency Modeling

For model parallelism using all-to-all, assume that there are `32MB` of embedding vectors to exchange on each GPU. Then, assume that the actual number of different items is about 6% the total number of samples. In this case, all-reduce is performed over `64MB` of embedding vectors on each GPU using data parallelism.

The solid lines in Figure 10 show the latency of both collective communication operations. The performance of all-to-all is better within a node, but its latency grows much larger than all-reduce when scaling up to more than 16 GPUs across multiple nodes, because it is unfriendly to the hierarchical connection topology. In contrast, all-reduce can scale up well across nodes, as the distributed reduction algorithms fit the hardware architecture and network topology better.

The dashed lines in Figure 10 show our estimation of communication latency of *2FP* on both datasets. When training over Kaggle dataset, we have $\rho(32,265) = 0.92$. This means about `32k` (2.2%) items are processed by all-reduce, and the all-to-all latency is reduced by 92%. Theoretically, the communication latency is reduced by 9.2×, compared to the original strategy that only applies model parallelism for the embedding vectors and performs all-to-all over all of them.

The case for Taobao is harder, but the opportunity is still significant. We use $\rho(227,058) = 0.86$, so having `227k` (13%) frequent items replicated removes 86% all-to-all overhead. As a result, the communication latency is reduced by 5.2×.

Even compared with data parallelism, there is $93\% - 245\%$ communication speed up. In fact, data parallelism is not practical because the actual memory footprint would be far more than that of all-to-all on the same number of items, because it duplicates instead of splitting data.

5.4 Contention-free Scheduling

Exchanging the embedding vector among processes is more complicated than a simple all-to-all collective communication operation. When the workers exchange the embedding vectors, they are first gathered into a temporary contiguous tensor from the chunks to utilize the network bandwidth. Similarly, during indexing, each worker iterates through its local DIT to respond to each indexing request.

We summarize the all-to-all request workload pattern as follow. Each process sends requests to all processes. To respond to the request, the remote process has to conduct computation locally before sending the results back. Basically, we employ a separate thread pool on each process to respond to the requests. Contention happens when multiple requests are sent to the same process simultaneously.

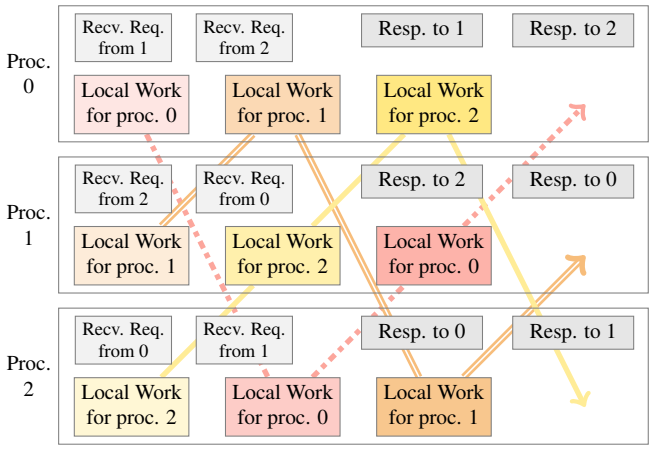


Figure 11: A 3-process Example of the Ring Schedule

HypeReca adopts a contention-free ring schedule to avoid the problem. Figure 11 instantiates the schedule with 3 processes. Formally, for a workload among n processes, n steps are performed. In step i, process r performs computation of request from process $((r+i) \mod n)$, and communication operations are executed accordingly.

For peak performance, the communication operations are all performed asynchronously to better overlap with computation. Computation for the request to the local process is

placed at the beginning of the schedule to minimize initialization overhead. Finally, at any point of time, each process has exactly one request to handle, so the demanded number of CPU cores remains stable through the training process, and the overall throughput is promising.

6 Evaluation

6.1 Setup and Methodology

We use two different clusters to evaluate *HypeReca*.

- *Antique*: Each node contains 8× `NVIDIA A100 SXM4` GPUs with `40GB` memory and dual `Infiniband HDR 200Gb/s` HCA cards. Due to limited quota, we are only able to compare scalability of *HypeReca* and HugeCTR over Terabytes dataset on this cluster.
- *Vintage*: Each node contains 8× `NVIDIA V100 PCIe` GPUs with `16GB` memory and an `Infiniband EDR 100Gb/s` HCA card. All experiments are conducted on this cluster unless otherwise stated.

Antique represents a high-end cluster setup for training large deep learning models, whereas *Vintage* has a more budget-friendly setting. The PCIe GPUs have similar communication bandwidth as consumer-grade GPUs, such as GTX 1080. As the DLRM training workload is communication-bound, we expect our system to achieve higher speedup on clusters with lower communication bandwidth, enabling DLRM training on these more economical hardware platforms.

We compare the performance of *HypeReca* with the following three open-source baselines.

TFDE [37] The name is short for TensorFlow Distributed Embeddings. It implements Parallax [17] for DLRMs using newer TensorFlow [1]. Compared with a bare data-parallel approach, this system maintains the sparse part by a distributed sparse parameter server. In fact, this is equivalent to the existing basic hybrid parallelism that applies model parallelism for the entire sparse part.

TorchRec [15] A coarse-grained hybrid parallel strategy [25, 33] is applied to utilize the skewness across embedding tables. It is easy to implement, yet missing opportunities within an embedding table. It also adopts CUDA unified memory technique for faster data transfer between host memory and GPU memory.

HugeCTR [39] This system is used by NVIDIA in the MLPerf DLRM benchmark, with a Caffe-like front end that supports a limited range of models. All the embedding tables are placed on GPUs, and it utilizes GPU-direct communication techniques to accelerate all-to-all.

We set up test cases using three public real-world datasets for three popular DLRMs, as shown in Table 1.

Taobao advertisement display and click dataset [4] is a public dataset of logs from *taobao.com*, one of the largest e-commerce websites in China. Sparse features of the dataset include IDs of products, customers, and campaigns. Criteo-Kaggle [7] dataset is a benchmark dataset for CTR prediction, with more masked sparse features per sample. Terabytes [6] is of similar data format, but much larger, used in MLPerf DLRM benchmark [41].

Table 1: Datasets and Models for Evaluation

Dataset Model	Taobao [4] DCN [27]	Criteo Kaggle [7] Legacy [39]	Terabytes [6] DLRM [41]
# Feat. (Den./Sp.)	4/4	1/39	14/26
# Samples	26M	36M	4.3B
Dataset Size	`1.17GB`	`4.3GB`	`671GB`
# Items	1.83M	1.60M	187M
Embedding Size	`117MB`	`411MB`	`96.1GB`

Deep and Cross Network (DCN) [27] uses a cross network layer, i.e. dot product of embedding vectors, to learn interactions between sparse features. The legacy model of HugeCTR [39] treats all features as sparse, and process them with a simple multi-layer perceptron (MLP) network. DLRM benchmark of MLPerf [41] uses an interaction layer with larger MLP, being a widely-used standard benchmark to evaluate DLRM training systems.

We first compare *HypeReca* with the baselines in end-to-end training. Each model is trained for 500 iterations, and the average trained samples per second is used as the metric of throughput. Then, we validate the performance model by testing different selection of $\mathbb{R}$. Also, we compare the indexing performance of *HypeReca* with a baseline implementation.

6.2 Scalability and End-to-end Results

We first measure both strong and weak scaling of *HypeReca*, i.e., keeping either global batch size or batch size per GPU the same, and use HugeCTR as a reference. We choose $\mathbb{R}$ of *HypeReca* for each setting according to the performance model, which is detailed in §6.5.

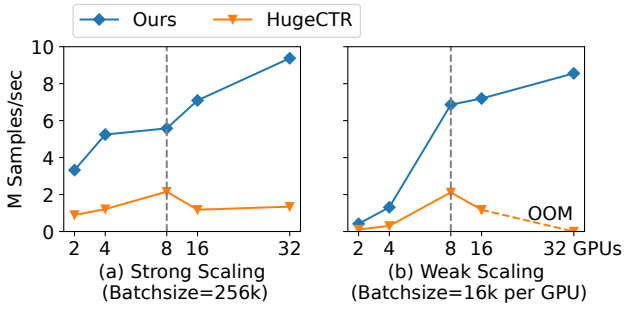


Figure 12: Intra- and Inter-Node Scalability on *Vintage*

The scalability on Criteo dataset is presented in Figure 12. Two systems show a similar tendency to scale up within a

node. However, when scaling to more than one node, *HypeReca* still speeds up as the number of GPUs increases, while HugeCTR becomes even slower. Notably, HugeCTR runs out of memory in the weak scaling experiment as the global batch size is too large, and it fails to utilize host memory. Meanwhile, *HypeReca* keeps most embedding vectors on the host side, using much less GPU memory, making it possible for larger batch sizes or more complicated dense parts.

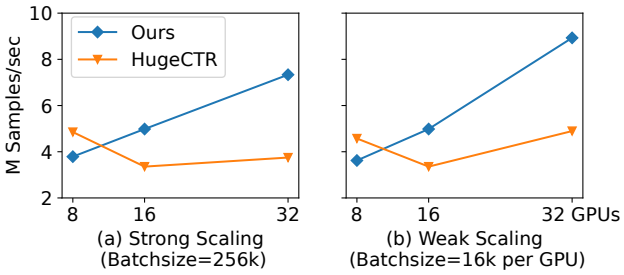


Figure 13: Inter-Node Scalability on *Antique* Cluster

Inter-node scalability on *Antique* over Terabytes dataset is shown in Figure 13. *HypeReca* is slower than HugeCTR on one node. HugeCTR benefits from the fast intra-node NVLink connection, while *HypeReca* has to copy embedding vectors in $\mathcal{C}_i$ from host memory via much slower PCIe connections. However, *HypeReca* outperforms HugeCTR on multiple nodes. Because the bandwidth of inter-node connection is lower (from `600 GB/s` per GPU to `400 Gbps` per node), HugeCTR is even slower on multiple nodes than on a single node. In contrast, *HypeReca* shows a better tendency to scale up to more nodes, even on a cluster with faster connections.

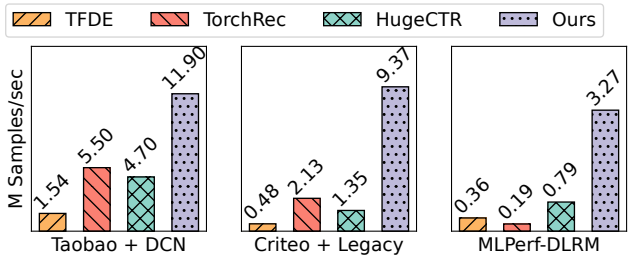


Figure 14: End-to-end Comparison on *Vintage* Cluster

Training throughput on 32 GPUs over different models and datasets are shown in Figure 14. TFDE achieves poor performance because of the high re-organization overhead. TorchRec outperforms HugeCTR in the first and second case, as it finds opportunities on certain embedding tables, while data transfer overhead between CPU and GPU slows it down on Terabytes dataset. *HypeReca* outperforms all baselines on all test cases, achieving 9.1×, 16.8×, and 4.2× speedup on the MLPerf benchmark, respectively.

Unfortunately, we are not able to compare *HypeReca* with other systems utilizing fine-grained skewness [2, 3, 34] due to issues on code availability or deployment feasibility. Meanwhile, they achieve less speed up over the same TorchRec baseline in test cases using similar models, datasets, and number of GPUs. As *HypeReca* outperforms TorchRec and HugeCTR on 32 GPUs, we believe that *HypeReca* achieves competitive performance and scalability.

6.3 Effectiveness of the Indexing Pipeline

We develop a micro-benchmark for the decentralized dynamic indexing module using different numbers of data-loading threads per process on each of 4 nodes. A node has 32 cores and 64 threads. As indexing is integrated into the data loader, the throughput of the data loader shall match the training throughput. We implement a baseline that uses hash tables without any schedule for the batches of queries. Another reference implementation is using a simple hash strategy for indexing without DITs, which involves the collision issue that degrades the model quality.

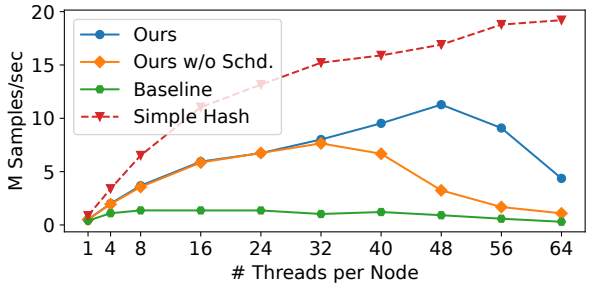


Figure 15: Multi-threaded Indexing Throughput

Figure 15 shows results of the micro-benchmark over Terabytes dataset. The baseline achieves a maximum throughput of 1.37M samples per second using 8 threads per node, much lower than the training throughput of *HypeReca*, 3.3M/s. Its throughput even decreases when using more threads, due to the high locking overhead of the DITs. In contrast, the indexing pipeline of *HypeReca* achieves more than 10M samples per second, 8.26× faster than the baseline. This matches the throughput of training the dense part on GPUs.

Because no communication is involved, the simple hash baseline is always faster than our system. Still, the optimally configured DITs achieve 50% the peak performance of such a performance-maximized implementation.

The *HypeReca w/o Schd.* curve shows the performance of *HypeReca* when we disable the contention-avoiding schedule of indexing. It is able to keep similar throughput with *HypeReca* using fewer than 32 threads. Because the CPU resource is relatively abundant, the requests can be handled quickly.

However, when having more threads that generate more requests simultaneously, contention severely slows down the indexing tasks, hence the whole system. Our schedule effectively reduces the contention, and is even able to improve the throughput when the CPU is oversubscribed. As there are other active threads, *HypeReca* achieves the peak performance when oversubscribing 50% CPU cores, which brings 40.8% extra throughput over the case using 32 cores.

6.4 Overhead for Changing Skewness

Regarding the change of skewness pattern during the training process, we first observe the Terabytes training data in an entire epoch. The data includes real online user access history across 24 days. We first get the $\mathcal{R}$ of different sizes according to the first 100 batches of the first day. Then, we inspect the coverage of access in the following iterations of the first day, and in some sampled batches of the next 23 days.

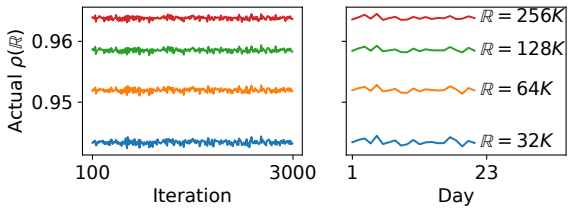


Figure 16: Variance of Skewness During Training

As shown in Figure 16, all of the actual $\mathcal{R}$ coverage rates are consistent, both within a day and across multiple days. Besides, shuffling of the dataset is a common practice for training on both offline and online training data. Therefore, this result is generalizable to other scenarios.

Second, we evaluate the overhead to update the $\mathcal{R}$ by writing all the embedding vectors back to the $\mathcal{C}_i$, and then pull new frequent embedding vectors into a new $\mathcal{R}$. The result of the micro-benchmark is shown in Figure 17.

Figure 17: Comparing the Latency of Re-sharding

As a reference, we include the latency of 10 training iterations of the MLPerf-DLRM model with different batch sizes. The overhead of such re-sharding operation increases with the size of $\mathcal{R}$. Meanwhile, even with the largest $\mathbb{R}$, it only takes about 1 second to complete the re-sharding tasks. The latency is at the same level of training for 10 iterations. Considering the above observations that the skewness pattern will likely remain stable, we can conclude that the overhead to update the $\mathcal{R}$ with the changing skewness pattern is minor in the whole training process.

6.5 Validating the Performance Model

We calculate $\rho(\mathbb{R})$ for all the datasets, as presented in Figure 18. When $\mathbb{R} = 64k$, $\rho(\mathbb{R})$ is 73.9% for Taobao, and around 96% for both Criteo and Terabytes. This contributes to the reduced communication volume of *HypeReca*, and explains the reason for *HypeReca* to perform better on the latter two.

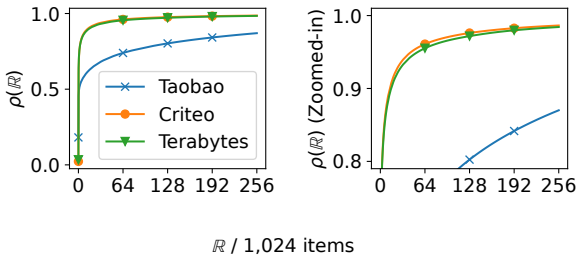


Figure 18: $\rho(\mathbb{R})$ for Used Datasets

The red dashed curve in Figure 19 indicates the latency predicted by our performance model using the above $\rho(\mathbb{R})$ in three setups on 32 GPUs. Except for a few outliers caused by performance disturbance, it matches the actual latency well. The average coefficient of determination, r^2, exceeds 0.9 when fitting parameters, C_{ar} and C_{a2a}, which validates the accuracy of our performance model.

The stacked areas of Figure 19 represent a break-down analysis of the latency over different $\mathbb{R}$. Latency of the dense part remains unchanged, because it only involves unchanged dense computation on GPUs. Overhead of data parallelism for $\mathcal{R}$ grows linearly with its size. The `all-to-all` overhead decreases in an L-shaped curve, in accordance with $(1 - \rho(\mathbb{R}))$. For Terabytes dataset, its latency reduces drastically when $\mathbb{R}$ increases from 0 to 1024, and then gradually gets smaller when increasing $\mathbb{R}$. As small $\mathcal{R}$ covers less items in Taobao dataset, the all-to-all latency decreases slower.

In all 3 different setups, the overall latency is a U-shaped curve over $\mathbb{R}$. An optimal $\mathbb{R}$ brings up to 7.80× speedup over Terabytes dataset, and 1.60× over Taobao, comparing with $\mathbb{R} = 0$, i.e., disabling *2FP*. This matches the speed-up over the baseline systems in Figure 14, and suggests that simply using host memory to store embedding vectors without *2FP* brings little speedup.

For Terabytes dataset, we conduct the ablation experiment over two different batch sizes. The optimal $\mathbb{R}$ gets larger as the batch size grows, which matches the observation from the equation of our performance model in §5.2.

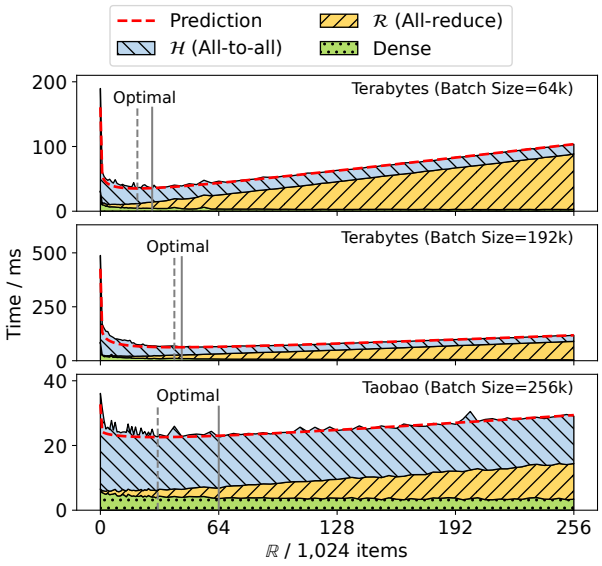


Figure 19: Iteration Time Prediction and Actual Breakdown

Meanwhile, the end-to-end latency varies little around the optimal $\mathbb{R}$ value. Estimating a brief range of $\mathbb{R}$ is enough to get near-optimal performance of *HypeReca*. In our experiments, the deviation of $\mathbb{R}$ estimations from real optimal values incurs no more than 1% overhead on actual latency.

As the optimal $\mathbb{R}$ ranges from 20k to 64k, the GPU memory of $\mathcal{R}$ ranges from `10MB` to `32MB` for the embedding vectors of 128 floating point numbers. Therefore the overhead to store and synchronize $\mathcal{R}$ is minor for GPUs.

7 Related Works and Discussion

General auto-parallel NN training systems. Recent works [26, 31, 32, 49] focus on partitioning typical dense models with a computation graph as input. These approaches fail to explore sparsity within layers, e.g. skewed data access pattern of embedding tables.

4-D parallelism for DLRM. A prior work [25] concludes all possible parallelism for DLRM into 4 types: column-wise and row-wise, table-wise and data parallelism. *2FP* generally contains the latter three ways of parallelism in a more flexible and high-performance way. While embedding vectors are mostly too short to be applied with column-wise parallelism in practice, *HypeReca* is also compatible with it.

Using GPU memory as cache. GPU memory is treated as a cache of CPU memory in many other works [9, 24, 42], and they achieve speedup on single GPUs. However, they select frequent items separately on each worker, and do not change the `all-to-all` communication pattern. Therefore, they do not scale up to multiple GPUs better than HugeCTR [39], which puts all data on GPUs, Furthermore, our work is orthogonal to these works, and can directly apply their techniques on $\mathcal{C}_i$ in *HypeReca*.

Distributed replicated cache [3] is partly equivalent to *2FP*, but online addition and eviction make its control flow too complicated to achieve competitive scalability.

Hardware-dependent systems. Introducing specific hardware can alleviate the communication bottleneck, e.g. utilizing fast NVLinks within a node [48] or pairing every GPU [25, 34] with high-performance NICs. Compared with these systems that require expensive hardware, *HypeReca* is more general and applicable to various setups with more affordable commodity hardware.

Changing the training algorithm. Embedding vector decomposition [35, 38, 40] and compression [10] reduce the memory footprint of the sparse part. Asynchronous training [21, 24, 43] and dropping some unimportant embedding vectors [20] are proved effective in certain cases. Model developers have to be aware of the deviation of algorithms which may affect model quality. *HypeReca* does not bring such concern, while being compatible with them.

DLRM as a graph. The training samples of DLRMs may be treated as edges in a hypergraph. Using graph-based methods to train the model may improve its efficiency [45]. However, it is hard for current graph-based training systems [13, 14] to meet the throughput demand of DLRMs and keep model-transparency at the same time.

Typical KV store systems. Cold/hot partitioning is employed by typical key-value store systems [8]. Meanwhile, applying such technique to the DLRM training workload demands extra dedicated design in synchronization policy and throughput-centric data management mechanism.

8 Conclusion

Scalability of DLRM training is important, while sparse communication of re-organizing data between its two parts introduces huge communication cost. We design an in-memory distributed embedding database, *HypeReca*, to reduce the cost. The throughput-centric data management module allows us to utilize GPU memory with the sparse access pattern of the workload. The two-fold parallel strategy effectively boosts the overall performance of the distributed DLRM training.

Acknowledgment

We would like to thank the anonymous reviewers and our shepherd Roberto Palmieri for their insightful comments. This work is supported by Beijing Natural Science Foundation (L242017), NSFC for Distinguished Young Scholar (62225206), National Natural Science Foundation of China (U23A6007), and Tsinghua University Initiative Scientific Research Program. Jidong Zhai is the corresponding author of this paper.

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: a system for large-scale machine learning. In *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, pages 265–283, 2016.

[2] Muhammad Adnan, Yassaman Ebrahimzadeh Maboud, Divya Mahajan, and Prashant J. Nair. Accelerating recommendation system training by leveraging popular choices. *Proc. VLDB Endow.*, 15(1):127–140, jan 2022.

[3] Saurabh Agarwal, Chengpo Yan, Ziyi Zhang, and Shivaram Venkataraman. Bagpipe: Accelerating deep recommendation model training. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 348–363, 2023.

[4] Alimama. Ad display/click data on taobao.com. https://www.kaggle.com/datasets/pavansanagapati/ad-displayclick-data-on-taobaocom, 2020.

[5] Qiwei Chen, Huan Zhao, Wei Li, Pipei Huang, and Wenwu Ou. Behavior sequence transformer for e-commerce recommendation in alibaba. In *Proceedings of the 1st International Workshop on Deep Learning Practice for High-Dimensional Sparse Data*, pages 1–4, 2019.

[6] Criteo. Criteo 1tb click logs dataset. https://ailab.criteo.com/download-criteo-1tb-click-logs-dataset, 2014.

[7] Criteo. Criteo display advertising challenge. http://www.kaggle.com/c/criteo-display-ad-challenge, 2014.

[8] dePaul Miller, Jacob Nelson, Ahmed Hassan, and Roberto Palmieri. Kvcg: a heterogeneous key-value store for skewed workloads. In Bruno Wassermann, Michal Malka, Vijay Chidambaram, and Danny Raz, editors, *SYSTOR '21: The 14th ACM International Systems and Storage Conference, Haifa, Israel, June 14-16, 2021*, pages 5:1–5:12. ACM, 2021.

[9] Jiarui Fang, Geng Zhang, Jiatong Han, Shenggui Li, Zhengda Bian, Yongbin Li, Jin Liu, and Yang You. A frequency-aware software cache for large recommendation system embeddings. *arXiv preprint arXiv:2208.05321*, 2022.

[10] Hao Feng, Boyuan Zhang, Fanjiang Ye, Min Si, Ching-Hsiang Chu, Jiannan Tian, Chunxing Yin, Summer Deng, Yuchen Hao, Pavan Balaji, Tong Geng, and Dingwen Tao. Accelerating communication in deep learning recommendation model training with dual-level adaptive lossy compression. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, SC '24. IEEE Press, 2024.

[11] Huifeng Guo, Ruiming Tang, Yunming Ye, Zhenguo Li, and Xiuqiang He. Deepfm: A factorization-machine based neural network for CTR prediction. In Carles Sierra, editor, *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017*, pages 1725–1731. ijcai.org, 2017.

[12] Jiaao He, Shengqi Chen, and Jidong Zhai. POSTER: pattern-aware sparse communication for scalable recommendation model training. In *Proceedings of the 29th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming, PPoPP 2024, Edinburgh, United Kingdom, March 2-6, 2024*, pages 466–468. ACM, 2024.

[13] Kezhao Huang, Haitian Jiang, Minjie Wang, Guangxuan Xiao, David Wipf, Xiang Song, Quan Gan, Zengfeng Huang, Jidong Zhai, and Zheng Zhang. Freshgnn: Reducing memory access via stable historical embeddings for graph neural network training. *Proc. VLDB Endow.*, 17(6):1473–1486, 2024.

[14] Kezhao Huang, Jidong Zhai, Zhen Zheng, Youngmin Yi, and Xipeng Shen. Understanding and bridging the gaps in current gnn performance optimizations. In Jaejin Lee and Erez Petrank, editors, *PPoPP '21: 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Virtual Event, Republic of Korea, February 27- March 3, 2021*, pages 119–132. ACM, 2021.

[15] Dmytro Ivchenko, Dennis Van Der Staay, Colin Taylor, Xing Liu, Will Feng, Rahul Kindi, Anirudh Sudarshan, and Shahin Sefati. Torchrec: a pytorch domain library for recommendation systems. In *Proceedings of the 16th ACM Conference on Recommender Systems*, pages 482–483, 2022.

[16] J. Kiefer and Jacob Wolfowitz. Stochastic estimation of the maximum of a regression function. *Annals of Mathematical Statistics*, 23:462–466, 1952.

[17] Soojeong Kim, Gyeong-In Yu, Hojin Park, Sungwoo Cho, Eunji Jeong, Hyeonmin Ha, Sanha Lee, Joo Seong Jeong, and Byung-Gon Chun. Parallax: Sparsity-aware data parallel training of deep neural networks. In George Candea, Robbert van Renesse, and Christof Fetzer, editors, *Proceedings of the Fourteenth EuroSys Conference*

2019, Dresden, Germany, March 25-28, 2019, pages 43:1–43:15. ACM, 2019.

[18] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

[19] Thorsten Kurth, Sean Treichler, Joshua Romero, Mayur Mudigonda, Nathan Luehr, Everett Phillips, Ankur Mahesh, Michael Matheson, Jack Deslippe, Massimiliano Fatica, et al. Exascale deep learning for climate analytics. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 649–660. IEEE, 2018.

[20] Fan Lai, Wei Zhang, Rui Liu, William Tsai, Xiaohan Wei, Yuxi Hu, Sabin Devkota, Jianyu Huang, Jongsoo Park, Xing Liu, et al. Adaembed: Adaptive embedding for Large-Scale recommendation models. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 817–831, 2023.

[21] Xiangru Lian, Binhang Yuan, Xuefeng Zhu, Yulong Wang, Yongjun He, Honghuan Wu, Lei Sun, Haodong Lyu, Chengjun Liu, Xing Dong, et al. Persia: An open, hybrid system scaling deep learning-based recommenders up to 100 trillion parameters. In *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 3288–3298, 2022.

[22] Zixuan Ma, Jiaao He, Jiezhong Qiu, Huanqi Cao, Yuanwei Wang, Zhenbo Sun, Liyan Zheng, Haojie Wang, Shizhi Tang, Tianyu Zheng, et al. Bagualu: targeting brain scale pretrained models with over 37 million cores. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 192–204, 2022.

[23] Peter Mattson, Christine Cheng, Gregory Diamos, Cody Coleman, Paulius Micikevicius, David Patterson, Hanlin Tang, Gu-Yeon Wei, Peter Bailis, Victor Bittorf, et al. Mlperf training benchmark. *Proceedings of Machine Learning and Systems*, 2:336–349, 2020.

[24] Xupeng Miao, Hailin Zhang, Yining Shi, Xiaonan Nie, Zhi Yang, Yangyu Tao, and Bin Cui. Het: Scaling out huge embedding model training via cache-enabled distributed framework. *Proc. VLDB Endow.*, 15(2):312–320, 2021.

[25] Dheevatsa Mudigere, Yuchen Hao, Jianyu Huang, Zhihao Jia, Andrew Tulloch, Srinivas Sridharan, Xing Liu, Mustafa Ozdal, Jade Nie, Jongsoo Park, et al. Software-hardware co-design for fast and scalable training of deep learning recommendation models. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, pages 993–1011, 2022.

[26] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, et al. Efficient large-scale language model training on gpu clusters using megatron-lm. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15, 2021.

[27] Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, Xiaodong Wang, Udit Gupta, Carole-Jean Wu, Alisson G Azzolini, et al. Deep learning recommendation model for personalization and recommendation systems. *arXiv preprint arXiv:1906.00091*, 2019.

[28] Zaifeng Pan, Zhen Zheng, Feng Zhang, Bing Xie, Ruofan Wu, Shaden Smith, Chuanjie Liu, Olatunji Ruwase, Xiaoyong Du, and Yufei Ding. Recflex: Enabling feature heterogeneity-aware optimization for deep recommendation models with flexible schedules. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, SC '24. IEEE Press, 2024.

[29] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

[30] Kishore Punniyamurthy, Khaled Hamidouche, and Bradford M. Beckmann. Optimizing distributed ml communication with fused computation-collective operations. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, SC '24. IEEE Press, 2024.

[31] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. Zero: Memory optimizations toward training trillion parameter models. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–16. IEEE, 2020.

[32] Samyam Rajbhandari, Olatunji Ruwase, Jeff Rasley, Shaden Smith, and Yuxiong He. Zero-infinity: Breaking the gpu memory wall for extreme scale deep learning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14, 2021.

[33] Geet Sethi, Bilge Acun, Niket Agarwal, Christos Kozyrakis, Caroline Trippel, and Carole-Jean Wu. Recshard: statistical feature-based memory optimization for industry-scale neural recommendation. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 344–358, 2022.

[34] Geet Sethi, Pallab Bhattacharya, Dhruv Choudhary, Carole-Jean Wu, and Christos Kozyrakis. Flexshard: Flexible sharding for industry-scale sequence recommendation models. *arXiv preprint arXiv:2301.02959*, 2023.

[35] Hao-Jun Michael Shi, Dheevatsa Mudigere, Maxim Naumov, and Jiyan Yang. Compositional embeddings using complementary partitions for memory-efficient recommendation systems. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '20, page 165–175, New York, NY, USA, 2020. Association for Computing Machinery.

[36] Hongyan Tang, Junning Liu, Ming Zhao, and Xudong Gong. Progressive layered extraction (ple): A novel multi-task learning (mtl) model for personalized recommendations. In *Proceedings of the 14th ACM Conference on Recommender Systems*, pages 269–278, 2020.

[37] Shashank Verma, Wenwen Gao, Hao Wu, Deyu Fu, and Tomasz Grel. Fast, terabyte-scale recommender training made easy with nvidia merlin distributed-embeddings. https://github.com/NVIDIA-Merlin/distributed-embeddings, 2022.

[38] Weihu Wang, Yaqi Xia, Donglin Yang, Xiaobo Zhou, and Dazhao Cheng. Accelerating distributed dlrm training with optimized tt decomposition and micro-batching. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, SC '24. IEEE Press, 2024.

[39] Zehuan Wang, Yingcan Wei, Minseok Lee, Matthias Langer, Fan Yu, Jie Liu, Shijie Liu, Daniel G Abel, Xu Guo, Jianbing Dong, et al. Merlin hugectr: Gpu-accelerated recommender system training and inference. In *Proceedings of the 16th ACM Conference on Recommender Systems*, pages 534–537, 2022.

[40] Zheng Wang, Yuke Wang, Boyuan Feng, Dheevatsa Mudigere, Bharath Muthiah, and Yufei Ding. El-rec: efficient large-scale recommendation model training via tensor-train embedding table. In *2022 SC22: International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1007–1020. IEEE Computer Society, 2022.

[41] Carole-Jean Wu, Robin Burke, Ed H Chi, Joseph Konstan, Julian McAuley, Yves Raimond, and Hao Zhang. Developing a recommendation benchmark for mlperf training and inference. *arXiv preprint arXiv:2003.07336*, 2020.

[42] Minhui Xie, Youyou Lu, Jiazhen Lin, Qing Wang, Jian Gao, Kai Ren, and Jiwu Shu. Fleche: an efficient gpu embedding cache for personalized recommendations. In *Proceedings of the Seventeenth European Conference on Computer Systems*, pages 402–416, 2022.

[43] Minhui Xie, Kai Ren, Youyou Lu, Guangxu Yang, Qingxing Xu, Bihai Wu, Jiazhen Lin, Hongbo Ao, Wanhong Xu, and Jiwu Shu. Kraken: memory-efficient continual learning for large-scale real-time recommendations. In Christine Cuicchi, Irene Qualters, and William T. Kramer, editors, *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2020, Virtual Event / Atlanta, Georgia, USA, November 9-19, 2020*, page 21. IEEE/ACM, 2020.

[44] Daochen Zha, Louis Feng, Qiaoyu Tan, Zirui Liu, Kwei-Herng Lai, Bhargav Bhushanam, Yuandong Tian, Arun Kejariwal, and Xia Hu. Dreamshard: Generalizable embedding table placement for recommender systems. *Advances in Neural Information Processing Systems*, 35:15190–15203, 2022.

[45] Qianru Zhang, Lianghao Xia, Xuheng Cai, Siu-Ming Yiu, Chao Huang, and Christian S. Jensen. Graph augmentation for recommendation. In *40th IEEE International Conference on Data Engineering, ICDE 2024, Utrecht, The Netherlands, May 13-16, 2024*, pages 557–569. IEEE, 2024.

[46] Mark Zhao, Dhruv Choudhary, Devashish Tyagi, Ajay Somani, Max Kaplan, Sung-Han Lin, Sarunya Pumma, Jongsoo Park, Aarti Basant, Niket Agarwal, Carole-Jean Wu, and Christos Kozyrakis. Recd: Deduplication for end-to-end deep learning recommendation model training infrastructure. *CoRR*, abs/2211.05239, 2022.

[47] Weijie Zhao, Deping Xie, Ronglai Jia, Yulei Qian, Ruiquan Ding, Mingming Sun, and Ping Li. Distributed hierarchical gpu parameter server for massive scale deep learning ads systems. *Proceedings of Machine Learning and Systems*, 2:412–428, 2020.

[48] Weijie Zhao, Jingyuan Zhang, Deping Xie, Yulei Qian, Ronglai Jia, and Ping Li. Aibox: Ctr prediction model training on a single node. In *Proceedings of the 28th ACM International Conference on Information and Knowledge Management*, pages 319–328, 2019.

[49] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P. Xing, Joseph E. Gonzalez, and Ion Stoica. Alpa: Automating inter- and Intra-Operator parallelism for distributed deep learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 559–578, Carlsbad, CA, 2022. USENIX Association.

CrossPipe: Towards Optimal Pipeline Schedules for Cross-Datacenter Training

Tiancheng Chen[1], Ales Kubicek[1], Langwen Huang[1] and Torsten Hoefler[1]

[1]Department of Computer Science, ETH Zürich

Abstract

Training large language models (LLMs) now requires resources that exceed a single datacenter, making cross-datacenter strategies increasingly crucial. We present CrossPipe, a framework designed to optimize model training across geographically distributed datacenters by explicitly modeling and mitigating the impact of network latency and limited bandwidth. It enables unified analysis and optimization incorporating both pipeline parallelism (PP) and opportunities for overlapping data parallelism (DP) communication. CrossPipe generates optimized pipeline schedules using either solver-based optimal or fast near-optimal greedy algorithms, built upon a flexible execution engine that separates scheduling logic from communication details. Our evaluation shows that CrossPipe reduces training time by up to 33.6% compared to traditional pipeline schedules under identical memory constraints. When memory constraints are relaxed, CrossPipe maintains strong performance despite communication delays, approaching the efficiency of idealized schedules without delays. CrossPipe offers improved scalability and resource utilization, particularly in environments with high network latency or limited bandwidth.

1 Introduction

Large language models (LLMs) have revolutionized natural language processing, demonstrating remarkable capabilities in tasks such as text generation, translation, and question answering. These models, trained on massive datasets, exhibit sophisticated context understanding and generate human-like responses. Their applications span scientific research [59], content generation [29], and personal assistants [31]. LLM training consists of three stages: pre-training, fine-tuning, and alignment, with pre-training consuming the majority of computational resources [11, 15, 66].

As LLM performance scales with model size and data volume, computational demands have increased exponentially. Recent studies estimate that training compute for state-of-the-art models quadruples annually [48], necessitating both vertical scaling (faster accelerators) and horizontal scaling (distributed computation). While GPU performance and energy efficiency continue to improve steadily [16], the power and infrastructure required to support LLM training is growing even faster. If current trends hold, GPU counts must nearly triple each year, with power consumption rising accordingly.

The escalating computational demands of LLMs are straining existing infrastructure, particularly power supply systems. To address these energy requirements, companies like Microsoft, Google, and Amazon are turning to nuclear energy sources to power their new AI datacenters [7, 36, 46], emphasizing the need for reliable and high-capacity power sources. Scaling a single datacenter introduces challenges including local power limitations and increased vulnerability to outages. Report suggests that deploying multiple smaller facilities is more practical than scaling a single massive one [5].

As LLM scale grows, multi-datacenter training is becoming essential [1], distributing both compute and energy loads. However, geographic distribution introduces significant communication inefficiencies that must be addressed to support this shift. In the context of cloud-based training, allocating large blocks of GPUs in one region is often infeasible [54], making cross-regional GPU acquisition a practical alternative. The high cross-region communication cost poses challenges to the efficiency of existing training methods. This work attempts to assess the impact of network inefficiencies to synchronous training tasks and improve the performance to reduce the cost and energy consumption.

This paper introduces CrossPipe[1], a framework that improves the efficiency of cross-datacenter (cross-DC) LLM training through the following contributions:

- **Analysis:** We present a comprehensive analysis of cross-DC training methodologies and show that pipeline parallelism is the most feasible approach in this setting.
- **Performance Model and Algorithm:** We present a latency

[1]The code is available at https://github.com/spcl/crosspipe.

and bandwidth-aware performance model specifically designed for the cross-DC environment. This model enables the co-optimization of pipeline schedules with potential data parallelism (DP) communication overlap, unifying the modeling of cross-DC PP and cross-DC DP. Next, we introduce a system-aware pipeline schedule generation algorithm: CrossPipe. The algorithm leverages either constraint optimization techniques to generate *optimal* cross-DC pipeline schedules (Section 4.1) or fast greedy algorithm to generate efficient and *near-optimal* schedules (Section 4.2).

- **Framework:** Finally, we propose and implement a flexible and easily extensible pipeline execution engine featuring a two-layer abstraction that decouples block scheduling from communication arrangement (detailed in Section 6.3). This design enables efficient deployment of different pipeline schedules, including those generated by CrossPipe.

2 Cross-DC Training

2.1 Parallelism Strategies

Distributed LLM pre-training [4] employs a combination of different parallelism strategies (termed *hybrid parallelism*) to partition the workload across GPU clusters. Table 1 lists symbols and notations used in this paper.

Tensor Parallelism (TP) [2]: Splits each model layer across multiple GPUs [27,53], requiring extensive collective communication (e.g., Reduce-Scatter and Allgather [37]) during both forward and backward passes. Due to limited opportunities for overlap [64] and high communication costs, TP is typically restricted to high-bandwidth domains (e.g., NVLink [40]), making it unsuitable for spanning geo-distributed DCs.

Pipeline Parallelism (PP): Divides the model layers into n_{PP} *stages*, with each stage assigned to a different GPU. Communication occurs only at stage boundaries via point-to-point send/receive of activations and gradients.

Data Parallelism (DP): Replicates the full model on each GPU, where distinct batches are processed independently and gradients are synchronized across replicas. DP is usually applied with ZeRO [45] to reduce memory redundancy. This work assumes DP with ZeRO stage 1, partitioning optimizer states without increasing communication overhead compared to vanilla DP. Higher ZeRO stages introduce extra communication with diminishing memory savings.

Sequence Parallelism (SP) [3]: Scales sequence dimension [21,33] and is typically applied at the end of pre-training to increase model context window [12].

Expert Parallelism (EP): Distributes the expert MLPs in Mixture of Experts (MoE) [20, 51] models. The per-layer, high-volume, and dynamic Alltoall communication in EP makes it challenging to deploy on cross-DC links.

[2]Also known as Operator Parallelism.
[3]Also known as Context Parallelism.

Notation	Description
n_{DC}	# of datacenters
$n_{\{TP,PP,DP\}}$	TP, PP, DP size
$T_{\{F,B,W\}}$	Runtime of F, B, W block
$M_{\{F,B,W,L\}}$	Net memory change in F, B, W block and memory budget
M_L	Memory limit per device
α, β	Communication latency and inverse of bandwidth
T_α, T_β	Communication cost matrix
b, $\hat{B}$	Microbatch size and global batch size
n_{mb}, ε	Number of microbatches per DP rank, and ratio n_{mb}/n_{PP}
s, d	Model sequence length and hidden dimension
n_{sub}	Number of parts in a sub-block schedule (Section 4.2)

Table 1: List of symbols and notations.

Key Insight: TP, SP, and EP introduce layer-wise communication with high frequency and/or volume, making them highly sensitive to the latency and limited bandwidth typical of cross-DC links. Therefore, PP and DP emerge as the primary candidates for cross-DC traffic, due to their less frequent (PP, DP) or point-to-point (PP) communication patterns.

2.2 Communication Model

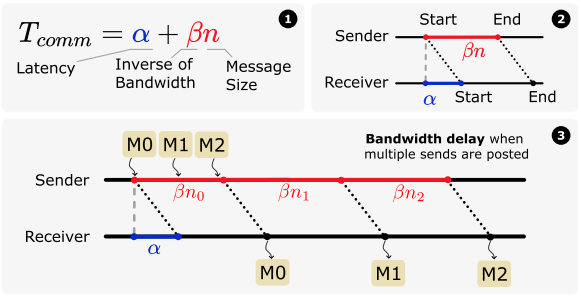


Figure 1: Alpha-Beta communication model ❶ sending a single message ❷. When multiple pending messages are to be sent, the latter messages need to wait for the previous ones to be placed on the transmission link first. This results in an extra bandwidth delay ❸.

In this work, we assume a small number of DCs (e.g., $n_{DC} \leq 4$). For modeling the communication time T_{comm}, we adopt the Alpha-Beta model, accounting for both latency (α) and bandwidth limitations (β). Multiple concurrent messages incur additional queuing delays, illustrated in Figure 1.

2.3 Distributed Training Infrastructure

Cross-DC infrastructure setups can be categorized into four primary types, as shown in Figure 2. We differentiate between high-performance clusters and public cloud environments, further classified by geographic proximity: either nearby (same-campus, same-region) or distant (cross-campus, cross-region).

- **Same-Campus Clusters** ❶ setup represents tightly interconnected DCs on the same campus, typically connected via frontend network or additional switch layer. This setup features low latency (up to 10 us [4]) and high bandwidth (800 Gb/s [4] per port), making communication overhead negligible, but still slightly higher than a single cluster.
- **Cross-Campus Clusters** ❷ setup represents geographically distributed DCs (typically up to 40 km apart, using public products like NVIDIA MetroX) interconnected with high-bandwidth links (200 Gb/s [4] per port). Network latency is bounded by the physical distance (10-200 us [4]).
- **Same-Region Cloud** ❸ setup represents closely allocated instances within the same cloud region. This setup features low bandwidth (around 11.3 Gb/s [5]) [55] and higher latency (around 1 ms [5]) [55] compared to the setups discussed above due to the usage of less specialized networking hardware.
- **Cross-Region Cloud** ❹ setup represents instances allocated across cloud regions or even continents. This setup inherits the same properties as a same-region cloud but with significantly higher latency (30-100 ms [5]) [55] and even lower bandwidth (1.4-5.0 Gb/s [5]) [55] due to distance.

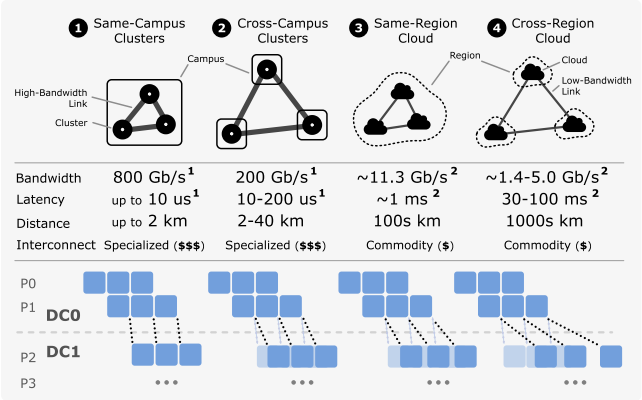


Figure 2: Cross-DC infrastructure setup types and their impact on the PP communication at DC boundaries.

Table 2: The hidden dimension d, number of parameters N of LLMs. D and E refer to dense and MoE models respectively.

Model	d	N (10^9)	N/d (10^6)
Mistral 7B (D) [23]	4096	7.24	1.77
Mixtral 8x7B (E) [24]	4096	46.7	11.4
Qwen2.5 32B (D) [44]	5120	32.8	6.41
DeepSeek V3 (E) [9]	7168	685	95.6
Llama 3 405B (D) [12]	16384	406	24.8

These infrastructure variations drastically affect communication characteristics. To analyze their impact independent of specific hardware or model configurations, we normalize the communication time components (T_{lat} and T_{bw}) by the max per-microbatch forward computation time per stage (T_F), see Section 3.3). This yields dimensionless ratios, T_{lat}/T_F and T_{bw}/T_F, which capture the relative cost of communication.

2.4 Cross-DC Parallel Dimension Selection

Hybrid parallelism exhibits structured communication patterns, with over 99% of GPU pairs having no direct traffic [65]. The choice of parallelism strategy across DCs significantly influences training efficiency. Figure 3 compares two viable options identified in Section 2.1: cross-DC PP and cross-DC DP. Cross-DC PP communication volume is characterized by $sd * n_{DP}$ while cross-DC DP communication volume is characterized by model parameters N. The key hyperparameters of some LLMs are shown in Table 2. We analyzed both cross-DC PP and cross-DC DP for Llama 3 405B in Section 5.2, demonstrating that cross-DC PP is generally the better choice. The increased popularity of MoE models further shifts the preference towards cross-DC PP [6], since experts in MoE models introduce extra DP communication volume compared to dense models with similar width d.

3 Pipeline Model

3.1 Computation Blocks

PP partitions model chunks across n_{PP} stages, processing input microbatches in sequence. During the backward pass, gradients propagate in reverse, from the last model chunk back to the initial one. Periods when devices remain idle while awaiting required data are termed pipeline bubbles. To evaluate pipeline efficiency, we define the bubble ratio as the fraction of idle time over total time per device.

We denote the Forward computation for each chunk as the F block, and the Backward computation as the B block. Each B block can be further split into an input data gradient computation block (DGrad, or D) and a weight gradient computation block (WGrad, or W) [43]. An illustration of this decomposition is provided in Appendix B. This finer granularity facilitates the construction of more efficient pipeline schedules with reduced bubble ratios.

3.2 System Parameters

Although our focus is on training Transformer-based [61] LLMs in cross-DC environments, our approach applies to large models structured as a sequence of layers. Relevant

[4] Calculation based on the best commercially available hardware (NVIDIA LinkX / MetroX), assuming 5 ns/m.

[5] Measured.

[6] Llama 3 405B is trained with $s = 8192$, $n_{DP} = 128$, yielding $N/sdn_{DP} \approx 23.6$. DeepSeek-V3 is trained with $s = 4096$, $n_{DP} = 128$, yielding $N/sdn_{DP} \approx 182$. Notice that Llama 3 405B is wider in d and therefore can be viewed as the basis of a MoE model much larger than DeepSeek-V3.

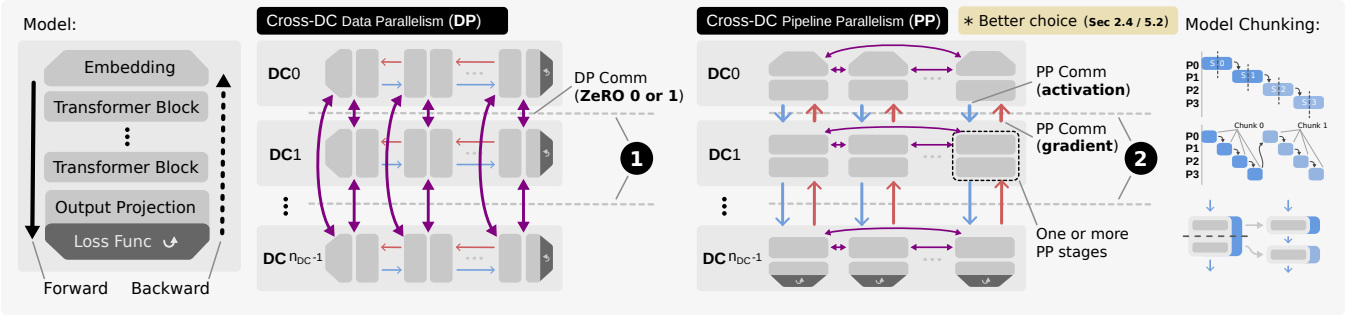


Figure 3: Typical LLM architecture (left). **Cross-DC DP:** Each DC maintains independent model copies. ❶ Collective operations (Allgather and Reduce-Scatter, or Allreduce) synchronize gradients and update parameters (ZeRO stage 0 or 1). **Cross-DC PP:** The model is partitioned among DCs at layer boundaries. Each DC holds one or more pipeline *stages*. DP communication happens internally within each DC to synchronize gradients of stages each holds. ❷ Inter-DC communication employs point-to-point send/receive operations for exchanging activations and gradients. Stages can further split into finer *chunks* to enhance scheduling efficiency (Section 3.3).

system parameters include memory usage, per-chunk computation time, and inter-stage communication latency and bandwidth delays. Memory consumption encompasses static elements (parameters, gradients, optimizer states) and dynamic allocations (activations cached during F blocks and released after D and W).

3.3 Pipeline Schedules

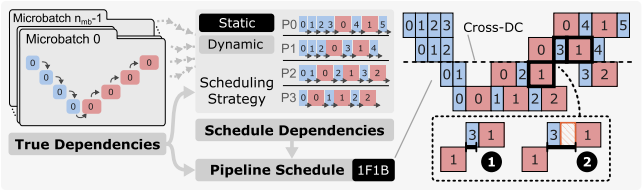


Figure 4: Construction of a 1F1B static schedule. True dependencies guide the creation of schedule dependencies. The resulting acyclic dependency graph governs execution and is used for runtime estimation (Section 3.5). Highlighted parts (right) show the timing of activation/gradient arrival either ❶ enables immediate scheduling or ❷ causes delays (bubbles).

Pipeline schedules are represented as acyclic dependency graphs, with vertices as pipeline blocks and edges representing two types of dependencies: true dependencies (data dependencies within each microbatch) and schedule dependencies (execution order within each pipeline stage). The construction of a 1F1B schedule is illustrated in Figure 4.

True dependencies reflect actual data flow across blocks (forward for activations, backward for gradients). Figure 5 shows key traversal (data flow) patterns: Unidirectional (UD), Bidirectional (BD), Loop, and Wave.

Schedule dependencies define the execution order of pipeline blocks within each stage. These dependencies are determined using either a static or dynamic scheduling strategy. Static strategies predetermine block placements (usually hand-optimized), while dynamic strategies adapt based on system parameters. CrossPipe schedules (Section 4) fall under dynamic scheduling.

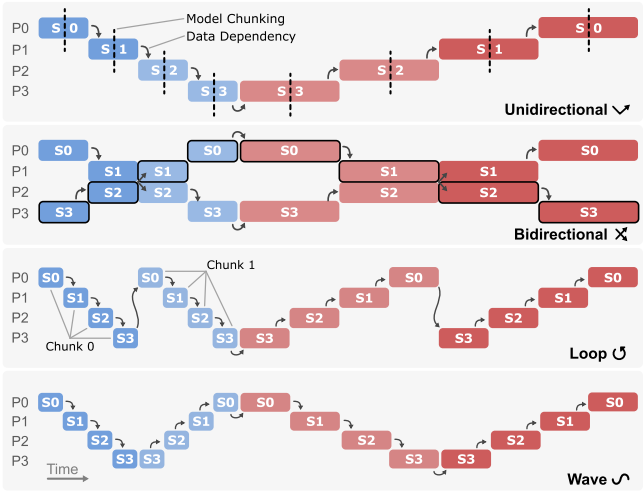


Figure 5: Traversal patterns for a single microbatch (two microbatches shown in the Bidirectional pattern). Loop and Wave patterns leverage model chunking to refine granularity.

3.4 Problems of Static Scheduling

In the presence of communication delays, we identify two major limitations that reduce the efficiency of static schedules in cross-DC PP: static execution order (scheduling-level) and static communication arrangement (implementation-level).

Static Execution Order: Static schedules are optimized under the assumption of negligible communication cost, as in single-DC settings. When directly applied to cross-DC training, they fail to adapt to higher communication delays, resulting in pipeline inefficiencies visualized as *bubble strides* (illustrated in ❷ Figure 6 and Appendix C). **Schedule** ❷ in

Figure 6 depicts a critical path in a 1F1B schedule across 2 DCs involving 8 cross-DC PP communications. Since the path consists solely of true and schedule dependencies (Section 3.3), its length imposes a lower bound on overall runtime. For a 1F1B schedule of n_{mb} microbatches, there exists a path containing $O(n_{mb})$ cross-DC communications. As a result, communication delays are amplified proportionally, significantly degrading training throughput. A detailed analysis of this amplification effect is presented in Section 5. CrossPipe addresses this limitation via dynamic scheduling strategies, detailed in Section 4. **Schedule ❸** in Figure 6 illustrates that reordering pipeline blocks can improve efficiency while maintaining the same peak activation memory if needed.

Static Communication Arrangement: Existing frameworks such as Megatron-LM often group pipeline communication operations (e.g., GPU 0 sending to GPU 1 while receiving from GPU 1) for simplicity and hardware efficiency, which introduces implicit synchronization. Moreover, even if this grouping is avoided, the two-sided communication pattern introduces synchronization between the sender and receiver in each send/recv operation. Due to variations in stage execution time, the receiver may fail to post the corresponding receive in time, causing the sender to wait. These delays disrupt stage alignment, which many hand-optimized schedules assume, and propagate bubbles across the pipeline. The interleaved 1F1B schedule [39] overlaps communication with one computation block to mitigate this synchronization cost. However, its static design is only effective under small delays. To address this, CrossPipe decouples scheduling logic from communication orchestration, allowing more fine-grained and adaptive execution, as elaborated in Section 6.3.

3.5 Pipeline Performance Model

We develop a performance model to estimate the runtime of a pipeline by leveraging the topological ordering of its dependency graph (Section 3.3). This model assumes that scheduling and communication orchestration are decoupled, thereby excluding delays caused by synchronization overhead (see Section 3.4). The start time of each block is determined by the maximum of two values: (1) the completion time of the preceding block on the same stage (❶ in Figure 4), and (2) the completion time of the dependent block plus the communication delay (❷ in Figure 4). The communication delay consists of a fixed latency component and a bandwidth-related component, which depends on both link bandwidth and current occupancy (Section 2.2).

4 CrossPipe Schedules

4.1 Optimal Schedule

The optimal pipeline schedule depends on the system parameters and can be framed as a job scheduling problem. Prior

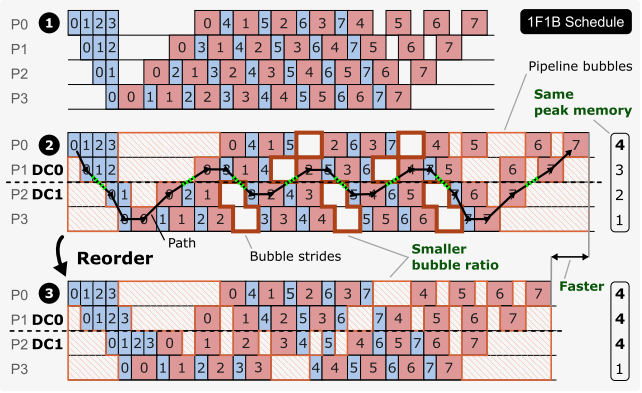


Figure 6: ❶ Original 1F1B schedule. ❷ 1F1B schedule with Cross-DC PP communication which leads to *bubble strides*. A *Path* is depicted (⟶) including cross-DC boundary crossings (⟿). ❸ The schedule after reordering is more efficient while maintaining the same peak memory. More microbatches or memory budget can help to further reduce runtime.

work [43] formulates this using mixed integer linear programming. Building on this, we elevate communication operations to first-class citizens alongside computation, incorporating both latency and bandwidth delays into the formulation, and generalize this to traversal patterns (Figure 5). This leads us to define the problem as a constraint optimization (CO) task. In addition to yielding start and end times for all operations, the solution inherently determines the execution order of communication operations that share the same cross-DC link, thereby handling link contention and scheduling order of cross-DC communications implicitly.

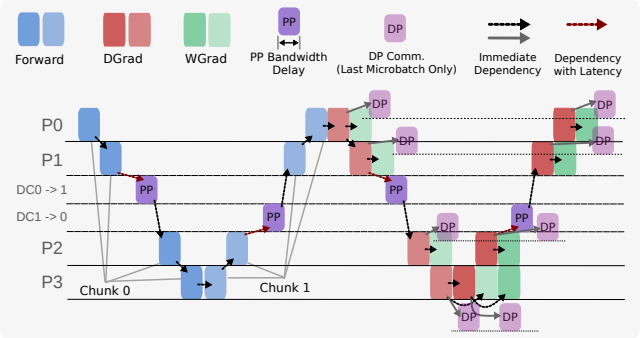


Figure 7: The data dependency in a Wave schedule with modeling of both computation and communication. For each model chunk, the DP communication only depends on the W block of last microbatch. The modeling of vanilla DP is shown in the figure. For DP with ZeRO stage 1, an Allgather block should precede the F block of the first microbatch in each model chunk.

Sets and Indices Every *compute* operation is uniquely identified by the triple $(s,k,t,m) \in \mathcal{S} \times \mathcal{K} \times \mathcal{T} \times \mathcal{M}$. Each communication operation $c \in \mathcal{C}$ transfers the data produced by a

compute operation on stage $\text{src}(c)$ to stage $\text{dst}(c)$.

- $\mathcal{M}$: Microbatches, indexed by $mb = 1, \ldots, n_{mb}$
- $\mathcal{S}$: Devices (pipeline stages), indexed by $s = 1, \ldots, n_{PP}$
- $\mathcal{T}$: Operation types, $\mathcal{T} = \{F, D, W\}$
- $\mathcal{K}$: Model chunks
- $\mathcal{P}$: Compute operations (indexed by (s, k, t, m))
- $\mathcal{C}$: Communication operations
- $\mathcal{O} = \mathcal{P} \cup \mathcal{C}$: All operations

Inputs The input variables are closely related to system parameters in Section 3.2.

- d_o: Duration of $o \in \mathcal{O}$ (bandwidth time for $c \in \mathcal{C}$, computation time otherwise)
- ℓ_c: Latency delay for $c \in \mathcal{C}$
- m_p: Net memory change after compute $p \in \mathcal{P}$ completes
- $M_s^{\max}$: Memory limit of device s
- $\text{Pred}(o)$: Immediate predecessor of o in data dependency

Decision Variables

- $t_o \in \mathbb{R}_{\geq 0}$: Start time of operation $o \in \mathcal{O}$
- $x_{o,o'} \in \{0, 1\}$: Order for operations sharing a device/link

Constraints

- **Data Dependencies** An example of data dependency of Wave schedules is shown in Figure 7.

$$\forall o \in \mathcal{O},\ \forall p \in \text{Pred}(o): \quad t_o \geq t_p + d_p + \begin{cases} \ell_p & \text{if } p \in \mathcal{C} \\ 0 & \text{if } p \in \mathcal{P} \end{cases}$$

- **Resource Non-overlap** For any $o, o' \in \mathcal{O}$ sharing a device or link, overlapping is not allowed:

$$t_o + d_o \leq t_{o'} + H(1 - x_{o,o'})$$
$$t_{o'} + d_{o'} \leq t_o + H x_{o,o'}$$

  H is a large constant bounding the scheduling horizon.
- **On-device Memory Capacity** Let $u_{p,q} = 1$ iff compute p completes before q starts. Then for all $s \in \mathcal{S}$ and $q \in \mathcal{P}$ assigned to s:

$$\sum_{\substack{p \in \mathcal{P} \\ \text{device}(p) = s}} m_p u_{p,q} \leq M_s^{\max}$$

- **Microbatch Order within Stage and Type** For any $o, o' \in \mathcal{P}$ on same device s and type $t \in \mathcal{T}$, if microbatch index $mb(o) < mb(o')$, then:

$$t_o + d_o \leq t_{o'}$$

  This constraint reduces the search space.

Objective We minimize the makespan, defined as the time from the earliest start to the latest finish on the first device:

$$\min \left(t_{\text{last}(0)} + d_{\text{last}(0)} - t_{\text{first}(0)} \right)$$

DP Overlap Modeling DP communication can be modeled as distinct operations triggered after the completion of the W block for the final microbatch of the current model chunk. In the case of ZeRO stage 1, the corresponding weight All-gather operations are scheduled before the first F block of each model chunk. An illustrative example is shown in Figure 7. The objective is then extended to account for the added communication.

Solver Scalability We evaluate the runtime performance of both MILP and CO solvers on identical pipeline scheduling problems, as detailed in Appendix D.1. It demonstrates the feasibility of using solver-based methods in production environments. However, the search space expands rapidly with the number of pipeline stages, which can lead to prohibitive computation time for extremely large-scale scenarios or dynamically changing system parameters. In such cases, solver-based scheduling becomes less practical, motivating the need for alternative approaches (Section 4.2).

4.2 Greedy Schedule

To address the scalability limitations of solver-based approaches, we introduce a greedy schedule generation algorithm. This method is designed to rapidly generate near-optimal pipeline schedules while adapting to potentially dynamic system conditions.

4.2.1 Greedy Sub-block Scheduling

The greedy algorithm operates using *local* information, only considering scheduled blocks and those ready for scheduling. To counter the suboptimal decisions typical of greedy methods, we employ block-splitting: each computation block is divided into n_{sub} sub-blocks. This finer granularity enhances the ability to reduce pipeline bubbles with negligible scheduling overhead compared to training iteration time.

Algorithm Inputs The inputs to the greedy algorithm include: Per-stage runtimes of F, D, and W blocks: T_F, T_D, T_W; corresponding memory usage: M_F, M_D, and M_W; per-stage memory limit: M_L; communication delay matrices: α and β, where $\alpha[i, j]$ and $\beta[i, j]$ represent latency and bandwidth inverse of communication from device i to device j. The scheduling procedure for the *CrossUDSub* schedule is outlined in Algorithm 1.

4.2.2 Scheduling Loop

The scheduling loop consists of three core steps:

Stage Selection The `next_stage_to_schedule` method identifies the stage with the earliest schedulable time, which is defined as the maximum between the end time of the last scheduled operation and the earliest available time of schedulable operations.

Algorithm 1 Greedy Generation for *CrossUDSub* Schedule

Output: Per-stage schedule $S_d, \forall d \in [n_{PP}]$
for i in $[n_{mb}]$ **do**
  Add F_i operation for stage 0 to S_0's schedulable operations.
end for
while True **do** *Scheduling Loop (Sec 4.2.2)*
  $cur \leftarrow$ `next_stage_to_schedule()`
  if no stage is schedulable **then**
    break
  end if
  $p_{cur} \leftarrow$ schedulable operation of highest priority on stage cur
  Schedule next sub-block of p_{cur}
  if $p_{cur}.type$ = D **and** no remaining sub-blocks of p_{cur} **then**
    Add W operation to schedulable operations of current stage
  end if
  Let p_{next} be the operation dependent on p_{cur}
  if p_{next} exists **and** no remaining sub-blocks of p_{cur} **then**
    $T_{lat} \leftarrow \alpha[cur, next]$
    $T_{bw} \leftarrow \beta[cur, next] * \text{Msg_Size}$
    $E_{bw} \leftarrow$ `bw_model`$(p_{cur}.T_{end}, cur, next, T_{bw})$
    $p_{next}.T_{avail} \leftarrow E_{bw} + T_{lat}$
    Add p_{next} to S_{next}'s schedulable operations.
  end if
end while

Operation Selection The scheduler selects operations available at or after the end of the last scheduled operation on the chosen stage. When multiple options exist, it applies a heuristic priority across three phases:

- **Warm-up phase**: prioritizes F blocks
- **Steady phase**: interleaves F and D full blocks
- **Tear-down phase**: prioritizes D over W blocks

When memory constraints prevent scheduling F or D blocks, a W sub-block is scheduled.

Operation Scheduling The selected operation is scheduled on its stage. If it is the last sub-block of a D block, the corresponding W is added to the schedulable operations of the stage. The dependent blocks are then made schedulable on the receiving stage with the earliest start time calculated using the communication model.

4.2.3 Bandwidth Occupancy Model

To model bandwidth contention, we use a simple range-based bandwidth occupancy model. Communication is assumed to begin immediately upon completion of the relevant computation block. The `BW_model`$(T_{ready}, src, dst, T_{bw})$ function determines the earliest available transmission window of length T_{bw}, starting at or after T_{ready}, and returns its end time.

4.2.4 Performance Characteristics

While greedy algorithms do not guarantee global optimality, our approach demonstrates strong empirical performance. As detailed in Section 5&7, the CrossUDSub schedule achieves:

- Equivalent performance to ZB-H1 [43] under negligible communication delays and same memory constraints.
- Faster than static schedules under non-negligible communication delays by filling sub-block size bubbles.
- Further improvements when memory constraints are relaxed, allowing greater scheduling flexibility.

4.2.5 Time Complexity

The main loop executes $3n_{mb}n_{sub}n_{PP}$ iterations, with each iteration scheduling one sub-block. Identifying the next stage and highest priority operation incurs $O(n_{mb}n_{sub}\log(n_{mb}n_{sub}))$ cost. Hence, the overall complexity is $O(n_{mb}^2 n_{sub}^2 n_{PP}\log(n_{mb}n_{sub}))$. In practice, since the number of schedulable operations per stage remains small, the runtime approximates $O(c \cdot n_{mb}n_{sub}n_{PP})$ for a small constant c.

5 Analysis

In this section, we use simulation experiments based on the performance model described in Section 3.5 to investigate two key questions: (1) How do different pipeline schedules respond to latency and bandwidth delays? (2) Between cross-DC PP and cross-DC DP, which is more efficient for training in cross-DC settings?

5.1 Schedule Efficiency

Schedule	Type	WGrad	Bubble Ratio	Memory	DP Overlap
1F1B [38]	UD	Combined	High	Medium	Medium
IV1F1B [39]	Loop	Combined	Medium	Medium+	Medium+
ZBH1 [43]	UD	Split	Medium	Medium	Low
ZBV [43]	Wave	Split	Low	Medium	Low

Table 3: Static pipeline schedules used in the analysis. UD and BD stands for unidirectional and bidirectional. IV1F1B is the abbreviation for the interleaved 1F1B schedule.

We compare various pipeline schedules under increasing communication delay in a cross-DC PP setting. Bidirectional (BD) schedules are excluded as they involve both PP and DP cross-DC communication. The main static schedules that we focus on are summarized in Table 8. We use $n_{PP} = 4, n_{mb} = 8$, and simulate 2 DCs with 2 stages each. Dynamic schedules are generated with the same memory limits as their static counterparts (e.g., CrossUD mirrors 1F1B). Delay sensitivity is measured as slowdown relative to the ZBV schedule under no communication delay. Delay is varied using T_{lat}/T_F (latency delay) and T_{bw}/T_F (bandwidth delay), where T_F is the per-stage forward computation time. Key observations from Figure 8 include:

- WGrad-split schedules consistently outperform unified-backward ones due to finer scheduling granularity.
- Wave schedules are more efficient in low-delay settings, while UD schedules become superior as delays grow.
- Loop schedules show the highest sensitivity to delays, due to more frequent cross-DC communication (6 per micro-batch, compared to 4 for Wave and 2 for UD).

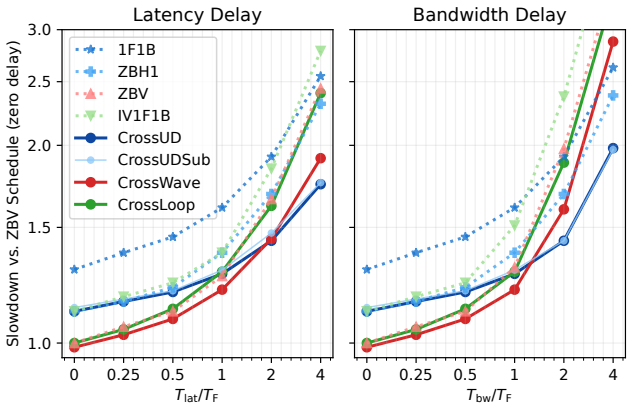


Figure 8: Impact of latency and bandwidth delay on runtime across different pipeline schedules. Static (····) and dynamic (—) schedules are compared. Setup: 4 stages, 8 microbatches (2 DCs, 2 stages per DC). *Cross*-prefixed schedules are generated by the CO solver (Section 4.1). Slowdown is measured relative to the ZBV schedule at zero delay.

- The greedy CrossUDSub schedule matches the solver-based CrossUD in most delay regimes, highlighting its efficacy as a lightweight alternative.
- When delays are small, latency and bandwidth contribute equally to runtime. However, once bandwidth delay exceeds the forward time per chunk, it induces additional pipeline bubbles from queuing (Section 2.2).

5.2 Cross-DC PP vs. Cross-DC DP

We simulate iteration times for cross-DC PP and cross-DC DP approaches using the Llama 3 405B model [12] under various latency and bandwidth conditions (detailed in Appendix E).

Results in Figure 9 show that latency (ranging from 4–128 ms) has little impact on runtime in this scenario, as the per-stage forward time ($T_F \approx 109$ms) keeps the delay ratio low. However, bandwidth significantly affects performance. Cross-DC PP outperforms cross-DC DP by up to 3.05x when the cross-DC link bandwidth is limited to 4 GB/s. This gap narrows as bandwidth increases, becoming negligible beyond 1024 GB/s. Compared to the ideal single-DC case, cross-DC PP sees only a 1.3× slowdown at 64 GB/s. These results suggest that for large models with long per-stage computation time (T_F), bandwidth is the primary bottleneck in cross-DC communication. Under such conditions, cross-DC PP offers superior efficiency relative to DP, particularly when network resources are constrained.

6 CrossPipe Implementation

Schedules generated by CrossPipe can adapt to configuration changes, including PP size, hybrid parallelism setups,

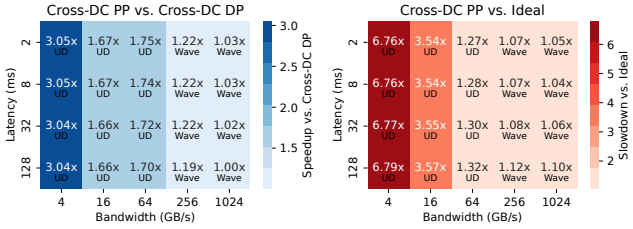


Figure 9: Simulation results comparing cross-DC PP and DP for Llama 3 405B training across two DCs. Left: Speedup of cross-DC PP over cross-DC DP. Right: Slowdown of cross-DC PP compared to an ideal single-DC setup. Labels indicate exact values and optimal schedule types per configuration.

and system parameters. In contrast, static PP modules in existing frameworks support only a limited, hard-coded range of schedules, making them difficult to adapt to and extend. Our implementation addresses these limitations through the CrossPipe module, which integrates seamlessly with the existing training framework. We use Megatron-LM as our base framework. The CrossPipe module is primarily implemented in Python, with components in C++ to enable latency and bandwidth injection (Section 6.4) for emulating cross-DC network conditions on a homogeneous cluster.

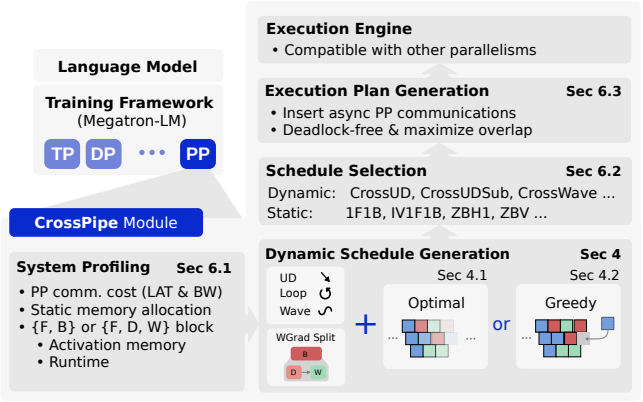


Figure 10: Components of the CrossPipe module.

An overview of our implementation is shown in Figure 10 . The module begins by collecting system parameters via lightweight benchmarks (Section 6.1). It then generates dynamic pipeline schedules (defining the order and timing of computation blocks) using either the constraint optimization solver (Section 4.1) or the greedy algorithm (Section 4.2). A schedule with the best simulation performance is selected (Section 6.2). Next, CrossPipe lowers it to a concrete execution plan by inserting and optimizing communication operations (Section 6.3). This decouples high-level scheduling logic from low-level execution, enabling dynamic, fine-grained control. The selected schedule can also be hot-swapped during training if better options are found.

6.1 System Profiling

CrossPipe collects critical metrics in a single iteration using lightweight profiling. These include runtime and memory usage of F, D, and W blocks, as well as communication delay parameters (α, β). We follow the model partitioning strategy of Llama 3 [12], treating embedding and output layers as transformer layers to ensure load balance across stages.

6.2 Schedule Selection

CrossPipe selects the schedule with the best estimated performance and supports hot-switching to adapt to changes during training.

- **Static schedules** are well-suited for single-DC training.
- **Dynamic schedules** are more suitable in cross-DC settings with high or varying communication costs. They also adapt better to available memory. Under rich memory budgets, dynamic schedules may increase the number of in-flight F blocks to improve efficiency.

6.3 Execution Plan

In this step, CrossPipe converts the selected pipeline schedule into an execution plan by inserting non-blocking communication operations. This plan is executed by the CrossPipe engine integrated into the training framework.

Communication Orchestration We use NCCL as the communication backend to leverage high-bandwidth intra-node interconnects and reduce inter-node data movement overheads. To decouple point-to-point communications in PP, we dedicate four GPU streams for each direction and role ({Send, Recv} × {Next, Prev}), avoiding interference and deadlocks. In both directions, Recv operations are reordered to align with the corresponding Sends, avoiding NCCL deadlocks. NCCL implements a rendezvous protocol for point-to-point communication, requiring both the sender and the receiver to synchronize before the transfer begins. To maximize communication overlap, we post Recv operations ahead of their corresponding Sends based on profiling estimates. This delay-aware arrangement improves overlap and is applied to both static and dynamic schedules (evaluated in Section 7).

6.4 Latency and Bandwidth Injection

We extend the PyTorch `ProcessGroupNCCL` C++ backend to inject latency and bandwidth delays in specific Send/Recv operations. This allows us to emulate various cross-DC network conditions (as described in Section 2.3) within a single cluster. Latency is injected on the receiver side of cross-DC communication, while bandwidth is throttled by running spinning kernels on the communication streams of both sender and receiver. Implementation details are provided in Appendix F.1, and validation results in Appendix F.2.

Name	Hidden Dim.	Int. Dim.	Att. Heads	KV Heads	Layers
M8	4096	14336	32	8	30+2
M70	8192	28672	64	8	62+2

Table 4: Hyperparameters of models used in the evaluation. Example: M70 is a model with Transformers layers of the same size as the ones in the Llama 3 70B model. The number of layers is reported as number of transformer layers + embedding & output layers.

7 Evaluation

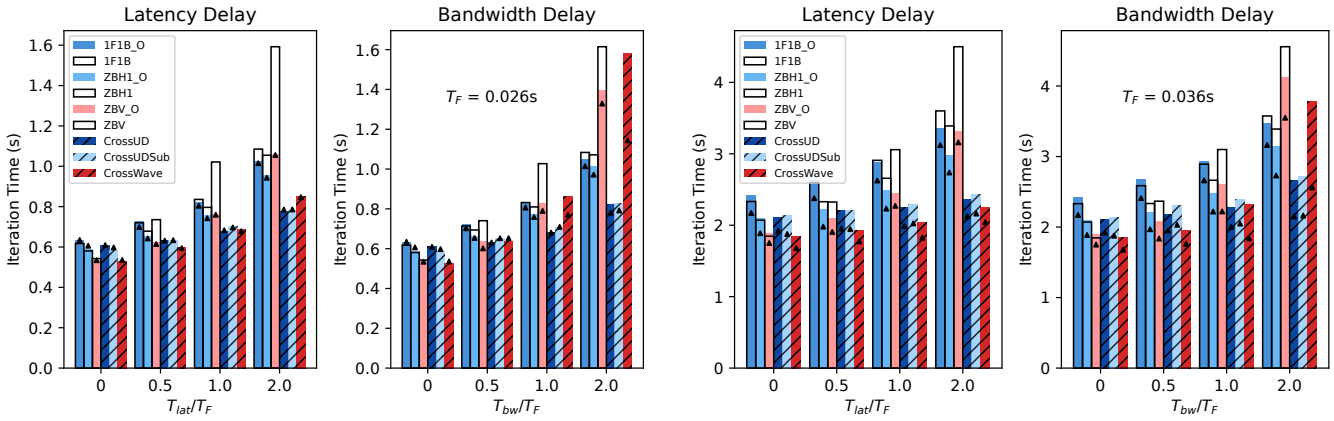
We conducted comprehensive evaluations on the Alps supercomputer to validate CrossPipe's performance and scalability. Each compute node is equipped with four GH200 Grace Hopper Superchips [41]. Each GH200 features 96 GB HBM3 memory integrated with the Hopper GPU die and 120 GB LPDDRX5 memory connected to the Grace CPU. The chips utilize a fully-connected topology with six NVLink 4.0 links between each GH200 pair, providing 200 Gb/s bandwidth per link per direction. Network connectivity is provided by HPE Cray Cassini-1 200 Gb/s NICs in a Dragonfly [25] topology using HPE Cray Slingshot-11 [8, 49] interconnect.

We use LLMs built up from the Llama-style Transformer layers, the hyperparameters of which are listed in Table 4. Each model configuration follows the naming convention `M{model_size}`, where the model_size indicates the size of Transformer layers it contains. For example, M70 uses the Transformer block of the same size as the one in the Llama 3 70B model. The models are then constructed by replicating and stacking identical Transformer layers, along with the vocabulary embedding layer and the output layer. We evaluate the following schedules: 1F1B, IV1F1B, ZBH1, ZBV, CrossUD, CrossUDSub, and CrossWave. The reason to exclude bidirectional (BD) and Loop schedules is explained in Section 5. By default, we set the microbatch size $b = 1$ and sequence length $s = 4096$. In these configurations, the message size for pipeline communication (activations/gradients) is approximately $32n_{DP}$ MB for M8 and $64n_{DP}$ MB for M70. FlashAttention [50] is enabled for higher throughput and less peak memory consumption. For each schedule, we measure 64 iterations and report the minimum value to minimize the network noise effects [17].

7.1 Impact of Latency and Bandwidth Delay

We evaluate the performance of schedules on the cluster with various emulated latency delay T_{lat} or bandwidth delay T_{bw} for each PP communication crossing the DC boundary, using the injection mechanism from Section 6.4. We conduct the experiments on both M8 and M70 models in a two-DC setting. The parallelism configurations are:

- **M8:** $n_{TP} = 2$, $n_{PP} = 4$, $n_{DP} = 1$, $GBS = 2n_{PP}n_{DP} = 8$.
- **M70:** $n_{TP} = 4$, $n_{PP} = 8$, $n_{DP} = 1$, $GBS = 2n_{PP}n_{DP} = 16$.



(a) Model M8

(b) Model M70

Figure 11: Evaluation of static and CrossPipe schedules under various emulated latency and bandwidth delay ratios. The runs of static schedules with communication arrangement optimization (Section 6.3) is marked with the suffix _O. The runtime prediction from the performance model (Section 3.5) is represented as (▲). Schedules are compared using the iteration time in seconds (lower is better). Dynamic schedules (Cross*) use the same peak memory budget as their corresponding static base schedule (e.g., CrossUD matches 1F1B).

We vary delay ratios $T_{lat}/T_F, T_{bw}/T_F \in \{0.0, 0.5, 1.0, 2.0\}$. T_F is defined as the maximum runtime of the per-microbatch forward computation among stages. The results are shown in Figure 11. For the static schedules, we vary two settings: with or without the delay-aware communication orchestration (Section 6.3). Runs with this optimization are marked with the suffix _O. This optimization reduces the impact of delayed receivers, aligning closely with the assumptions of our performance model (Section 3.5). This model accurately predicts the runtime of tested schedules in most configurations. Overall, the CrossPipe schedules show superior performance compared to static schedules, with a reduction in runtime of up to 33.6% (vs. original) or 21.9% (vs. optimized), achieved by the M70 model at $T_{bw}/T_F = 2$.

To ground these delay ratios in realistic scenarios, we take the M70 model as an example. We assume a practical DP size of $n_{DP} = 16$, with intra-DC DP communication fully overlapped. Under this setting, the message size per PP communication is calculated by $bsdn_{DP} * 2 = 1$ GB. Given a forward time of $T_F = 0.038$ s (Figure 11.b), the resulting injected latency delays range from 19 ms to 76 ms, and simulated bandwidth from 105 Gbps to 421 Gbps.

7.2 Further Bubble Reduction

Section 7.1 shows that block reordering helps to reduce PP runtime when the memory budget and global batch size (GBS) are strictly constrained. However, substantial bubble ratios persist under high communication costs. This observation necessitates examination of trade-offs among runtime, GBS and memory budget. Also, we take layer-wise activation recomputation [6] into consideration. Since the GBS varies among the settings, we use runtime per microbatch to compare schedules. We conduct the experiments on the M70 model in a 2-DC setting, with $n_{TP} = 4, n_{PP} = 8, n_{DP} = 1$, under delay combinations: $(T_{lat}/T_F, T_{bw}/T_F) \in \{(0,0), (0.25,2), (2,0.25), (2,2)\}$, across three configurations:

- **Case 1:** $GBS = 2n_{PP}n_{DP} = 16$, activation memory budget 1.0× (same as 1F1B), no recomputation.
- **Case 2:** $GBS = 32$, activation memory budget 1.0×, layer-wise recomputation.
- **Case 3:** $GBS = 32$, activation memory budget 2.0×, no recomputation.

The result is shown in Table 5. Static schedules cannot leverage extra memory budget to further reduce pipeline bubbles ('-' in case 3). Without delay, where $(T_{lat}/T_F, T_{bw}/T_F) = (0,0)$, the efficiency of the CrossPipe schedules is comparable to manually optimized static schedules. Increasing GBS amortizes bubbles in the warm-up and tear-down phases by extending the length of the steady phase which contains fewer bubbles. Under high latency (e.g.,$(2,0.25)$), increasing GBS and memory budget helps the CrossWave schedule to achieve pipeline efficiency (0.115 s per microbatch), matching the no delay case (0.118 s per microbatch). When the bandwidth delay dominates (e.g., $(2,0.25)$ or $(2,2)$), increasing both GBS and memory budget improves schedule efficiency by up to 1.33× (0.196 s to 0.147 s per microbatch for CrossUD). In general, the bandwidth delay is harder to mitigate than the latency delay under the same settings of GBS and memory budget. Layer-wise recomputation generally does not improve the runtime of dynamic schedules, as the recomputation during the backward pass negates its low memory benefits.

$\frac{T_{lat}}{T_F}$	$\frac{T_{bw}}{T_F}$	Case	Static			Dynamic (This Work)		
			1F1B	ZBH1	ZBV	UDSub	UD	Wave
0	0	1	0.151	0.133	**0.118**	0.137	0.137	0.119
		2	0.174	0.168	0.161	-	0.165	**0.157**
		3	-	-	-	0.121	0.118	**0.108**
0.25	0.25	1	0.168	0.15	0.148	0.149	0.142	**0.127**
		2	0.193	0.187	0.177	-	0.17	**0.159**
		3	-	-	-	0.123	0.123	**0.112**
0.25	2	1	0.241	0.23	0.315	0.181	**0.177**	0.25
		2	0.262	0.259	0.33	-	**0.185**	0.274
		3	-	-	-	**0.144**	0.15	0.235
2	0.25	1	0.242	0.229	0.314	0.16	0.153	**0.148**
		2	0.262	0.258	0.329	-	0.173	**0.163**
		3	-	-	-	0.127	0.124	**0.115**
2	2	1	0.321	0.309	0.473	0.198	**0.196**	0.256
		2	0.333	0.331	0.476	-	**0.196**	0.291
		3	-	-	-	**0.145**	0.147	0.245

Table 5: M70 model, 2-DC training. Runtime per microbatch of cross-DC PP solutions under various configurations (case 1-3) and communication delay. The best result of each configuration is shown in bold (lower is better). CrossUDSub, CrossUD and CrossWave schedules are abbreviated as UDSub, UD and Wave, respectively.

7.3 Scale to More DCs

We extend our analysis to 4 homogeneous interconnected DCs with uniform cross-DC link characteristics, using the same setup as the previous section (M70, $n_{TP} = 4$, $n_{PP} = 8$, $n_{DP} = 1$, now with 2 stages per DC). Table 6 confirms previous findings: CrossPipe is competitive without delay and outperforms static schedules in cross-DC scenarios. CrossWave excels CrossUD(Sub) at low delays but suffers under higher delays, especially under high bandwidth delays. With extra memory budget and GBS (Case 3), CrossUD schedule achieves 0.178 s per microbatch, only 22.8% slower than the corresponding 2-DC scenario at $(T_{lat}/T_F, T_{bw}/T_F) = (2,2)$. The larger bubble size in 4-DC training makes recomputation more effective here. Layer-wise recomputation with increased GBS (Case 2) is comparable to or outperforms the baseline without recomputation (Case 1) in most delay settings.

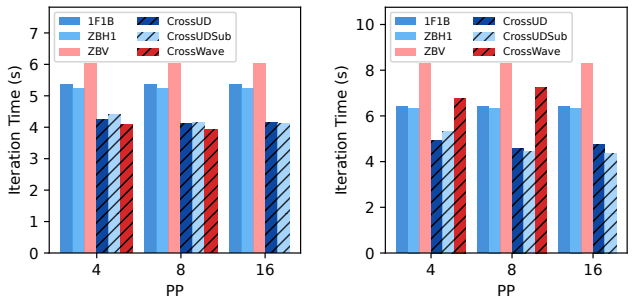
7.4 Trade-off of PP and DP

We analyze the choice of large PP size vs. large DP size in a 2-DC homogeneous training scenario. With a fixed number of compute nodes per DC and setting n_{TP} to the number of GPUs within each compute node, the product $n_{PP} \times n_{DP}$ equals the total node count of 2 DCs. Given the number of microbatches $n_{mb} = \varepsilon n_{PP}$, the global batch size $GBS = \varepsilon n_{PP} n_{DP}$ depends solely on the ratio ε. Then the GBS remains the same with various combinations of PP and DP.

We evaluated the trade-off on the M70 model across three PP configurations ($n_{PP} \in \{4, 8, 16\}$) with four combinations

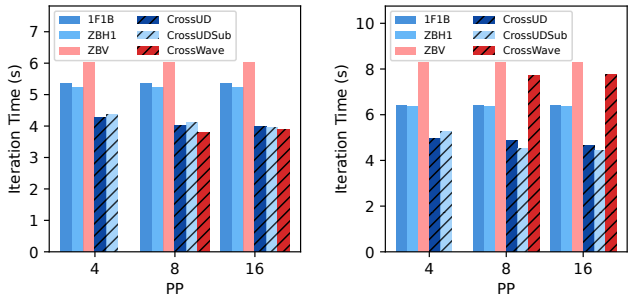
$\frac{T_{lat}}{T_F}$	$\frac{T_{bw}}{T_F}$	Case	Static			Dynamic (This Work)		
			1F1B	ZBH1	ZBV	UDSub	UD	Wave
0	0	1	0.149	0.133	**0.119**	0.138	0.138	0.122
		2	0.173	0.168	0.16	-	0.167	**0.157**
		3	-	-	-	0.123	0.119	**0.115**
0.25	0.25	1	0.177	0.158	0.161	0.155	0.148	**0.141**
		2	0.198	0.19	0.181	-	0.173	**0.163**
		3	-	-	-	0.126	0.123	**0.115**
0.25	2	1	0.269	0.249	0.339	**0.216**	0.217	0.286
		2	0.274	0.269	0.331	-	**0.198**	0.29
		3	-	-	-	**0.158**	0.162	0.262
2	0.25	1	0.268	0.248	0.337	0.2	**0.197**	0.214
		2	0.274	0.269	0.33	-	0.184	**0.177**
		3	-	-	-	**0.138**	**0.138**	0.139
2	2	1	0.359	0.338	0.512	**0.268**	0.271	0.339
		2	0.349	0.346	0.479	-	**0.213**	0.295
		3	-	-	-	**0.178**	**0.178**	0.264

Table 6: M70 model, 4-DC training. Runtime per microbatch of each PP schedule is listed. The rest of the configurations remain the same as Table 5.



(a) $T_{lat}/T_F = 1, T_{bw}/T_F = 0.25$; Static Mem. ($n_{PP} = 4$): 25%

(b) $T_{lat}/T_F = 0.125, T_{bw}/T_F = 2$; Static Mem. ($n_{PP} = 4$): 25%



(c) $T_{lat}/T_F = 1, T_{bw}/T_F = 0.25$; Static Mem. ($n_{PP} = 4$): 50%

(d) $T_{lat}/T_F = 0.125, T_{bw}/T_F = 2$; Static Mem. ($n_{PP} = 4$): 50%

Figure 12: Trade-off between PP and DP ($n_{PP} \times n_{DP}$ is fixed) in a 2-DC training. Using M70 with 16 nodes ($n_{TP} = 4$ fixed) and a fixed GBS. Each subplot shows iteration time vs. n_{PP}. Labels indicate delay ratios (T_{lat}/T_F, T_{bw}/T_F) and static memory usage. Memory in percentage shows the consumption of static memory at $n_{PP} = 4$ w.r.t. device limit.

of communication delay and memory settings. For communication delay $(T_{lat}/T_F, T_{bw}/T_F)$, we test high latency, low bandwidth $(1, 0.25)$ and low latency, high bandwidth $(0.125, 2)$ settings, with a static memory budget of 25% and 50%. This percentage is defined as the allocation of static memory (parameters, gradients, and optimizer states) at $n_{PP} = 4$ w.r.t. the device memory limit. Larger n_{PP} reduces the static memory per stage and the activation memory size per microbatch, so that it increases the activation memory budget and allows for more in-flight forward blocks. Also with larger n_{PP}, the communication volume is lower due to smaller DP, but at a higher frequency. While the absolute latency remains constant when increasing n_{PP}, the latency delay ratio T_{lat}/T_F increases due to reduced T_F as n_{PP} grows (less work per stage). $\varepsilon = 4$ to enable more in-flight blocks in each schedule to benefit from available memory. Since the GBS (total workload) and the number of compute nodes (total GPUs) are fixed, we use runtime to compare different configurations. As shown in Figure 12, these factors balance out. The schedule efficiency remains largely invariant to PP/DP configurations across scenarios. This balance likely stems from the flexibility of CrossPipe to fully utilize the available memory with sufficient GBS.

8 Discussion

8.1 Heterogeneous DCs

In this work, we primarily evaluated homogeneous GPUs and nodes across DCs due to cluster constraints. However, our approach can be extended to heterogeneous environments. If compute resources (e.g., GPU, network) differ significantly between DCs, our generated schedules may be suboptimal because faster nodes will finish computation earlier and create bubbles in the pipeline, and slower nodes may run out of memory since their device memory is usually more limited. A practical solution is to maintain homogeneity within each pipeline stage, which aligns with current DC practices where nodes within the same cabinet tend to be identical. The compute nodes with faster GPUs can be assigned more layers to balance the computation time across stages. Our formulation (Section 4.1) and greedy algorithm (Section 4.2) already support stage-specific parameters, naturally supporting this adjustment.

8.2 Network Dynamics and Fault Tolerance

Real-world cross-DC networks exhibit variability and are prone to failures. CrossPipe can employ several strategies to enhance robustness:

- Short-Term Variations (seconds or less): Section 7.2 shows that CrossPipe schedules can trade system resources (e.g., device memory, by allowing more in-flight microbatches) for efficiency. Conservative (higher) latency and/or (lower) bandwidth estimations can be used to generate schedules that tolerate small spikes in communication delay.
- Longer-Term Variations (minutes or more): Network conditions can shift due to traffic or routing. CrossPipe's flexible execution engine (Section 6) supports hot-switching of pipeline schedules. The system with CrossPipe enabled can periodically re-profile network conditions (Section 6.1) and generate new, tailored greedy schedules (Section 4.2) to adapt without interrupting training.
- Packet Loss/Link Errors: Transient network errors such as packet drops can be handled by mechanisms like Forward Error Correction (FEC) [14] or Selective Repeat [2], leading to spikes in communication delay as mentioned above.
- Node Failures: The failure of a compute node requires higher-level mechanisms beyond pipeline scheduling. Efficient checkpointing, such as asynchronous methods [35] and in-memory approaches [67], is necessary to save training state efficiently and recover from the last checkpoint with minimal progress loss.

9 Related Works

Pipeline Parallelism (PP): 1F1B in PipeDream [38], GPipe [19], DAPPLE [13], interleaved 1F1B [39], bidirectional pipeline from Chimera [30], BPipe [26] and MPress [69] for memory balancing, BFSPP for more DP communication overlap [28], Hanayo [34] for wave-like schedules, ooo backprop [42] and zero-bubble PP [43] for weight gradient splitting, AdaPipe [57] for co-optimizing layer distribution and recomputation, DHelix [62] for microbatch co-execution to overlap communication, DistMM [18] for multimodal model training. Sequence-level pipeline parallelism [32, 56].

Training on restricted networks with PP: Varuna [3] explores training on spot VMs with commodity networking. Bamboo [60] studies resilient training on preemptible instances. Oobleck [22] improves training resilience via pipeline templates. SWARM [47] proposes reliable training via temporary randomized pipelines. [68] studies device assignment in hybrid parallel training with geo-distributed nodes. CocktailSGD [63] combines multiple compression techniques to train models efficiently on low-bandwidth networks. FusionLLM [58] accelerates decentralized training via activation and gradient compression. DiLoCo [10] explores robust asynchronous training on poorly connected machines.

10 Conclusion

In this work, we first introduced a validated pipeline performance model that explicitly accounts for latency and bandwidth delays in cross-datacenter links. Using this model, we demonstrated that pipeline parallelism is often the superior approach for the parallelism dimension spanning across dat-

acenters, especially under constrained network conditions. Next, we leveraged the model to develop optimal and near-optimal algorithms for generating pipeline schedules that minimize cross-datacenter communication delays while adhering to memory constraints. Finally, we integrated these methods into a flexible execution engine featuring a two-layer abstraction (block scheduling and communication arrangement) that works seamlessly with existing training systems, such as Megatron-LM.

Our evaluation shows that CrossPipe effectively overcomes the challenges of cross-datacenter training. It reduces the training time by up to 33.6% compared to traditional pipeline schedules in a cross-DC setup, all while maintaining the same memory constraints. When memory constraints are relaxed, CrossPipe in a cross-DC setup is able to achieve a similar training time as a static ZBV schedule in a single-DC setup where there is almost no communication delay. CrossPipe thus offers improved scalability and resource utilization, making large-scale distributed training more feasible and efficient.

Acknowledgments

We are grateful to our shepherd and the anonymous reviewers for their insightful comments and constructive feedback. We thank the CSCS team for providing access to the Ault and Alps machines, as well as for their outstanding technical support. We are grateful to Siyuan Shen and Mikhail Khalilov for their valuable advice, and to Timo Schneider for his assistance with infrastructure at SPCL. We also acknowledge the Polish high-performance computing infrastructure PLGrid (HPC Center: ACK Cyfronet AGH) for providing computational resources and support.

References

[1] Reed Albergotti. Microsoft Azure CTO: US data centers will soon hit size limits. *Semafor*, October 2024. Technology.

[2] Miltiades Anagnostou and Emmanuel Protonotarios. Performance analysis of the selective repeat arq protocol. *IEEE Transactions on Communications*, 34(2):127–135, 2003.

[3] Sanjith Athlur, Nitika Saran, Muthian Sivathanu, Ramachandran Ramjee, and Nipun Kwatra. Varuna: Scalable, low-cost training of massive deep learning models, 2021.

[4] Tal Ben-Nun and Torsten Hoefler. Demystifying parallel and distributed deep learning: An in-depth concurrency analysis, 2018.

[5] Bloomberg. Tech firms are asking energy giant nextera for enough electricity to power miami, 2024.

[6] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. Training deep nets with sublinear memory cost, 2016.

[7] João da Silva. Google turns to nuclear to power AI data centres. *BBC News*, October 2024. Business.

[8] Daniele De Sensi, Salvatore Di Girolamo, Kim H McMahon, Duncan Roweth, and Torsten Hoefler. An in-depth analysis of the slingshot interconnect. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14. IEEE, 2020.

[9] DeepSeek-AI, Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, Damai Dai, Daya Guo, Dejian Yang, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Han Bao, Hanwei Xu, Haocheng Wang, Haowei Zhang, Honghui Ding, Huajian Xin, Huazuo Gao, Hui Li, Hui Qu, J. L. Cai, Jian Liang, Jianzhong Guo, Jiaqi Ni, Jiashi Li, Jiawei Wang, Jin Chen, Jingchang Chen, Jingyang Yuan, Junjie Qiu, Junlong Li, Junxiao Song, Kai Dong, Kai Hu, Kaige Gao, Kang Guan, Kexin Huang, Kuai Yu, Lean Wang, Lecong Zhang, Lei Xu, Leyi Xia, Liang Zhao, Litong Wang, Liyue Zhang, Meng Li, Miaojun Wang, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Mingming Li, Ning Tian, Panpan Huang, Peiyi Wang, Peng Zhang, Qiancheng Wang, Qihao Zhu, Qinyu Chen, Qiushi Du, R. J. Chen, R. L. Jin, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, Runxin Xu, Ruoyu Zhang, Ruyi Chen, S. S. Li, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shaoqing Wu, Shengfeng Ye, Shengfeng Ye, Shirong Ma, Shiyu Wang, Shuang Zhou, Shuiping Yu, Shunfeng Zhou, Shuting Pan, T. Wang, Tao Yun, Tian Pei, Tianyu Sun, W. L. Xiao, Wangding Zeng, Wanjia Zhao, Wei An, Wen Liu, Wenfeng Liang, Wenjun Gao, Wenqin Yu, Wentao Zhang, X. Q. Li, Xiangyue Jin, Xianzu Wang, Xiao Bi, Xiaodong Liu, Xiaohan Wang, Xiaojin Shen, Xiaokang Chen, Xiaokang Zhang, Xiaosha Chen, Xiaotao Nie, Xiaowen Sun, Xiaoxiang Wang, Xin Cheng, Xin Liu, Xin Xie, Xingchao Liu, Xingkai Yu, Xinnan Song, Xinxia Shan, Xinyi Zhou, Xinyu Yang, Xinyuan Li, Xuecheng Su, Xuheng Lin, Y. K. Li, Y. Q. Wang, Y. X. Wei, Y. X. Zhu, Yang Zhang, Yanhong Xu, Yanhong Xu, Yanping Huang, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Li, Yaohui Wang, Yi Yu, Yi Zheng, Yichao Zhang, Yifan Shi, Yiliang Xiong, Ying He, Ying Tang, Yishi Piao, Yisong Wang, Yixuan Tan, Yiyang Ma, Yiyuan Liu, Yongqiang Guo, Yu Wu, Yuan Ou, Yuchen Zhu, Yuduan Wang, Yue Gong, Yuheng Zou, Yujia He, Yukun Zha, Yunfan Xiong, Yunxian Ma, Yuting Yan, Yuxiang Luo, Yuxiang You, Yuxuan Liu, Yuyang Zhou, Z. F. Wu, Z. Z. Ren, Zehui Ren, Zhangli Sha, Zhe Fu, Zhean Xu, Zhen Huang, Zhen Zhang, Zhenda Xie, Zhengyan Zhang, Zhewen Hao, Zhibin Gou, Zhicheng Ma, Zhigang Yan, Zhihong Shao, Zhipeng Xu, Zhiyu Wu, Zhongyu Zhang, Zhuoshu Li, Zihui Gu, Zijia Zhu, Zijun Liu, Zilin Li, Ziwei Xie, Ziyang Song, Ziyi Gao, and Zizheng Pan. Deepseek-v3 technical report, 2025.

[10] Arthur Douillard, Qixuan Feng, Andrei A. Rusu, Rachita Chhaparia, Yani Donchev, Adhiguna Kuncoro, Marc'Aurelio Ranzato, Arthur Szlam, and Jiajun Shen. Diloco: Distributed low-communication training of language models, 2024.

[11] Jiangfei Duan, Shuo Zhang, Zerui Wang, Lijuan Jiang, Wenwen Qu, Qinghao Hu, Guoteng Wang, Qizhen Weng, Hang Yan, Xingcheng Zhang, et al. Efficient training of large language models on distributed infrastructures: A survey. *arXiv preprint arXiv:2407.20018*, 2024.

[12] Abhimanyu Dubey et al. The llama 3 herd of models, 2024.

[13] Shiqing Fan, Yi Rong, Chen Meng, Zongyan Cao, Siyu Wang, Zhen Zheng, Chuan Wu, Guoping Long, Jun Yang, Lixue Xia, Lansong Diao, Xiaoyong Liu, and Wei Lin. Dapple: A pipelined data parallel approach for training large models, 2020.

[14] R. W. Hamming. Error detecting and error correcting codes. *The Bell System Technical Journal*, 29(2):147–160, 1950.

[15] Zeyu Han, Chao Gao, Jinyang Liu, Sai Qian Zhang, et al. Parameter-efficient fine-tuning for large models: A com-

prehensive survey. *arXiv preprint arXiv:2403.14608*, 2024.

[16] Marius Hobbhahn, Lennart Heim, and Gökçe Aydos. Trends in machine learning hardware, 2023. Accessed: 2024-10-15.

[17] Torsten Hoefler, Timo Schneider, and Andrew Lumsdaine. The impact of network noise at large-scale communication performance. In *2009 IEEE International Symposium on Parallel Distributed Processing*, pages 1–8, 2009.

[18] Jun Huang, Zhen Zhang, Shuai Zheng, Feng Qin, and Yida Wang. DISTMM: Accelerating distributed multimodal model training. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 1157–1171, Santa Clara, CA, April 2024. USENIX Association.

[19] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, et al. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *Advances in neural information processing systems*, 32, 2019.

[20] Robert A Jacobs, Michael I Jordan, Steven J Nowlan, and Geoffrey E Hinton. Adaptive mixtures of local experts. *Neural computation*, 3(1):79–87, 1991.

[21] Sam Ade Jacobs, Masahiro Tanaka, Chengming Zhang, Minjia Zhang, Shuaiwen Leon Song, Samyam Rajbhandari, and Yuxiong He. Deepspeed ulysses: System optimizations for enabling training of extreme long sequence transformer models, 2023.

[22] Insu Jang, Zhenning Yang, Zhen Zhang, Xin Jin, and Mosharaf Chowdhury. Oobleck: Resilient distributed training of large models using pipeline templates. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 382–395. ACM, October 2023.

[23] Albert Q. Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, Lélio Renard Lavaud, Marie-Anne Lachaux, Pierre Stock, Teven Le Scao, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. Mistral 7b, 2023.

[24] Albert Q. Jiang, Alexandre Sablayrolles, Antoine Roux, Arthur Mensch, Blanche Savary, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Emma Bou Hanna, Florian Bressand, Gianna Lengyel, Guillaume Bour, Guillaume Lample, Lélio Renard Lavaud, Lucile Saulnier, Marie-Anne Lachaux, Pierre Stock, Sandeep Subramanian, Sophia Yang, Szymon Antoniak, Teven Le Scao, Théophile Gervet, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. Mixtral of experts, 2024.

[25] John Kim, Wiliam J Dally, Steve Scott, and Dennis Abts. Technology-driven, highly-scalable dragonfly topology. *ACM SIGARCH Computer Architecture News*, 36(3):77–88, 2008.

[26] Taebum Kim, Hyoungjoo Kim, Gyeong-In Yu, and Byung-Gon Chun. BPipe: Memory-balanced pipeline parallelism for training large language models. In Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett, editors, *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pages 16639–16653. PMLR, 23–29 Jul 2023.

[27] Vijay Korthikanti, Jared Casper, Sangkug Lym, Lawrence McAfee, Michael Andersch, Mohammad Shoeybi, and Bryan Catanzaro. Reducing activation recomputation in large transformer models, 2022.

[28] Joel Lamy-Poirier. Breadth-first pipeline parallelism, 2023.

[29] Junyi Li, Tianyi Tang, Wayne Xin Zhao, Jian-Yun Nie, and Ji-Rong Wen. Pre-trained language models for text generation: A survey. *ACM Computing Surveys*, 56(9):1–39, 2024.

[30] Shigang Li and Torsten Hoefler. Chimera: efficiently training large-scale neural networks with bidirectional pipelines. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14, 2021.

[31] Yuanchun Li, Hao Wen, Weijun Wang, Xiangyu Li, Yizhen Yuan, Guohong Liu, Jiacheng Liu, Wenxing Xu, Xiang Wang, Yi Sun, et al. Personal llm agents: Insights and survey about the capability, efficiency and security. *arXiv preprint arXiv:2401.05459*, 2024.

[32] Zhuohan Li, Siyuan Zhuang, Shiyuan Guo, Danyang Zhuo, Hao Zhang, Dawn Song, and Ion Stoica. Terapipe: Token-level pipeline parallelism for training large-scale language models. In Marina Meila and Tong Zhang, editors, *Proceedings of the 38th International Conference on Machine Learning*, volume 139 of *Proceedings of Machine Learning Research*, pages 6543–6552. PMLR, 18–24 Jul 2021.

[33] Hao Liu, Matei Zaharia, and Pieter Abbeel. Ring attention with blockwise transformers for near-infinite context, 2023.

[34] Ziming Liu, Shenggan Cheng, Haotian Zhou, and Yang You. Hanayo: Harnessing wave-like pipeline parallelism for enhanced large model training efficiency. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–13, 2023.

[35] Jayashree Mohan, Amar Phanishayee, and Vijay Chidambaram. {CheckFreq}: Frequent,{Fine-Grained}{DNN} checkpointing. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 203–216, 2021.

[36] Andrew Moseman. Amazon vies for nuclear-powered data center: The deal has become a flash point over energy fairness. *IEEE Spectrum*, August 2024.

[37] MPI Forum. *MPI: A Message-Passing Interface Standard Version 3.1*, 2015.

[38] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R Devanur, Gregory R Ganger, Phillip B Gibbons, and Matei Zaharia. Pipedream: Generalized pipeline parallelism for dnn training. In *Proceedings of the 27th ACM symposium on operating systems principles*, pages 1–15, 2019.

[39] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, et al. Efficient large-scale language model training on gpu clusters using megatron-lm. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15, 2021.

[40] NVIDIA. Nvidia dgx-1 with tesla v100 system architecture, 2017.

[41] NVIDIA. Nvidia grace hopper superchip architecture. Whitepaper, NVIDIA Corporation, 2024.

[42] Hyungjun Oh, Junyeol Lee, Hyeongju Kim, and Jiwon Seo. Out-of-order backprop: An effective scheduling technique for deep learning. In *Proceedings of the Seventeenth European Conference on Computer Systems*, pages 435–452, 2022.

[43] Penghui Qi, Xinyi Wan, Guangxing Huang, and Min Lin. Zero bubble pipeline parallelism, 2023.

[44] Qwen, :, An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jingren Zhou, Junyang Lin, Kai Dang, Keming Lu, Keqin Bao, Kexin Yang, Le Yu, Mei Li, Mingfeng Xue, Pei Zhang, Qin Zhu, Rui Men, Runji Lin, Tianhao Li, Tianyi Tang, Tingyu Xia, Xingzhang Ren, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yu Wan, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, and Zihan Qiu. Qwen2.5 technical report, 2025.

[45] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. Zero: Memory optimizations toward training trillion parameter models. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, November 2020.

[46] Reuters. Microsoft deal signals booming demand from data centers to power AI. *Reuters*, September 2024. Energy, Grid & Infrastructure, Nuclear.

[47] Max Ryabinin, Tim Dettmers, Michael Diskin, and Alexander Borzunov. Swarm parallelism: Training large models can be surprisingly communication-efficient, 2023.

[48] Jaime Sevilla and Edu Roldán. Training compute of frontier ai models grows by 4-5x per year, 2024. Accessed: 2024-10-15.

[49] Kawthar Shafie Khorassani, Chen Chun Chen, Bharath Ramesh, Aamir Shafi, Hari Subramoni, and Dhabaleswar Panda. High performance mpi over the slingshot interconnect: Early experiences. In *Practice and Experience in Advanced Research Computing*, pages 1–7. 2022.

[50] Jay Shah, Ganesh Bikshandi, Ying Zhang, Vijay Thakkar, Pradeep Ramani, and Tri Dao. Flashattention-3: Fast and accurate attention with asynchrony and low-precision, 2024.

[51] Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer, 2017.

[52] Siyuan Shen, Langwen Huang, Marcin Chrapek, Timo Schneider, Jai Dayal, Manisha Gajbe, Robert Wisniewski, and Torsten Hoefler. Llamp: Assessing network latency tolerance of hpc applications with linear programming, 2024.

[53] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism, 2020.

[54] Foteini Strati, Paul Elvinger, Tolga Kerimoglu, and Ana Klimovic. Ml training with cloud gpu shortages: Is cross-region the answer? In *Proceedings of the 4th Workshop on Machine Learning and Systems*, EuroMLSys '24, page 107–116, New York, NY, USA, 2024. Association for Computing Machinery.

[55] Foteini Strati, Paul Elvinger, Tolga Kerimoglu, and Ana Klimovic. Ml training with cloud gpu shortages: Is cross-region the answer? In *Proceedings of the 4th Workshop on Machine Learning and Systems*, pages 107–116, 2024.

[56] Ao Sun, Weilin Zhao, Xu Han, Cheng Yang, Xinrong Zhang, Zhiyuan Liu, Chuan Shi, and Maosong Sun. Seq1f1b: Efficient sequence-level pipeline parallelism for large language model training, 2024.

[57] Zhenbo Sun, Huanqi Cao, Yuanwei Wang, Guanyu Feng, Shengqi Chen, Haojie Wang, and Wenguang Chen. Adapipe: Optimizing pipeline parallelism with adaptive recomputation and partitioning. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS '24, page 86–100, New York, NY, USA, 2024. Association for Computing Machinery.

[58] Zhenheng Tang, Xueze Kang, Yiming Yin, Xinglin Pan, Yuxin Wang, Xin He, Qiang Wang, Rongfei Zeng, Kaiyong Zhao, Shaohuai Shi, Amelie Chi Zhou, Bo Li, Bingsheng He, and Xiaowen Chu. Fusionllm: A decentralized llm training system on geo-distributed gpus with adaptive compression, 2024.

[59] Arun James Thirunavukarasu, Darren Shu Jeng Ting, Kabilan Elangovan, Laura Gutierrez, Ting Fang Tan, and Daniel Shu Wei Ting. Large language models in medicine. *Nature medicine*, 29(8):1930–1940, 2023.

[60] John Thorpe, Pengzhan Zhao, Jonathan Eyolfson, Yifan Qiao, Zhihao Jia, Minjia Zhang, Ravi Netravali, and Guoqing Harry Xu. Bamboo: Making preemptible instances resilient for affordable training of large dnns, 2022.

[61] A Vaswani. Attention is all you need. *Advances in Neural Information Processing Systems*, 2017.

[62] Haiquan Wang, Chaoyi Ruan, Jia He, Jiaqi Ruan, Chengjie Tang, Xiaosong Ma, and Cheng Li. Hiding communication cost in distributed llm training via micro-batch co-execution, 2024.

[63] Jue Wang, Yucheng Lu, Binhang Yuan, Beidi Chen, Percy Liang, Christopher De Sa, Christopher Re, and Ce Zhang. CocktailSGD: Fine-tuning foundation models over 500Mbps networks. In Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett, editors, *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pages 36058–36076. PMLR, 23–29 Jul 2023.

[64] Shibo Wang, Jinliang Wei, Amit Sabne, Andy Davis, Berkin Ilbeyi, Blake Hechtman, Dehao Chen, Karthik Srinivasa Murthy, Marcello Maggioni, Qiao Zhang, Sameer Kumar, Tongfei Guo, Yuanzhong Xu, and Zongwei Zhou. Overlap communication with dependent computation via decomposition in large deep learning models. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, ASPLOS 2023, page 93–106, New York, NY, USA, 2022. Association for Computing Machinery.

[65] Weiyang Wang, Manya Ghobadi, Kayvon Shakeri, Ying Zhang, and Naader Hasani. Rail-only: A low-cost high-performance network for training llms with trillion parameters, 2024.

[66] Yufei Wang, Wanjun Zhong, Liangyou Li, Fei Mi, Xingshan Zeng, Wenyong Huang, Lifeng Shang, Xin Jiang, and Qun Liu. Aligning large language models with human: A survey. *arXiv preprint arXiv:2307.12966*, 2023.

[67] Zhuang Wang, Zhen Jia, Shuai Zheng, Zhen Zhang, Xinwei Fu, T. S. Eugene Ng, and Yida Wang. Gemini: Fast failure recovery in distributed training with in-memory checkpoints. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 364–381, New York, NY, USA, 2023. Association for Computing Machinery.

[68] Binhang Yuan, Yongjun He, Jared Quincy Davis, Tianyi Zhang, Tri Dao, Beidi Chen, Percy Liang, Christopher Re, and Ce Zhang. Decentralized training of foundation models in heterogeneous environments, 2023.

[69] Quan Zhou, Haiquan Wang, Xiaoyan Yu, Cheng Li, Youhui Bai, Feng Yan, and Yinlong Xu. Mpress: Democratizing billion-scale model training on multi-gpu servers via memory-saving inter-operator parallelism. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 556–569, 2023.

A Runtime Definition

In cross-DC training, due to the relatively high communication cost, the stage 0 usually finishes last among all the stages. So we adopt the most used definition of pipeline runtime as the following: the duration from the start of the first forward block on PP stage 0 till the completion of the last block on any PP rank (or the final DP communication on any rank). This definition aligns with frameworks that apply additional synchronization for global gradient norm computation (for clipping) and numerical anomaly detection (NANs/INFs) in mixed-precision training.

B Weight Gradient Separation

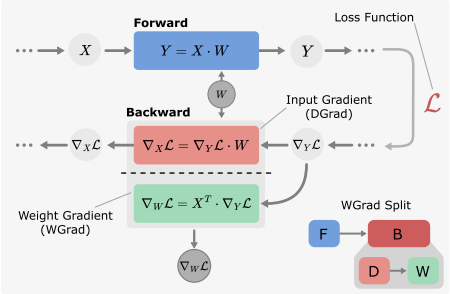


Figure 13: Separating input gradient computation (DGrad, or D) and weight gradient computation (WGrad, or W) in a linear layer.

Figure 13 illustrates how each B block can be further divided into two parts: input data gradient computation (DGrad, or D) and weight gradient computation (WGrad, or W) [43].

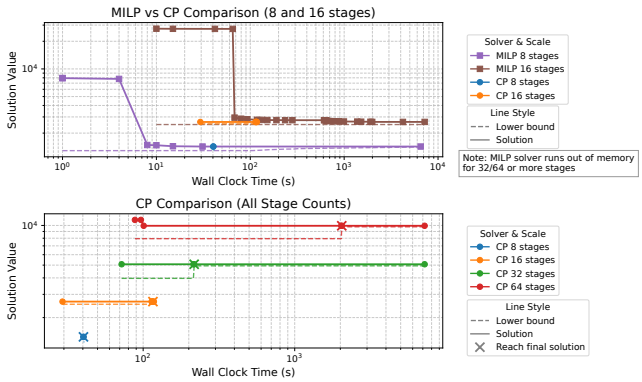


Figure 14: Scalability of MILP and CP solvers finding optimal pipeline schedules. Results show time-to-solution for varying pipeline stage counts (n_{PP}) with a fixed microbatch factor ($n_{mb} = 2n_{PP}$). Markers indicate the moments when the solver discovers an improved solution.

C Bubble Strides in Schedules

Figure 15 further illustrates the existence of *bubble strides*. We show schedules with 16 stages and 2 DCs for the static schedules that we compared in our work (1F1B, IV1F1B, ZBH1, and ZBV). The latency delay on the cross-DC link is set to $1.5 \cdot T_F$. Despite this relatively small latency (compared to the size of the pipeline blocks), the delay accumulates throughout the schedule rather than being absorbed or mitigated. This accumulation underscores the need for adaptive scheduling.

D Optimal Scheduling

D.1 Scalability of MILP and CO Solvers

We compare the scalability of both MILP and CO solvers, Gurobi (12.0.0) and CPLEX (22.1.1) respectively, both among the best solvers in their fields. The MILP formulation is from [43] and the CO formulation from Section 4.1. We conduct the experiments on a machine with AMD EPYC 7742 @ 2.25GHz CPU (128 physical cores), 256GB RAM. Each solver is configured to 256 worker threads, with a time limit of 7200 seconds. We use Gurobi Optimizer with the `NodefileStart=0.5` parameter to handle potential memory limitations.The solver is early-terminated if the relative gap between the objective value of the best integer solution found so far (ObjVal) and the best objective bound (ObjBound) is below 1%, defined as $\frac{|ObjBound - ObjVal|}{|ObjVal|} \leq 0.01$. The time to solution of both solvers are shown in Figure 14. The CO solver scales better than the MILP solver for pipeline schedule generation, ideally because of its specialized optimization w.r.t. job scheduling problems. The MILP solver runs out of memory even for runs of 32/64 stages and converges slower than CO solver. On the other hand, CO solver shows tractable performance when problem size scales up, and is able to find a good feasible solution in a reasonable amount of time while spending most of the time searching for a tighter bound.

E Comparing Cross-DC PP and Cross-DC DP

We use a 2 DC setup for the Llama 3 405B model training. The training configuration is taken from Llama 3 [12]: $n_{TP} = 8$, $n_{PP} = 16$, $n_{DP} = 64$, $seq_len = 8192$, $GBS = 2n_{PP}n_{DP}$.

We estimate the computation time of each layer by the following equation:

$$T_{layer} = \frac{C_{layer}}{P_{GPU} \times n_{TP}}$$

where C_{layer} is the FLOPs count of a transformer layer in the 405B model, P_{GPU} is the practical BF16 performance of GPU (500 TFLOPs per second is used in the analysis, considering the TP communication overhead). Assuming the GPUs

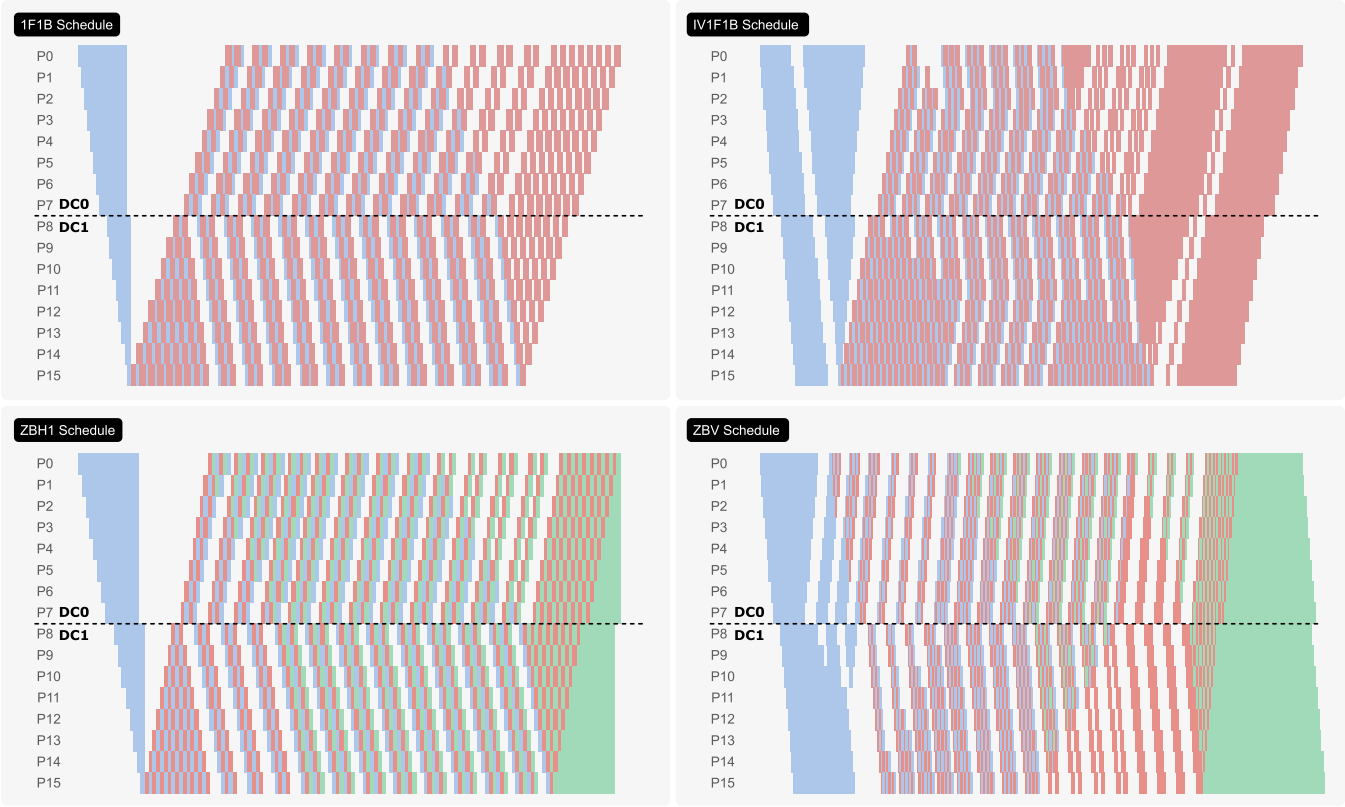


Figure 15: Illustration of bubble strides in various static pipeline schedules (1F1B, IV1F1B, ZBH1, and ZBV). Setup: 16 stages, 2DCs (8 stages per DC), $n_{mb} = 32$.

are equally distributed in 2 DCs. In cross-DC PP, we apply solver-based schedules (CrossUD or CrossWave, whichever is better for the given delay). In cross-DC DP, we use the ZBV schedule and assume that the extra cross-DC communication is not overlapped. For Allreduce (or Reduce-Scatter + Allgather) DP communications, we apply the bandwidth-optimal Ring-algorithm. The cross-DC DP communication cost can be estimated as $2 \times (\alpha + 2 \times \frac{N}{2} \times \beta) = 2\alpha + 2N\beta$ which accounts for two rounds of communication between two DCs and half of the parameters/gradients are sent/received. N is the number of model parameters, the extra factor 2 in the bandwidth term comes from the size of the BF16 datatype (2 bytes/param).

F Latency and Bandwidth Delay Injection

F.1 Implementation Details

Figure 16 illustrates our mechanism for injecting latency and bandwidth delay, used to emulate cross-DC network behavior in a controlled setting, without modifying core NCCL behavior or requiring network hardware manipulation. GPU kernel execution, including that of NCCL communication kernels, is asynchronous to the host CPU, making precise delay injection non-trivial.

- **Bandwidth Delay**: To simulate limited bandwidth, we inject additional delay into the communication path by occupying the communication stream after each NCCL send/receive. Specifically, a spin kernel is posted to the same stream immediately after the NCCL call, both on the sender and receiver sides. The spin kernel duration is computed to match the target delay. This effectively stalls further communication or computation that shares the stream, emulating a fully utilized link.
- **Latency Delay**: We extend PyTorch NCCL backend by adding a new method, `handle.wait_with_lat_delay()`, which is invoked on the receiver side during each blocking wait on NCCL communication. It is similar to the original `wait()` method but adds a controlled amount of host-side spinning to delay the launch of subsequent computation kernels.

The injection process contains the following steps: ❶ Sender and receiver post matching asynchronous point-to-point operations.❷ Optional compute kernels may be posted to overlap with communication. ❸ The receiver calls `wait_with_lat_delay` to synchronize the computation stream with the communication stream. ❹ A CUDA event measures the elapsed time from the completion of the communication kernel to the current time on the computation stream. ❺ The host then synchronizes with the computation stream

to retrieve the elapsed time and calculates the remaining delay (if any) to inject. ❻ A spinning kernel is posted to the computation stream for the remaining delay. This mechanism ensures that delay is not introduced when communication has already completed, and only partially injected when communication is partially complete (e.g., due to overlap with computation), thereby preserving the correct performance behavior of latency-hiding schedules.

This injection method is specifically designed for the CrossPipe implementation, where communication in four directions is split across four concurrent streams. Injecting delays into collective communications is more complex. While latency delays can be introduced within communication libraries [52], simulating bandwidth delays may require additional configurations at the network switch level.

In our communication model, we account only for latency and bandwidth delays along the critical path. We assume that the transmission of control messages, such as "ready to send" signals in a rendezvous protocol, is removed from the critical path. This assumption is crucial for maintaining performance, especially under high-latency conditions.

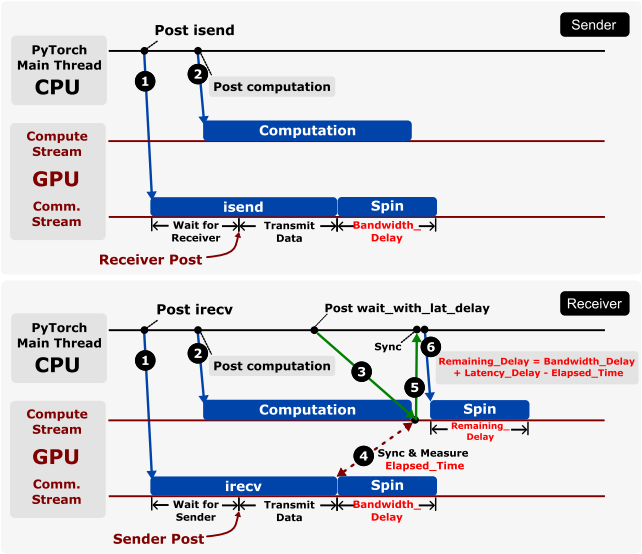


Figure 16: Mechanism for latency and bandwidth delay injection. Top: sender side. Bottom: receiver side.

F.2 Validation

To validate the accuracy of our delay injection methods (Section 6.4), we conduct experiments on a single GH200 node using 4 GB messages.

For latency tests, we use ping-pong communication between two processes: each round consists of a sender transmitting a message, waiting for a reply, and measuring the round-trip time. We divide the round-trip latency by two and compare it against the baseline (no injection). For bandwidth

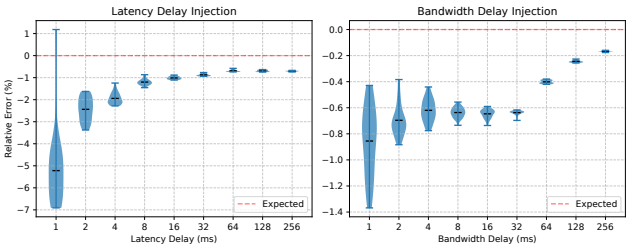


Figure 17: Validation of latency and bandwidth delay injection methods described in Section 6.4.

tests, the sender transmits multiple large messages back-to-back, while the receiver posts matching receives. We compute the average time per message and compare it to the baseline.

For each delay setting, we compute the relative error between the observed and expected delays. Specifically, we perform multiple iterations per setting, discard outliers, and report the percentage deviation from the expected delay. These measurements are then visualized using violin plots.

Results in Figure 17 demonstrate that our injection methods accurately reflect the communication model described in Section 2.2. Minor deviations primarily stem from host CPU synchronization and slight underestimation of GPU clock rate by `cudaGetDeviceProperties` in the spinning kernel. The validation shows the injected delay closely matches the target delay, confirming the mechanism's suitability for emulating cross-DC network conditions.

Unveiling Compiler Faults via Attribute-Guided Compilation Space Exploration

Jiangchang Wu, Yibiao Yang*, Maolin Sun, Yuming Zhou
State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China

Abstract

Compiler testing is critically important, as compilers serve as the foundational infrastructure in system software development. A comprehensive exploration of the compilation space is essential for uncovering bugs in compilers. Existing methods primarily involve the utilization of various compilation options alongside test programs as inputs for stress-testing compilers. However, these compilation options are typically applied uniformly across all program elements-such as functions and variables–by default, limiting the ability to thoroughly explore the compilation space. In programming languages like C and C++, attributes such as the `__attribute__((always_inline))` directive provide a mechanism for programmers to specify additional information for specific code elements to the compiler. These attributes allow for precise control over the compilation process, such as enforcing constraints and customizing optimization passes for particular elements. This flexibility in specifying attributes offers opportunities to investigate previously unexamined areas within compilers. Unfortunately, few studies have leveraged attributes for compiler testing. To this end, we propose ATLAS, an attribute-guided approach that strategically inserts attributes into test programs to facilitate a more thorough exploration of the compilation space. Our key insight is that attributes specified for individual program elements can provide a more flexible means of exploring the compilation space. Our extensive experiments on GCC and LLVM demonstrate the superiority of ATLAS over baseline testing techniques that do not employ attributes, particularly in terms of bug detection and code coverage. Furthermore, ATLAS has led to the discovery of 73 unique bugs in GCC and LLVM, 58 of which have already been confirmed or fixed, showcasing its practical utility.

*Corresponding author

1 Introduction

Compilers are fundamental to the development of system software, as nearly all system software relies on them for construction [2]. However, the complexity and extensive scope of compilers make them prone to errors. Bugs hidden within compilers can have far-reaching consequences, affecting system correctness, security, reliability, performance, and maintenance. Therefore, ensuring the reliability and correctness of compilers is of critical importance.

To ensure the quality of compilers, various compiler testing methods have been proposed [5]. Csmith [52], a notable tool in this domain, randomly generates diverse test programs based on predefined rules or grammar. More recently, YARPGen [32] and its successor YARPGen v.2 [33] have been proposed, focusing on generating test programs specific to the compiler's loop optimization strategies. Both Csmith and YARPGen are capable of generating a large number of test programs to stress test compilers. During the testing process, different combinations of compilation options are employed which aim to explore the compilation space, thereby exposing potential bugs in compilers. EMI [25], a representative metamorphic testing technique [9], mutates existing programs by removing or inserting random code while preserving the original input/output behaviors. EMI also applies various compilation options to compile the mutated programs which also intend to trigger unexplored behaviors of compilers. This line of research has yielded significant results, uncovering a considerable number of bugs in mature and well-established compilers [26, 46].

However, existing compiler testing methods primarily depend on the use of different compilation options to explore compilation space. These options are typically applied to the entire program, which limits their capabilities to explore the deeper states of compilers. In practice, **attributes** in programming languages such as C and C++ provide a valuable mechanism for programmers to specify additional information to the compiler to enforce constraints, optimize specific sections of code, or facilitate targeted code generation. The additional

information can include the intended use and behavior of variables, functions, and types [13, 19, 20], which can influence various stages of the compilation process, from optimization to code generation, and even runtime behavior. For instance, the `noinline` attribute instructs the compiler not to inline a function during optimization [13]. The `aligned` attribute specifies the necessary alignment for data structures during code generation [20]. Similarly, the `noreturn` attribute indicates that a function does not return, allowing the compiler to optimize control flow and potentially eliminate unnecessary cleanup code [13].

Approach In this paper, we propose ATLAS [1], an attribute-guided comprehensive compiler testing technique. The key insight underlying ATLAS is that attributes provide flexible control over the compilation process for individual elements within test programs, thereby facilitating a more thorough exploration of compiler behavior. Specifically, for a given test program, ATLAS first identifies the program elements, i.e., the functions, variables, and statements, that potential attributes can be specified. Subsequently, ATLAS randomly inserts appropriate attributes for those elements, resulting in mutated test programs with the inserted attributes. These mutated test programs will be then fed to compilers for subsequent testing. By strategically combining various compilation options with the attributes inserted programs, ATLAS can thoroughly explore the functionalities of the compiler, thereby improving the effectiveness in exposing compiler faults.

To evaluate the performance of ATLAS, we conduct an extensive study based on GCC and LLVM. In the study, we compared ATLAS with those baseline techniques without inserting any attributes. Our experimental results show that ATLAS outperforms these methods in terms of the number of bugs found within the same testing period. In particular, ATLAS can detect 109.1% more bugs than the baseline approaches on average. Moreover, ATLAS achieves higher code coverage of GCC and LLVM triggered by the mutated test programs than the baseline approaches. Furthermore, we applied ATLAS to test the trunk version of both GCC and LLVM. In total, ATLAS has successfully detected 73 unique bugs, 58 of which have been confirmed or fixed by developers, demonstrating the effectiveness of ATLAS.

Contributions. We make the following major contributions:

- **Novelty:** We introduce an attribute-guided compilation space exploration approach for comprehensive compiler testing. Specifically, we propose to strategically insert appropriate attributes for program elements to systematically explore the compilation space for testing compilers.
- **Prototype:** We have developed a prototype implementation of our proposed approach, named ATLAS, which leverages various attributes to more effectively explore the compilation space of compilers.
- **Usefulness:** We conduct an extensive study to evaluate ATLAS on two most widely used C compilers (i.e., GCC and LLVM), demonstrating its effectiveness. In particular, ATLAS has led to 73 unique bugs for GCC and LLVM, 58 of which have been confirmed or fixed by developers.

[1] ATLAS: **At**tribute-Guided Compi**la**tion **S**pace Exploration

Paper Organization. The rest of this paper is structured as follows. Section 2 describes the background and our motivation. Section 3 illustrates our proposed approach, ATLAS. Section 4 presents our extensive evaluation of ATLAS. Section 5 is the discussion. Section 6 is the related works. Section 7 is our conclusion.

2 Background and Motivation

This section first presents a brief introduction to compiler optimizations and compiler attributes. Then, we motivate our technique using a real bug example.

2.1 Compiler Optimizations

Modern compilers, such as GCC and Clang, use optimization techniques to reduce compilation time, minimize binary size, and improve runtime performance [23]. These compilers typically provide multiple standard optimization levels, such as -O1, -O2, -Os, and -O3, each comprising a set of optimization options. For example, the -O1 optimization level in GCC includes options like `-fdce`, which eliminates unreachable code within the programs, and `-fmerge-constants`, which consolidates duplicate constants within the programs. These optimization options can be applied as part of optimization levels or individually specified in the compilation setting [18]. Regardless of whether they are enabled by default within a compilation level or specified individually, these strategies usually apply to the entire program rather than to specific code elements (e.g., a single function).

Over the past two decades, substantial research efforts have been devoted to the field of compiler testing [6, 12, 25, 27, 30, 32, 33, 46, 53]. Various techniques have been proposed that primarily focus on generating diverse test programs and then utilizing different optimization levels or employing various combinations of optimization options to perform differential testing of compilers. However, these approaches overlook the application of attributes, which allow for the flexible specification of different compilation strategies for individual code elements. As a result, they miss the opportunity to explore the extensive compilation space thoroughly.

2.2 Compiler Attributes

Attributes provide a mechanism for passing additional details about code generation and code optimization to the compiler [1]. For instance, attributes can control inlining decisions, specify memory alignment requirements, or impose specific

constraints on variable usage [13, 20]. Unlike compiler options, attributes operate at a finer granularity, allowing precise control over specific program elements, such as individual functions or variables. Common types of attributes include function attributes [13], variable attributes [20], and type attributes [19], covering functionalities such as optimization control, memory alignment, warning suppression, and specifying function characteristics. For instance, the `noreturn` attribute indicates that a function does not return, while the `pure` attribute specifies that a function has no side effects [13]. Such fine-grained functionality cannot be achieved through compilation options alone.

During compilation, the compiler identifies and parses these attributes in the syntax analysis phase. In the semantic analysis phase, it adjusts its behavior based on the information provided by the attributes. For instance, recognizing a `noreturn` attribute allows the compiler to handle functions that do not return appropriately. In the optimization phase, attributes can guide the compiler to perform specific optimizations or disable certain optimizations. Ultimately, in the code generation phase, the compiler produces the target code following these attributes. By applying different attributes to finer-grained program elements, such as functions or variables, the compiler can trigger additional behaviors, maybe further exploring previously unexamined areas within compilers.

```
struct S {
  int i;
};
inline struct S bar() {
  struct S s = { 0 };
  return s;
}
__attribute__((optimize("-ftree-loop-distribution")))
void foo() {
  bar();
}
```

Figure 1: GCC bug 43234. The optimization attribute `optimize("-ftree-loop-distribution")` specified for the function `foo()` triggers a GCC bug.

As depicted in Figure 1, the highlighted code snippet demonstrates the use of an optimization attribute applied to the `foo()` function. The `-ftree-loop-distribution` optimization option, which can improve cache performance on big loop bodies and allow further loop optimizations [18], is typically enabled at the `-O3` optimization level. When used in conjunction with `attribute`, this optimization can be selectively applied to individual functions. In this case, when GCC compiled the program with the optimization attribute applied to the `foo()` function, a crash occurred. Interestingly, the original program does not have an optimization attribute on `foo()`, and compiling the original program with the optimization option directly did not result in a compiler crash. This approach allows for finer-grained compilation control over individual functions within the program.

Proper use of attributes is a vital strategy for improving both code quality and compiler efficiency. Through the judicious application of attributes, programmers can achieve substantial gains in performance, and clarity of code intent, making attributes an essential tool for enhancing the quality of code and the performance of the compiler.

```
int **a;
__attribute__((always_inline)) int *foo(void) {
  int b[1];
  (void) b;
}
__attribute__((no_sanitize_address)) static char
bar(void) {
  *a = foo();
}
void baz(void) {
  bar();
}
```

Figure 2: GCC bug 114956, which is triggered by a test program generated by ATLAS. When this generated program is compiled with the flags `-O2 -fsanitize=address,null`, it results in a crash.

2.3 Motivation

To illustrate the motivation behind our work, consider the compilable program shown in Figure 2. It is generated by our tool, ATLAS, which triggered an internal compiler error (ICE) when GCC-15 compiled it with `-O2 -fsanitize=address,null`. The `-fsanitize=address` uses AddressSanitizer (ASan) [44] to detect memory-related bugs in programs and the option `-fsanitize=null` is used to detect null pointer in programs. The highlighted code of the figure denotes the attribute segments inserted by ATLAS. While the original program compiled normally, the introduction of these attributes caused the compiler to crash during the compilation of the modified program. The `always_inline` attribute is used to force the compiler to inline a function, even if the compiler's optimization heuristics would normally choose not to do so. The `no_sanitize_address` attribute is used to instruct the compiler not to instrument a particular function with ASan marks, even if ASan is enabled globally.

In this case, the `always_inline` function `foo()` is inlined into the `no_sanitize_address` function `bar()`, ASan marks (like `.ASAN_MARK` calls) are still left in the code during compiler optimization (`O2`). These leftover marks cause issues during later compilation stages, resulting in an internal compiler error (ICE). The compiler does not correctly remove the ASan instrumentation calls when inlining functions into a caller with sanitization disabled, leading to this bug. When attempting to inline the `foo()` function via the command line, the program must be compiled with the `-finline-functions` option. However, this would apply inlining to all functions within the program. Additionally, compiling the program with the `-finline-functions` and `-fsanitize=address` options, without applying the `always_inline` attribute to `foo()`,

does not trigger a compiler crash. The crash is only triggered when the `always_inline` and the `no_sanitize_address` attributes are applied separately to the `foo()` and `bar()` functions, respectively.

The example demonstrates that the combination of optimizations and attributes can result in uncommon scenarios and expose bugs within the compiler. The occurrence of such situations would not have been possible without the use of attributes. This also serves as evidence for the feasibility of utilizing attributes to achieve fine-grained control over the exploration of the compilation space for compiler testing.

3 Approach

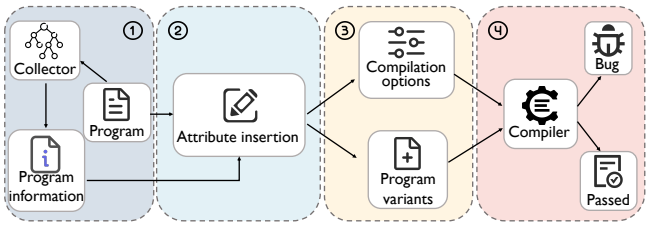


Figure 3: The workflow of ATLAS.

The fundamental workflow of our proposed approach is illustrated in Figure 3. Initially, in step ①, the program $\mathcal{P}$ is parsed to identify potential candidates that attribute could be inserted, forming the candidate set $\mathcal{S}$. Subsequently, in step ②, compiler attributes are randomly inserted into program $\mathcal{P}$ based on $\mathcal{S}$, creating a set of mutated program variants $\mathcal{S}_p$. Each variant $\mathcal{P}'$ in $\mathcal{S}_p$ explores a broader range of code paths within the compiler compared to $\mathcal{P}$. As discussed in Section 2.2, some attributes only take effect when specific compilation options are used. To ensure the inserted attributes are effective, we not only insert the attributes but also select the relevant compilation options. To enhance the complexity of the compilation options, a carefully designed strategy for combining compilation options is applied in step ③. This strategy involves selecting more diverse compilation options for each $\mathcal{P}'$, aiming to generate a more intricate combination of compilation options. In step ④, the compilation options selected in step ③ are combined with optimization options to compile the mutated program variants. If the compiler encounters any crashes during the compilation process, the bugs are found.

This section details our approach to combining compilation options and attributes to explore the compilation space for validating compiler reliability. Our strategy attempts to maximize the diversity of compilation option combinations with a wide range of attributes. This approach not only broadens the exploration of the compilation space but also increases the chances of uncovering potential compiler bugs.

3.1 Collecting Information

The initial step of our approach involves gathering essential information about functions, variables, and statements present in the test programs. Test programs typically comprise multiple functions and assorted variables. During the front-end parsing phase, the compiler analyzes different code fragments and variables individually and then performs targeted optimizations tailored to their specific requirements. As an illustration, LLVM follows a multistep process for program compilation. Initially, the program undergoes translation into an Intermediate Representation (IR). The compiler then traverses the IR, identifies specific structural patterns, and applies customized optimizations for those patterns [24, 34]. For instance, the `Instcombine` pass is responsible for identifying duplicate instructions and merging redundant ones through the execution of the pass [39]. Due to the different structure of functions, the compiler can perform different optimizations on them.

ATLAS collects function information, variable information, and statement information within the programs. We define the collector $\widehat{Collect}$ as follows:

Definition 1 (Collection). Given a program $\mathcal{P}$ and the function set $\mathcal{S}_f$, variable set $\mathcal{S}_v$, and statement set $\mathcal{S}_s$, the collector $\widehat{Collect}$ records the property tuple $\mathcal{A}$ for each function, variable, and statement in $\mathcal{P}$.

To provide a more detailed explanation, we define the following tuples to depict information from $\widehat{Collect}$:

- $\mathcal{A}_{\langle n,t,pos\rangle}$ contains the following information in $\mathcal{S}_f$: the function name n, the function return type t, and the function definition locations pos.
- $\mathcal{A}_{\langle t,pos\rangle}$ contains the following information in $\mathcal{S}_v$: the variable type t, and the variable declaration locations pos.
- $\mathcal{A}_{\langle vl,pos\rangle}$ contains the following information in $\mathcal{S}_s$: the variables vl in statement, and the statement locations pos.

To acquire insights into the program's structure, ATLAS initially extracts the Abstract Syntax Tree (AST) of the program. In our implementation, given a new seed program $\mathcal{P}$, we first utilize the $\widehat{Collect}$ collecting method to obtain $\mathcal{S}_f$, $\mathcal{S}_v$, and $\mathcal{S}_s$. Afterward, we generate program variant set $\mathcal{S}_p$ based on $\mathcal{S}_f$, $\mathcal{S}_v$, and $\mathcal{S}_s$.

3.2 Inserting Attributes

As detailed in Section 2.2, attributes allow developers to finely tune their code's behavior and performance, granting them better control over the compilation process. For instance, we can use `__attribute__((optimize(1))) int f();` to instruct the compiler to optimize the function `f()` at the O1 level. In light of this, we propose a method for fine-grained

control of compilation by leveraging compiler function attributes. More specifically, we utilize these attributes to explicitly influence the decisions made by the compiler. Algorithm 1 lists the primary procedure used in ATLAS to achieve fine-grained control of compilation by randomly inserting attributes into the program.

Algorithm 1: Insert Attributes

Input: $\mathcal{P}$, original program
$\mathcal{S}_{fa}$, function attribute set
$\mathcal{S}_{va}$, variable attribute set
$\mathcal{S}_{sa}$, statement attribute set
$\mathcal{S}_f$, function set of the original program
$\mathcal{S}_v$, variable set of the original program
$\mathcal{S}_s$, statement set of the original program
$\mathcal{N}$, the number of mutated programs
Output: $\mathcal{S}_p$, the set of programs with attributes inserted

1 $\mathcal{S}_p \leftarrow [\,]$;
2 $\mathcal{P}' \leftarrow \mathcal{P}$;
3 $n \leftarrow 0$;
4 **while** *true* **do**
5 **foreach** $\mathcal{A}_{\langle n,t,pos\rangle} \in \mathcal{S}_f$ **do**
6 **if** *FlipCoin()* **then**
7 $attr \leftarrow formFuncAttr(\mathcal{S}_{fa}, \mathcal{A}_{\langle n,t,pos\rangle})$;
8 $pos \leftarrow \mathcal{A}_{\langle n,t,pos\rangle}.getPos()$;
9 $\mathcal{P}' \leftarrow insertAttr(\mathcal{P}', attr, pos)$;
10 **foreach** $\mathcal{A}_{\langle t,pos\rangle} \in \mathcal{S}_v$ **do**
11 **if** *FlipCoin()* **then**
12 $attr \leftarrow selectVarAttr(\mathcal{S}_{va}, \mathcal{A}_{\langle t,pos\rangle})$;
13 $pos \leftarrow \mathcal{A}_{\langle t,pos\rangle}.getPos()$;
14 $\mathcal{P}' \leftarrow insertAttr(\mathcal{P}', attr, pos)$;
15 **foreach** $\mathcal{A}_{\langle vl,pos\rangle} \in \mathcal{S}_s$ **do**
16 **if** *FlipCoin()* **then**
17 $attr \leftarrow formStmtAttr(\mathcal{S}_{sa}, \mathcal{A}_{\langle vl,pos\rangle})$;
18 $pos \leftarrow \mathcal{A}_{\langle vl,pos\rangle}.getPos()$;
19 $\mathcal{P}' \leftarrow insertAttr(\mathcal{P}', attr, pos)$;
20 $\mathcal{S}_p.add(\mathcal{P}')$;
21 n++;
22 **if** $n >= \mathcal{N}$ **then**
23 **break**
24 **return** $\mathcal{S}_p$

Function Attribute Insertion. For each function within program $\mathcal{P}$, ATLAS employs $FlipCoin()$ to probabilistically determine the inclusion of each attribute from the attribute set $\mathcal{S}_{fa}$ (Line 5-9 in Algorithm 1). The $formFuncAttr()$ method selects one or more attributes and ATLAS inserts them into each function (Line 7 in Algorithm 1). Figure 4 illustrates an example of how ATLAS inserts function attributes. First, ATLAS identifies the location of function definitions based on position information *pos* from $\mathcal{A}_{\langle n,t,pos\rangle}$ marking these locations with <func>. After marking the locations, ATLAS selects appropriate function attributes based on the function return type t and inserts them at the specified positions. For instance, the return type of `c()` is `int`, we cannot apply `noreturn` attribute to it. Since $\mathcal{P}_1$ contains two function definitions, two corresponding marked locations are created. The mutated program $\mathcal{P}_f$ shown in Figure 4-(b), is derived from $\mathcal{P}_1$, with the highlighted code segments representing the function attributes inserted by ATLAS.

Variable Attribute Insertion. For each variable within program $\mathcal{P}$, the method *selectVarAttr*() picks appropriate variable attributes through type checking (Line 12 in Algorithm 1). For example, the `packed` attribute, which instructs the compiler to pack structure members tightly together without inserting padding [20], cannot be applied to variables of type `int`. As such, the method *selectVarAttr*() will only apply `packed` attribute to member variables within a structure. In the case of program $\mathcal{P}_2$ shown in Figure 5-(a), ATLAS identifies the positions of all variables and marks them with <var> using the position information *pos* from $\mathcal{A}_{\langle t,pos\rangle}$. The mutated program $\mathcal{P}_v$ in Figure 5-(b) is derived from $\mathcal{P}_2$, where ATLAS inserts the `vector_size` attribute into a member variable of the structure `d`. Since there are two <var> marks in $\mathcal{P}_2$, ATLAS selects the one within the structure `d` based on the outcome of the $FlipCoin()$ decision.

```
int a;
int b(int);
<func> int c() {
  a = 1;
  while (a)
    a = 2;
  return a;
}
<func> int e(void) {
  int d = c();
  b(d);
}
```

(a) Original Program $\mathcal{P}_1$.

```
int a;
int b(int);
__attribute__((pure,
  returns_twice))
int c() {
  a = 1;
  while (a)
    a = 2;
  return a;
}
int e(void) {
  int d = c();
  b(d);
}
```

(b) Mutated Program $\mathcal{P}_f$.

Figure 4: Illustration examples of function attributes insertion. $\mathcal{P}_f$ was mutated from $\mathcal{P}_1$ by inserted function attribute. This program triggers the GCC bug 114687 when compiled with -fsanitize=address -O1.

```
<var> int a;
void b();
int e(int *c, struct d {
<var> char an[*c]} *) {
  sizeof(struct d);
}
void f() {
  if (e(&a, 0))
    b();
}
```

(a) Original Program $\mathcal{P}_2$.

```
int a;
void b();
int e(int *c, struct d {
__attribute__((
  vector_size(4)))
  char an[*c]} *) {
  sizeof(struct d);
}
void f() {
  if (e(&a, 0))
    b();
}
```

(b) Mutated Program $\mathcal{P}_v$.

Figure 5: Illustration examples of variable attributes insertion. $\mathcal{P}_v$ was mutated from $\mathcal{P}_2$ by inserted variable attribute. $\mathcal{P}_v$ triggers the GCC bug 117145 when compiled with -O1.

Statement Attribute Insertion. The process of statement attribute insertion is outlined in Algorithm 2. This process begins with the use of the function $FlipCoin()$ to determine which attributes from the set $\mathcal{S}_{sa}$ will be utilized in subsequent steps (Line 4). The method *isEmpty*() is employed to verify the presence of variables within the statement (Line 6). If variables are indeed present, the *Construct*() method gener-

ates a simple expression based on the identified variable list *vl* to accomplish the corresponding attribute. Notably, each statement is associated with only a single attribute. To provide a clearer understanding, consider program $\mathcal{P}_3$ as depicted in Figure 6-(a), where ATLAS annotates the loop statement with `<stmt>`. The variables `x` and `y` are included in the variable list; consequently, the *Construct*() method selects a `linear` modifier and produces simple expression `(uval (x): y + 1)`. The highlighted code in Figure 6-(b) illustrates the attribute generated by $formStmtAttr()$. `#pragma` is also a form of attribute. For clarity, this line of code has been folded to enhance readability.

```
void a (int &x, int &y) {
  <stmt>
  for (i=0; i<10; i++) {
    x += y + 1;
  }
}
```

(a) Original Program $\mathcal{P}_3$.

```
void a (int &x, int &y) {
  #pragma omp simd
    linear(uval (x): y+1)
  for (i=0; i<10; i++) {
    x += y + 1;
  }
}
```

(b) Mutated Program $\mathcal{P}_s$.

Figure 6: Illustration examples of statement attributes insertion. $\mathcal{P}_s$ was mutated from from $\mathcal{P}_3$ by inserted statement attribute. This program triggers the LLVM bug 108166.

Algorithm 2: Form Statement Attributes

```
1  Function formStmtAttr(S_sa, A_⟨vl,pos⟩):
2      a ← " ";
3      foreach attr ∈ S_sa do
4          if FlipCoin() then
5              var_list ← A_⟨vl,pos⟩.getVarList();
6              if !var_list.isEmpty() then
7                  a ← Construct(var_list);
8                  break
9              else
10                 a ← attr;
11     return a
```

3.3 Selecting Options

In practical use, certain attributes become effective only when specific compilation options are applied. As illustrated in Figure 2, we utilize the `no_sanitize_address` attribute to disable the instrumentation of `bar()` function. Such attribute will only exert its influence during the compilation process if the `-fsanitize=address` option is specified. Following the insertion of attributes in Section 3.2, ATLAS selects the appropriate compilation options based on the attributes inserted to ensure their effectiveness.

The procedure for this selection is outlined in Algorithm 3. In the map $\mathcal{M}_o$, each attribute (key) corresponds to the compilation options (value) that activate it. The *getInsertedAttr*() method retrieves all attributes that have been inserted into the program $\mathcal{P}$ (Line 2). By iterating the attributes in the list $\mathcal{L}_{attr}$,

Algorithm 3: Select Compilation Options

```
Input: P, the programs with attributes inserted
       M_o, the map of attributes and their options
Output: O, the compilation option of P
1  select ← set();
2  L_attr ← getInsertedAttr(P);
3  foreach attr ∈ L_attr do
4      if attr ∈ M_o.keys() then
5          select.add(M_o.getValues(attr));
6  O ← FormOpt(select);
7  return O
```

the corresponding compilation options are retrieved (Line 3-5). These options are then consolidated to form the final compilation options O for the program $\mathcal{P}$ (Line 6).

3.4 Validating Compiler

Once the program variants are generated, the testing process becomes straightforward. The modifications made by ATLAS, such as attribute insertion, do not compromise the overall structure of the programs. For each variant, a stochastic selection process is employed to randomly choose a sequence of optimization options from a set of options O_{all}. O_{all} contains all optimization options for the compiler. The number of selected optimization options does not exceed five, as previous research investigations have indicated that the average number of optimization options leading to bugs is five [6]. These selected options are then combined with the options in O determined by Algorithm 3 for compilation along with the program variant. We define the validation process as follows.

Definition 2 (Validation). Program $\mathcal{P}_i$ is a variant of program $\mathcal{P}$ and the option O_i is carefully designed. The compiler under test is denoted as $\mathcal{C}omp$, which can receive compilation options and test programs as inputs. If $\mathcal{C}omp(\mathcal{P}_i, O_i)$ crashed, a compiler bug is identified.

Our key insight is that when considering a valid program $\mathcal{P}$, the compiler should exhibit stability without encountering any crashes throughout the compilation process with a specific compilation option, denoted as O_p. Figure 2 illustrates the final results of our tool. In the case of the attribute-inserted variant, detecting the bug requires both the selection of appropriate compiler options (`-fsanitize=address,null`) and the application of suitable optimization settings (`-O2`). The results illustrated in Figures 4 and 5 further substantiate the importance of selecting the appropriate optimization options in the identification of compiler bugs.

4 Evaluation

This section presents our extensive evaluation of ATLAS to find bugs in compilers. Our evaluation is based on the follow-

ing research questions:

- **RQ1 (Effectiveness):** How many compiler bugs have been found by ATLAS?
- **RQ2 (Efficiency):** How efficient is ATLAS in uncovering compiler bugs?
- **RQ3 (Capability):** Can ATLAS improve code coverage?

4.1 Implementation and Evaluation Setup

Implementation. ATLAS is designed to run in a fully automated way, including program information collection, attribute-guided mutation, and validation stages. ATLAS utilizes LLVM's LibTooling library [36] to facilitate the program information collection task as described in Section 3.1. Once launched, ATLAS will automatically generate programs and feed them into the compilers under test.

Target Compilers. We used ATLAS to continuously test the trunk version of two widely used compilers: GCC and LLVM. Specifically, we obtained daily builds of GCC from its repositories and utilized nightly packages of LLVM [37]. Testing on the latest development versions of software offers several advantages:

- By testing these trunk versions, developers can immediately identify and address these bugs.
- This approach can avoid duplicate bug reports as much as possible because once bugs are found in the trunk version, developers tend to quickly fix them [25, 52].

Test programs. In our testing, we have drawn from two sources of programs to generate test programs:

- *Compiler Test Suites*: Each of GCC and LLVM has an already sizable and expanding regression test suite [17, 35], which we can use for generating various test programs. We chose GCC test suite and LLVM test suite because: (1) they are open source which can be easily obtained; (2) they are typically used for compiler regression testing, covering a wide range of C semantics; and (3) many of these test programs can be compiled and executed independently, as they do not rely on external libraries and have fixed inputs.
- *Generated Code*: We choose Csmith [52] and YARPGen [32, 33] as the random program generators to generate C programs to test compilers for the following main reasons: (1) they are extensively used in the literature of C compiler testing [6, 26, 46]; (2) they are effective in finding bugs as thousands of bugs have been exposed and reported for the most widely-used C compilers; (3) each test case generated by Csmith and YARPGen is valid and does not require external inputs; and (4) they are efficient as a test program with tens of thousands of lines can be generated quickly.

Attributes. All the attributes we used are from the GCC and LLVM documentation [10, 15]. The attributes and options map $\mathcal{M}_o$ described in Algorithm 3 was also formed from the documentation. We chose attributes for the following main standard: (1) Programming Language Compatibility: The attribute must be compatible with the C programming language. (2) Exclusions: Attributes that are specifically designed for GPU support are not considered. The attribute sets utilized for GCC and LLVM in our experiments were distinct.

Hardware. Our evaluation was conducted on a workstation with an Intel 48-core 2.30GHz CPU, 120GiB RAM, and Ubuntu 22.04.2 LTS operating system.

Testing process. Our testing process is fully automated and runs continuously. ATLAS first inserts attributes into programs for the test program to get the program variants set, then chooses a proper compilation option for each variant. The program variant with a range of compilation options is used to test GCC and LLVM. Once a program triggers a compiler bug, we use C-Reduce [43] to reduce the program. If the program triggers a new compiler bug, we report the bug to the developers. The only manual step in the testing is analyzing and reporting bugs.

4.2 RQ1: Bug-finding

Table 1 summarizes the bugs we discovered during our testing period. Overall, we reported 73 bugs in total: 44 for GCC and 29 for LLVM. The developers have confirmed 58 of them. This highlights the bug-finding capability of ATLAS. Of all these bugs, 17 of them have been fixed of which 13 fixed bugs are in GCC and 4 fixed bugs are in LLVM[2]. We also experienced that the LLVM developers were less responsive than GCC and mostly only labelled our reports as bugs without further diagnosis. The results presented in the "Options" column indicate whether the triggering of a bug requires compilation options. If such options are necessary, the bug is marked with ✓; if not, it is marked with ✗. A total of 49 bugs require compilation options for triggering, while only 24 bugs can be triggered solely through the insertion of attributes. In the Table 1, the "Priority" is priority of a bug report. The field priority of a bug report is determined by developers for prioritizing bugs, expressing the order in which bugs should be fixed [47]. P1 is the most urgent level and bugs of this level should be fixed soon, and P3 is the default priority level. LLVM developers do not prioritize bugs explicitly in the bug repository, so we only present the priority on GCC.

Note that the bug management guidelines of GCC and LLVM are not exactly similar. Specifically, a reported bug in GCC Bugzilla will be initially labelled as "UNCONFIRMED", and the status will turn to "NEW" if developers confirm it. However, in LLVM issue tracker, a reported bug

[2]For one GCC bug (ID: 116687), developers commented it as "Confirmed", unless they marked them as "NEW", we still regard it as unconfirmed.

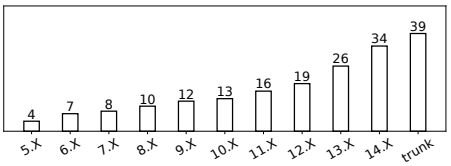
Table 1: The bugs reported by ATLAS.

ID	Priority	Status	Options	Attributes	ID	Priority	Status	Options	Attributes
The reported bugs of GCC					The reported bugs of GCC				
112709	**P2**	Fixed	✓	returns_twice	117461	P3	Unconfirmed	✓	returns_twice
112860	P3	Fixed	✓	vector_size(var)	117479	P5	Confirmed	✗	vector_size(func)
113043	P3	Confirmed	✓	interrupt	117489	**P2**	Confirmed	✓	pure,sanitize
113264	P3	Confirmed	✗	alias,sanitize,target_clones,copy	117505	P3	Confirmed	✗	used
114687	**P1**	Fixed	✓	pure, returns_twice	117512	**P2**	Fixed	✗	aligned(var)
114956	**P2**	Fixed	✓	always_inline,sanitize	117540	P3	Duplicate	✓	flatten
115509	P3	Unconfirmed	✗	aligned(func),sentinel	117979	**P2**	Assigned	✓	returns_twice,const(func)
115530	P3	Confirmed	✓	simd(func)	The reported bugs of LLVM				
115548	P3	Unconfirmed	✗	malloc,simd(func)	73378	-	Confirmed	✓	noinline
115549	P3	Fixed	✗	aligned(func),optimize(s)	73797	-	Duplicate	✓	noinline
115573	P3	Unconfirmed	✓	no_reorder	74189	-	Fixed	✓	packed
115595	P3	Unconfirmed	✓	optimize(1),noipa	75019	-	Confirmed	✗	noinline
115815	**P2**	Fixed	✓	destructor	75301	-	Confirmed	✓	vector_size
115816	P3	Confirmed	✓	vector_size(var),target	75321	-	Confirmed	✓	loop_vectorize(stmt)
115847	P3	Confirmed	✗	simd(func),vector_size(func)	75690	-	Duplicate	✓	vector_size
115848	P3	Assigned	✓	strub	88692	-	Confirmed	✓	minsize,sanitize
115859	P3	Confirmed	✓	optimize(3)	95892	-	Confirmed	✗	opencl_global_device
115861	P3	Unconfirmed	✗	deprecated	95928	-	Confirmed	✓	preserve_none
116607	P3	Fixed	✓	sanitize	96018	-	Confirmed	✗	interrupt
116612	P3	Duplicate	✗	aligned(var)	96383	-	Confirmed	✓	simd
116659	P3	Duplicate	✓	optimize("Ofast")	98635	-	Confirmed	✗	regcall
116687	P3	Unconfirmed	✓	simd(func)	98663	-	Confirmed	✗	vectorcall
117142	**P2**	Fixed	✓	returns_twice	107979	-	Confirmed	✓	atomic
117145	**P2**	Fixed	✓	vector_size(var)	108166	-	Confirmed	✓	simd
117167	P3	Duplicate	✓	const(func)	113285	-	Confirmed	✓	nodebug
117197	P3	Confirmed	✗	vector_size(var)	113401	-	Confirmed	✓	preserve_most
117209	**P2**	Fixed	✓	returns_twice,pure	113403	-	Confirmed	✓	speculative_load_hardening
117230	**P2**	Fixed	✓	vector_size(var)	113420	-	Confirmed	✓	target
117245	**P2**	Assigned	✗	vector_size(var)	115299	-	Fixed	✗	cpu_specific
117254	**P2**	Fixed	✗	nonstring	115411	-	Fixed	✓	optnone
117295	P3	Unconfirmed	✓	returns_twice	115490	-	Confirmed	✗	no_caller_saved_registers
117326	P3	Confirmed	✓	constructor	115602	-	Confirmed	✓	cold
117358	**P2**	Assigned	✓	const(func)	115632	-	Confirmed	✓	address_space
117380	P4	Confirmed	✗	vector_size(var)	115655	-	Confirmed	✗	vector_size
117429	P3	Duplicate	✓	no_sanitize,flatten	116010	-	Confirmed	✓	alias,constructor
117437	P3	Confirmed	✓	sanitize	116124	-	Fixed	✗	noderef
117440	**P2**	Confirmed	✓	pure	121865	-	Confirmed	✗	vectorcall

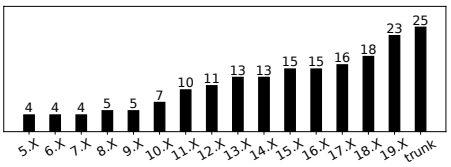
will be labelled as “new issue" by default, and if the developers confirm it, they will remove the “new issue" label and add other labels. For example, if the bug is related to link time optimization, it will be labelled as “LTO". In addition, if the status of a bug is “Assigned" means the bug is confirmed and is assigned to a certain developer to deal with it. If a bug is fixed by developers in LLVM issue tracker, the corresponding issue report will be closed. If the bug has the same root cause with the bug reported by other developers previously, we categorize it as “Duplicate".

Priority of bugs. To analyze the importance of the bugs, we present the priority levels of GCC bugs in the Table 1. One bug (ID:114687) has been marked as P1 priority by the developers, and 14 bugs have been marked as P2 priority. The higher priority assigned to these bugs indicates the developers' attention to the bugs. Developers have to fix all P1 bugs before making the next release [30].

Influence of bugs. To further understand the influence of our reported bugs, we investigate the official release versions of compilers affected by the reported bugs. Here, we select GCC-5 (released in 2015) and LLVM-5 (released in 2017) as the earliest versions of the subjects because they are the first stable versions that support sanitizers [29] and our compilation options include sanitize options. The results indicate that



(a) Affected GCC versions



(b) Affected LLVM versions

Figure 7: Compiler versions affected by the reported bugs.

ATLAS is capable of identifying critical bugs. As illustrated in Figure 7, four enduring bugs spanning eight years are evident within the GCC-5, signifying its persistence over a considerable period. Similarly, in LLVM, four bugs have persisted within the compiler for a duration of six years. Most bugs affect at least two compiler versions. These findings demonstrate the capability of ATLAS in uncovering long-standing bugs, further emphasizing the significance and impact of our

bug-finding results.

Severity of the bugs. Some of the identified bugs have impact on compiler correctness and reliability. For example, GCC Bug 112709 leads to an ICE during sanitizer instrumentation on function return values, which directly causes the compiler to crash. GCC Bug 114687 reflects an overly aggressive constraint in handling the function call during instrumentation, which may result in incorrect control-flow or optimization behavior. These issues demonstrate fragility in the compiler's Static Single Assignment (SSA) and Control Flow Graph (CFG) transformation logic, especially under instrumentation. Other bugs affect semantic correctness or type handling. For instance, GCC Bug 117145 occurs when a vector_size attribute is applied to a struct, and the compiler fails to treat it as a Variable Modified Type (VMT), leading to type analysis errors. GCC Bug 117230 misuses type precision on vector types, potentially resulting in incorrect code generation or undefined behavior when vector types are involved. Bugs related to SIMD support can result in broken vectorization or compilation failure. GCC Bug 115847 misinterprets the simd attribute on functions with vector return types, breaking code generation. LLVM Bugs 96383 and 108166 show the compiler's inability to handle loop-level SIMD attributes properly, which causes compilation errors when auto-vectorization is attempted. These examples represent only a subset of the bugs we identified. Many other issues were found across different components of the compiler. Overall, these bugs impact the reliability of compilers in real-world scenarios.

Table 2: The number of affected compiler components.

Component	GCC	LLVM	Total
Front-End	17	6	23
Middle-End	13	8	21
Optimization	11	5	16
Back-End	3	10	13

Affected Compiler Components. As shown in Table 2, the 73 identified bugs span a wide range of compiler components, with the front-end being the most affected, accounting for 31.5% of the total. Additionally, 28.8% of the bugs occur in the middle-end, suggesting that attributes have a significant impact on the generation and manipulation of intermediate representations (IR), a critical phase where high-level code is transformed into an optimized, lower-level form. Furthermore, 16 bugs were found within the optimization components. This finding highlights that attribute insertion can influence the compiler's optimization processes, triggering rarely exercised code paths. The presence of such bugs underscores the importance of thoroughly testing the interaction between attributes and optimization routines, particularly in complex or highly optimized code scenarios.

Valuable attributes. The attribute combination of each bug is listed in Table 1 tables. Since some attributes can be applied to both functions and variables, we explicitly annotate their usage: "func" represents function attribute, "var" represents variable attribute, "stmt" represents statement attribute. To ascertain the effectiveness of various attributes in bug detection, we analyzed the 73 test cases that trigger bugs. Specifically, we find that each test case contained at least one attribute, indicating that attribute insertion is essential. Delving deeper, we find that certain attributes are more frequently involved in bug-triggering test cases. Table 3 presents the number and types of attributes involved in the test cases. The attributes of GCC and LLVM displayed in the table reveal differences. For example, in the context of optimization-related attributes, LLVM and GCC have distinct approaches to applying optimization settings to functions. LLVM typically disables optimization for a function by specifying the `optnone` attribute on the function. Notably, the largest number of function attributes is `return_twice` in GCC (7). A variety of optimization-related attributes were also identified. Additionally, the attributes `always_inline`, `noipa` and `noinline` are utilized to control function-level optimizations, influencing how specific functions are treated during the optimization process. These findings highlight the effectiveness of our approach in uncovering bugs by fine-tuning function optimization through attribute manipulation. The `vector_size` attribute, which can be applied to both variables and functions, requires careful distinction based on its usage context. This attribute proves effective in detecting bugs in both GCC and LLVM, as compilers often struggle with optimizing code when `vector_size` is used, particularly in complex expressions or with multiple vectorized operations. Furthermore, the

Table 3: The distribution of the valuable attributes. "#" denotes the number of bugs detected by ATLAS.

Attribute	Type	#	Attribute	Type	#
The attributes of GCC					
vector_size	var.	7	alias	func.	1
return_twice	func.	7	no_reorder	func.	1
sanitize	func.	5	noipa	func.	1
simd	func.	4	destructor	func.	1
optimize	func.	4	malloc	func.	1
pure	func.	4	strub	func.	1
const	func.	3	deprecated	func.	1
malloc	func.	2	copy	func.	1
target	func.	2	sentinel	func.	1
aligned	var.	2	nonstring	var.	1
aligned	func.	2	copy	func.	1
flatten	func.	2	always_inline	func.	1
vector_size	func.	2	used	func.	1
constructor	func.	1	interrupt	func.	1
The attributes of LLVM					
noline	func.	3	speculative	func.	1
vector_size	var.	3	interrupt	func.	1
simd	stat.	2	atomic	func.	1
vectorcall	func.	2	optnone	func.	1
preserve_none	func.	1	nodebug	func.	1
minsize	func.	1	preserve_most	func.	1
sanitize	func.	1	packed	var.	1
opencl	func.	1	regcall	var.	1
loop_vectorize	stmt.	1	target	func.	1
cpu_specific	func.	1	register	func.	1
cold	func.	1	address_space	var.	1
alias	func.	1	constructor	func.	1
noderef	var.	1	-	-	-

`simd` attribute uncovered several bugs in both compilers due to its role in generating SIMD instructions for loops. Automatically transforming scalar operations into vectorized forms is a complex process; the compiler must analyze dependencies and ensure that operations can be executed in parallel without introducing errors. Such observations suggest that compiler developers should pay more attention to these attributes to ensure the compiler's correctness.

Commonly Used Attributes. The attributes that triggered bugs found by ATLAS are not rarely used; rather, they are commonly found in real-world software systems. For instance: the `optimize` and `optnone` attributes are used to enforce or disable optimization on specific functions, which is common in debugging, benchmarking, and embedded systems; The `simd` attribute is prevalent in high-performance computing (HPC), multimedia processing, and numerical libraries where loop vectorization is critical; The `malloc` attribute is frequently applied to memory allocation routines to improve alias analysis and enable optimizations; The `aligned` attribute is widely used in performance-critical software to ensure proper memory alignment for data structures, particularly in low-level systems code and signal processing; The `vector_size` attribute is common in SIMD programming and is used in many numerical and graphics applications. These attributes appear regularly in system libraries, graphics engines, deep learning frameworks, HPC software, and performance-tuned applications. As a result, the bugs associated with them are likely to be triggered in real-world development.

4.3 RQ2: ATLAS Efficiency

Baseline approaches. We compared ATLAS with the most direct and the state-of-the-art approach for compiler testing:

- **Random Option (RO):** To isolate and understand the contribution of ATLAS, we evaluate ATLAS without incorporating attributes into the programs. The **RO** approach represents the conventional approach, where programs are generated randomly and paired with randomly selected optimization flags. This method aims to explore the compilation space without any guided mechanism, thereby serving as a fundamental baseline for assessing the efficacy of our proposed method.
- **Attribute Only (AO):** Since ATLAS also compiles programs with compilation options, it is necessary to investigate the contribution of this part. Therefore, we evaluate ATLAS without compilation options compiled with programs. In this scenario, attributes that depend on specific options to be effective are excluded from insertion into the program. **AO** helps in quantifying the extent to which compilation options enhance the exploration of the compilation space. The approach **AO** ensures broad coverage of the compilation space by using attributes, albeit without specific direction.
- **Equivalence Modulo Inputs (EMI):** We also compared ATLAS against Hermes [46], which is the most recent and powerful EMI-based compiler testing tool.

These comparative analyses provide a comprehensive understanding of the strengths and weaknesses of ATLAS in relation to both the **EMI**, **RO** and **AO**, a method devoid of its core attribute-based exploration strategy. We use Csmith and YARPGen as the generators to generate the test programs.

Measurements. Following the existing work [6, 7, 51], we used the number of detected bugs within the same testing period to measure the performance of ATLAS. Same as the existing study [6, 7, 51], we tested each subject using each approach for 10 days , and adopted Correcting Commits [4], a method commonly used in existing studies [6, 8, 51], to identify the number of detected bugs from a set of failing test programs. More specifically, for any test program that triggers a bug of a compiler C whose commit version is x, the Correcting Commits method checks subsequent commits of the compiler and determines which commits fix the bug. If two failing test programs have the same correcting commit, they are regarded as triggering the same bug. The number of correcting commits is approximately regarded as the number of detected bugs. In fact, the accuracy of this method is quite substantial as existing studies [4, 8] demonstrate. This method may be a threat to our study. However, it is the only automatic method to measure the number of detected bugs and its accuracy has been demonstrated by the existing study, and thus this threat may be not serious. We applied **RO**, **AO**, **EMI**, and ATLAS to test GCC-5.1 and LLVM-5.0.0. We choose these old releases rather than the latest release since: 1) they are the stable versions that support sanitizers [29] and the compilation options includes sanitizer options; 2) they released for a long time, and thus most bugs have been exposed; 3) most bugs have been fixed as they have been maintained for a long period, thus we will have enough correcting commits to evaluate different strategies.

Table 4: The number of detected bugs and generated compilable programs within 10 Days.

Generators	Techniques	GCC Bugs(#)	LLVM Bugs(#)	Total Bugs(#)	Compilable Programs(#)	Total Programs(#)
Csmith	RO	13	5	18	53,458	53,481
	AO	19	11	30	32,043	47,825
	EMI	17	11	28	48,765	49,279
	ATLAS	32	17	49	30,170	46,416
YARPGen	RO	8	6	14	123,087	123,192
	AO	12	10	22	70,361	108,085
	EMI	10	9	19	101,734	103,628
	ATLAS	25	13	38	62,142	99,439

Number of detected bugs. Table 4 shows the number of detected bugs within 10 days. For programs generated by Csmith, the total number of bugs identified by ATLAS, **AO**, **RO** and **EMI** is 49, 30, 18 and 28, respectively. For programs

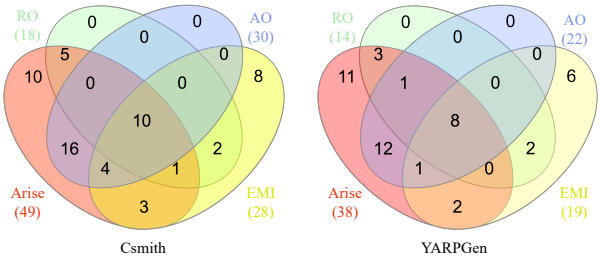


Figure 8: Unique bugs found by different techniques.

generated by YARPGen, the corresponding numbers are 38, 22, 12, and 19. A comparative analysis indicates that ATLAS consistently outperforms **AO**, **RO** and **EMI** across both generators, demonstrating the effectiveness of ATLAS.

Number of unique bugs. We further analyzed the number of unique bugs detected by each approach, whose results are shown in Figure 8. More specifically, these Venn diagrams show various relations among the bugs detected by the four approaches, including the unique bugs and the common bugs. For programs generated by Csmith, ATLAS detects 10 unique bugs, accounting for approximately 20.4% of its total findings. This result demonstrates the distinct bug-finding capability of ATLAS. Moreover, ATLAS can detect most (18) of the bugs detected by **EMI**. **AO** identifies more bugs than **RO** and `EMI`, highlighting the effectiveness of using attribute insertion for compiler testing. Furthermore, ATLAS is able to detect all the bugs identified by method **AO**. In the case of YARPGen-generated programs, ATLAS again detected all bugs identified by **AO** and uncovered 11 unique bugs.

Although ATLAS is superior in detecting compiler bugs compared with **RO** and **AO**, **EMI** can also detect some unique bugs. We further analyze the reason for this phenomenon. Regarding **EMI**, it modifies the program's structure and enhances its diversity by adding and removing code. Additionally, it compiled and executed more test programs than ATLAS during the same testing period. As demonstrated by the existing study [7], the number of test programs indeed affects the performance of compiler testing to some degree, and thus this may result in the unique bugs detected by **EMI**.

There is a different performance between LLVM and GCC in Table 4. We summarize the reasons as follows: (1) The release time of the LLVM subjects is about two years later than the GCC subjects. Thus, these LLVM subjects should have fewer bugs than those GCC subjects. The chosen LLVM subjects are LLVM-5.0.0 released on 9/7/2017 [38]. While GCC-5.1 were released in 4/22/2015 [16]. There is a two-year gap between the LLVM subjects and the GCC subjects. In addition, Csmith was released in 2011. At that time, Csmith found 203 bugs for LLVM and 79 for GCC [52]. In other words, most bugs found by Csmith have been fixed in those LLVM subjects as they were released two years later after Csmith. (2) Each LLVM version is maintained for a shorter period than the three subject GCC versions. LLVM-5.0.0 was only maintained for ten months [40]. While the GCC-5.1 are respectively maintained for 30 months [14]. Since the GCC subjects were maintained for much longer than the LLVM subjects, the GCC subjects should have much more bug-fixing commits than LLVM during the maintenance period.

Compilable Programs. As shown in Table 4, **RO** and **EMI** generated the highest proportion of compilable programs among these approaches. This outcome is expected, as the programs generated by Csmith and YARPGen strictly adhere to the C language standard, whereas **EMI** performs equivalent mutations on these programs. The total number of test programs generated by **AO** and ATLAS is smaller than those produced by **RO** and **EMI**. This is because the program generation phase in **AO** and ATLAS involves both modifying the original program and selecting attributes, resulting in slightly higher overhead compared to **EMI**. Additionally, the proportion of programs successfully compiled by **AO** and ATLAS is lower than that of **RO** and **EMI**. This is primarily due to the insertion of certain attributes—particularly those related to variables, such as `vector_size`—which may conflict with the program's requirements for variable usage.

Time Trends. Figure 9 illustrates the time trends in the number of unique bugs discovered for both GCC and Clang, with the top two subplots corresponding to programs generated by Csmith, and the bottom two to those generated by YARPGen. For the Csmith generated programs, **RO** reached a saturation point after approximately 40 hours, making it challenging to identify additional bugs in GCC and LLVM. **EMI** approached saturation after more than 200 hours. Meanwhile, **AO** and ATLAS continued to uncover bugs over a period of 10 days. For programs generated by YARPGen, all four techniques were able to detect bugs continuously. However, as the number of discovered unique bugs increased, the time required for **AO**, **RO**, and **EMI** to find additional bugs also grew significantly. In comparison, ATLAS maintained a more consistent discovery rate, further demonstrating its efficiency and robustness in uncovering compiler bugs over time.

Table 5: Code coverage of GCC and LLVM.

Compiler	Techniques	Line	Function	Branch
GCC	**RO**	33.5%	35.2%	21.1%
	AO	34.7%(+13,080)	36.9%(+419)	22.0%(+10,697)
	EMI	35.3%(+19,619)	36.8%(+314)	22.4%(15,450)
	ATLAS	36.3%(+30,519)	37.6%(+1,152)	23.2%(+24,958)
LLVM	**RO**	34.4%	25.6%	20.0%
	AO	35.6%(+20,324)	26.5%(+849)	20.9%(+6,472)
	EMI	35.1%(+12,435)	26.9%(+1,307)	21.3%(+9,592)
	ATLAS	36.8%(+41,806)	27.6%(+1,953)	21.7%(+12,954)

4.4 RQ3: Code Coverage

To evaluate the effectiveness of ATLAS, we analyzed the coverage achieved while testing the compiler using this approach.

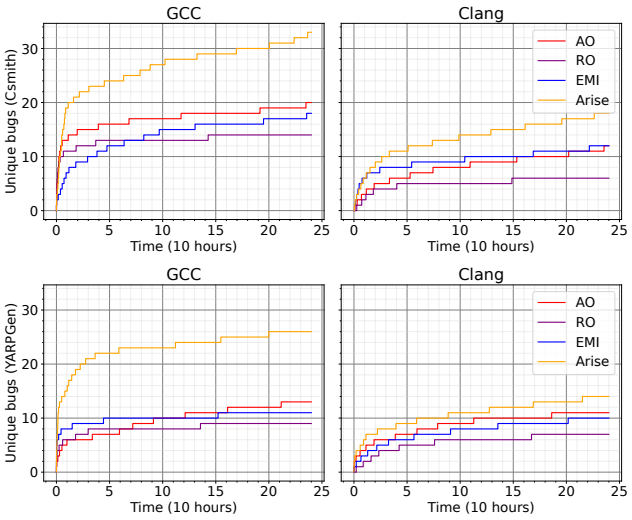


Figure 9: The earliest time each unique bug was discovered.

We randomly generated 10,000 seed programs by Csmith following the methodology outlined in previous work [30] and inserted attributes into these programs. These test programs were distinct from those generated in Section 4.3. For each set of seed programs and their corresponding variants, we utilized the most recent releases of compilers, such as LLVM-19.0.0 and GCC-15.0.0. The compilation of each program was constrained to a maximum of 10 seconds, and we measured the code coverage achieved during these compilations. The primary aim was to determine if the variants produced by ATLAS could improve code coverage compared to the original seed programs. An improvement in code coverage would suggest that our transformations effectively explore additional code paths within the compilers, thereby increasing the likelihood of triggering compiler bugs.

The results of these experiments are summarized in Table 5 (the best results are marked in gray), which displays the code coverage metrics for line, branch, and function coverage achieved by four techniques: **RO** and **AO**, **EMI** and ATLAS. Although the coverage percentages are relatively similar, a detailed analysis of the specific values reveals an improvement of tens of thousands in the number of covered lines and branches. These results indicate that ATLAS improves code coverage across all metrics of the compilers under test. By triggering more code paths in the compilers, ATLAS not only enhances the thoroughness of compiler testing but also increases the potential for uncovering hidden bugs. This demonstrates the practical value and efficacy of ATLAS.

5 Discussion

5.1 Threats to Validity

We have identified the following main threats to validity:

Impact of subjects. One potential threat to the validity of our results is the representativeness of the compiler versions used in our experiments. In Section 4.3, we acknowledge that using a single version of a compiler may not adequately reflect the diversity of compilation scenarios encountered in practice. To mitigate this threat, we selected the two most widely used C compilers, GCC and LLVM, and evaluated multiple versions of each. This approach is consistent with existing studies [3, 4, 6, 21, 22, 26, 27, 41, 45, 47, 52], which have demonstrated the importance of considering a variety of compiler versions to ensure a comprehensive evaluation. By incorporating different versions, we aimed to enhance the generalizability and robustness of our findings regarding the effectiveness of ATLAS.

Impact of metrics. The accuracy of our bug detection metrics is another critical aspect that could affect the validity of our study. We employed the Correcting Commit approach, as used in prior research [4, 6–8], to estimate the number of bugs in our test programs that trigger errors. Although this method may not achieve perfect precision, it is currently the only automated approach available for estimating bug counts. Previous research [4] has demonstrated the reliability of this method, suggesting that its limitations do not pose significant concerns. Furthermore, we took measures to mitigate the occurrence of duplicate false positives by carefully analyzing crash messages. False positives were rare in our study, and we did not encounter any instances of false positives when testing with GCC and LLVM.

Test programs. To reduce the threat posed by the test programs, we used the most widely used C test program generation tools, Csmith and YARPGen, following many existing compiler-testing studies [11, 25–27, 48]. Additionally, many other test program generation tools are also adapted from Csmith and YARPGen [24, 29]. We also incorporated test suites from both GCC and LLVM for testing.

5.2 Integration with Other Techniques

ATLAS can be easily integrated with other compiler testing techniques. Specifically, ATLAS can take test programs generated by tools such as Csmith [52], YARPGen [32, 33], and EMI [25] as seed inputs. For each seed program, ATLAS parses its AST and identifies locations where attributes can be meaningfully inserted. By systematically injecting attributes at these locations, ATLAS enables exploration of additional compiler compilation space, such as those involving optimization, code generation for special attributes, or attribute-sensitive transformations. Take EMI [25] as an example. EMI generates program variants that are semantically equivalent on a given input. After these variants are created, ATLAS can further process each of them by analyzing their ASTs and inserting attributes at appropriate positions. This layered approach enables the discovery of bugs that only manifest under specific combinations of program structure and attributes. A similar integration can be achieved with

Creal [30], which generates test cases by inserting real-world functions. ATLAS can apply attribute insertion to these programs as well. In summary, ATLAS is highly compatible with existing compiler testing frameworks and can be seamlessly integrated to explore a broader compilation space.

6 Related Work

This section introduces the related work on compiler testing techniques and compilation space exploration.

6.1 Compiler Testing Techniques

The identification of compiler bugs has been the subject of extensive research, with significant efforts dedicated to testing various functionalities of compilers. Notable contributions in this area include the widely-used Csmith [52], a program generator specifically designed for C, which has successfully uncovered numerous compiler crashes and correctness bugs. It is worth noting that programs generated by Csmith are guaranteed to be devoid of undefined behavior. Another approach, Equivalent Modulo Input (EMI) [25], has been proposed for mutating or transforming seed programs while preserving their semantics under the same input. EMI can be implemented through techniques such as deleting dead statements [25], inserting new code in inactive regions [26], or synthesizing equivalent code in active regions [46]. When combined with Csmith, EMI has proven effective in uncovering thousands of compiler optimization bugs. YARPGen [32, 33] is another program generator for C/C++ that specifically focuses on testing scalar optimizations in compilers. SPE focuses on enumerating variable usage patterns in skeleton-based test synthesis [55]. GrayC designs five semantic-aware mutators specifically for creating more compilable programs [11]. Recent researchers have leveraged large language models (LLMs) in compiler testing: Ou et al. [42] proposed using LLMs to generate mutators, and Gao et al. [12] introduced a bracket-based masking strategy driven by historical Rust compiler bugs to synthesize valid test inputs, leading to the discovery of multiple confirmed bugs. Li et al. [30] use real-world code snippets to construct semantics-preserving programs for testing optimizing compilers.

In addition to optimization correctness,other aspects of compiler reliability such as incorrect debug information [31, 50], debugger toolchain bugs [54], and missed optimizations [48, 49], have also been extensively studied.

6.2 Compilation Space Exploration

Li et al. [28] introduced the concept of the compilation space exploration(CSE), which encompasses a comprehensive collection of Just-In-Time (JIT) compilation options. These options can be cross-validated to ensure the correctness of JIT compilation. To systematically explore the compilation space, Li et al. employed strategic mutations in the code structure of test programs. These mutations were thoughtfully designed to preserve the code's semantics while effectively triggering a wide range of JIT compilation choices. Chen et al. [6] conducted an empirical study on GCC and LLVM, highlighting the distinctive value of exploring the space of compiler optimizations. They proposed the COTest by considering both optimization settings and test cases to boost compiler testing.

Different from these approaches, our approach integrates not only the optimization options but also compilation options with the attributes to facilitate the exploration of the compilation space. Additionally, we introduce the insertion of attributes into the program, enabling a fine-grained and in-depth exploration of the compilation space. This integration of attributes with compilation options represents a novel approach, providing a more detailed and comprehensive examination of compiler behavior and reliability.

7 Conclusion

This study introduces a novel approach that utilizes attributes to guide the exploration of compilation space for compiler testing. To achieve this, we strategically incorporate attributes into existing test programs to generate mutated versions. Since attributes enable precise control over the compilation strategies applied to individual code segments, the mutated versions can be utilized to explore previously unexamined areas of compilers. We implement this concept as a prototype called ATLAS. Through extensive evaluation, we demonstrate the practical value of ATLAS, as it successfully detected a total of 73 bugs for GCC and LLVM. Notably, 58 of these bugs have been confirmed, and 17 have been resolved by the developers, underscoring the significant impact and efficacy of ATLAS in identifying compiler bugs.

Acknowledgments

We are grateful to the anonymous reviewers and our shepherd, Aravind Machiry, for their valuable and constructive feedback on this work. We are also indebted to the GCC and LLVM developers, particularly Jakub Jelinek, Richard Biener, Marek Polacek, and Martin Jambor for inspecting and fixing our reported bugs. This work is partially supported by the National Natural Science Foundation of China (Grants 62072194, 624B2067, 62172205, 62272214), the Jiangsu Natural Science Foundation under Grant BK20231402, the Collaborative Innovation Center of Novel Software Technology and Industrialization, and the Fundamental Research Funds for the Central Universities (XJ2024003301, XJ2025000601).

References

[1] Attribute specifier sequence. https://en.cppreference.com/w/cpp/language/attributes. Accessed October 20, 2024.

[2] Andrew W. Appel. *Modern Compiler Implementation in C*. Cambridge University Press, 1998.

[3] Junjie Chen, Yanwei Bai, Dan Hao, Yingfei Xiong, Hongyu Zhang, and Bing Xie. Learning to prioritize test programs for compiler testing. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*, pages 700–711. IEEE / ACM, 2017.

[4] Junjie Chen, Wenxiang Hu, Dan Hao, Yingfei Xiong, Hongyu Zhang, Lu Zhang, and Bing Xie. An empirical comparison of compiler testing techniques. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, pages 180–190. ACM, 2016.

[5] Junjie Chen, Jibesh Patra, Michael Pradel, Yingfei Xiong, Hongyu Zhang, Dan Hao, and Lu Zhang. A survey of compiler testing. *ACM Comput. Surv.*, 53(1):4:1–4:36, 2020.

[6] Junjie Chen and Chenyao Suo. Boosting compiler testing via compiler optimization exploration. *ACM Trans. Softw. Eng. Methodol.*, 31(4):72:1–72:33, 2022.

[7] Junjie Chen, Guancheng Wang, Dan Hao, Yingfei Xiong, Hongyu Zhang, and Lu Zhang. History-guided configuration diversification for compiler test-program generation. In *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019*, pages 305–316. IEEE, 2019.

[8] Junjie Chen, Guancheng Wang, Dan Hao, Yingfei Xiong, Hongyu Zhang, Lu Zhang, and Bing Xie. Coverage prediction for accelerating compiler testing. *IEEE Trans. Software Eng.*, 47(2):261–278, 2021.

[9] Tsong Yueh Chen, Shing-Chi Cheung, and Siu-Ming Yiu. Metamorphic testing: A new approach for generating next test cases. *CoRR*, abs/2002.12543, 2020.

[10] LLVM developers. LLVM function attributes. https://clang.llvm.org/docs/AttributeReference.html. Accessed October 20, 2024.

[11] Karine Even-Mendoza, Arindam Sharma, Alastair F. Donaldson, and Cristian Cadar. Grayc: Greybox fuzzing of compilers and analysers for C. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2023, Seattle, WA, USA, July 17-21, 2023*, pages 1219–1231. ACM, 2023.

[12] Hongyan Gao, Yibiao Yang, Maolin Sun, Jiangchang Wu, Yuming Zhou, and Baowen Xu. ClozeMaster: Fuzzing Rust Compiler by Harnessing LLMs for Infilling Masked Real Programs . In *2025 IEEE/ACM 47th International Conference on Software Engineering, ICSE 2025*, pages 712–712. IEEE / ACM, 2025.

[13] GCC developers. Declaring Attributes of Functions. https://gcc.gnu.org/onlinedocs/gcc/Function-Attributes.html. Accessed October 20, 2024.

[14] GCC developers. GCC development plan. https://gcc.gnu.org/develop.html. Accessed October 20, 2024.

[15] GCC developers. GCC function attributes. https://gcc.gnu.org/onlinedocs/gcc/Common-Function-Attributes.html. Accessed October 20, 2024.

[16] GCC developers. GCC releases. https://gcc.gnu.org/releases.html. Accessed October 20, 2024.

[17] GCC developers. Installing GCC: Testing. https://gcc.gnu.org/install/test.html. Accessed October 20, 2024.

[18] GCC developers. Options that control optimization. https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html. Accessed October 20, 2024.

[19] GCC developers. Specifying Attributes of Types. https://gcc.gnu.org/onlinedocs/gcc/Type-Attributes.html. Accessed October 20, 2024.

[20] GCC developers. Specifying Attributes of Variables. https://gcc.gnu.org/onlinedocs/gcc/Variable-Attributes.html. Accessed October 20, 2024.

[21] Alex Groce, Mohammad Amin Alipour, Chaoqiang Zhang, Yang Chen, and John Regehr. Cause reduction for quick testing. In *Seventh IEEE International Conference on Software Testing, Verification and Validation, ICST 2014, March 31 2014-April 4, 2014, Cleveland, Ohio, USA*, pages 243–252. IEEE Computer Society, 2014.

[22] Alex Groce, Mohammad Amin Alipour, Chaoqiang Zhang, Yang Chen, and John Regehr. Cause reduction: delta debugging, even without bugs. *Softw. Test. Verification Reliab.*, 26(1):40–68, 2016.

[23] Kenneth Hoste and Lieven Eeckhout. Cole: compiler optimization level exploration. In *Sixth International Symposium on Code Generation and Optimization (CGO 2008), April 5-9, 2008, Boston, MA, USA*, pages 165–174. ACM, 2008.

[24] Raphael Isemann, Cristiano Giuffrida, Herbert Bos, Erik van der Kouwe, and Klaus von Gleissenthall. Don't look UB: exposing sanitizer-eliding compiler optimizations. *Proc. ACM Program. Lang.*, 7(PLDI):907–927, 2023.

[25] Vu Le, Mehrdad Afshari, and Zhendong Su. Compiler validation via equivalence modulo inputs. In Michael F. P. O'Boyle and Keshav Pingali, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, pages 216–226. ACM, 2014.

[26] Vu Le, Chengnian Sun, and Zhendong Su. Finding deep compiler bugs via guided stochastic program mutation. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*, pages 386–399. ACM, 2015.

[27] Vu Le, Chengnian Sun, and Zhendong Su. Randomized stress-testing of link-time optimizers. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis, ISSTA 2015, Baltimore, MD, USA, July 12-17, 2015*, pages 327–337. ACM, 2015.

[28] Cong Li, Yanyan Jiang, Chang Xu, and Zhendong Su. Validating JIT compilers via compilation space exploration. In *Proceedings of the 29th Symposium on Operating Systems Principles, SOSP 2023, Koblenz, Germany, October 23-26, 2023*, pages 66–79. ACM, 2023.

[29] Shaohua Li and Zhendong Su. Ubfuzz: Finding bugs in sanitizer implementations. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1, ASPLOS 2024, La Jolla, CA, USA, 27 April 2024- 1 May 2024*, pages 435–449. ACM, 2024.

[30] Shaohua Li, Theodoros Theodoridis, and Zhendong Su. Boosting compiler testing by injecting real-world code. *Proc. ACM Program. Lang.*, 8(PLDI):223–245, 2024.

[31] Yuanbo Li, Shuo Ding, Qirun Zhang, and Davide Italiano. Debug information validation for optimized code. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, pages 1052–1065. ACM, 2020.

[32] Vsevolod Livinskii, Dmitry Babokin, and John Regehr. Random testing for C and C++ compilers with yarpgen. *Proc. ACM Program. Lang.*, 4(OOPSLA):196:1–196:25, 2020.

[33] Vsevolod Livinskii, Dmitry Babokin, and John Regehr. Fuzzing loop optimizations in compilers for C++ and data-parallel languages. *Proc. ACM Program. Lang.*, 7(PLDI):1826–1847, 2023.

[34] LLVM developers. Core Pipelines. https://llvm.org/docs/GlobalISel/Pipeline.html. Accessed October 20, 2024.

[35] LLVm developers. Installing gcc: Testing. https://llvm.org/docs/TestSuiteGuide.html. Accessed October 20, 2024.

[36] LLVM developers. Libtooling. https://clang.llvm.org/docs/LibTooling.html. Accessed October 20, 2024.

[37] LLVM developers. Llvm debian/ubuntu nightly packages. https://apt.llvm.org. Accessed October 20, 2024.

[38] LLVM developers. Llvm download page. https://releases.llvm.org/. Accessed October 20, 2024.

[39] LLVM developers. LLVM's Analysis and Transform Passes. https://llvm.org/docs/Passes.html. Accessed October 20, 2024.

[40] LLVM developers. Release 5.x commits. https://github.com/llvm/llvm-project/commits/release/5.x/. Accessed October 20, 2024.

[41] William M. McKeeman. Differential testing for software. *Digit. Tech. J.*, 10(1):100–107, 1998.

[42] Xianfei Ou, Cong Li, Yanyan Jiang, and Chang Xu. The mutators reloaded: Fuzzing compilers with large language model generated mutation operators. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4, ASPLOS 2024, Hilton La Jolla Torrey Pines, La Jolla, CA, USA, 27 April 2024 - 1 May 2024*, pages 298–312. ACM, 2024.

[43] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. Test-case reduction for C compiler bugs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, pages 335–346. ACM, 2012.

[44] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. Addresssanitizer: A fast address sanity checker. In *Proceedings of the 2012 USENIX Annual Technical Conference, USENIX ATC 2012, Boston, MA, USA, June 13-15, 2012*, pages 309–318. USENIX Association, 2012.

[45] Chengnian Sun, Vu Le, and Zhendong Su. Finding and analyzing compiler warning defects. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, pages 203–213. ACM, 2016.

[46] Chengnian Sun, Vu Le, and Zhendong Su. Finding compiler bugs via live code mutation. In Eelco Visser and Yannis Smaragdakis, editors, *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016*, pages 849–863. ACM, 2016.

[47] Chengnian Sun, Vu Le, Qirun Zhang, and Zhendong Su. Toward understanding compiler bugs in GCC and LLVM. In *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016*, pages 294–305. ACM, 2016.

[48] Theodoros Theodoridis, Manuel Rigger, and Zhendong Su. Finding missed optimizations through the lens of dead code elimination. In *ASPLOS '22: 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, 28 February 2022 - 4 March 2022*, pages 697–709. ACM, 2022.

[49] Theodoros Theodoridis and Zhendong Su. Refined input, degraded output: The counterintuitive world of compiler behavior. *Proc. ACM Program. Lang.*, 8(PLDI):671–691, 2024.

[50] Theodore Luo Wang, Yongqiang Tian, Yiwen Dong, Zhenyang Xu, and Chengnian Sun. Compilation consistency modulo debug information. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2023, Vancouver, BC, Canada, March 25-29, 2023*, pages 146–158. ACM, 2023.

[51] Jiangchang Wu, Yibiao Yang, and Yuming Zhou. Boosting compiler testing via eliminating test programs with long-execution-time. In *IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2023, Taipa, Macao, March 21-24, 2023*, pages 593–603. IEEE, 2023.

[52] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, pages 283–294. ACM, 2011.

[53] Yibiao Yang, Qingyang Li, Maolin Sun, Jing Yang, Jiangchang Wu, and Yuming Zhou. Isolating compiler faults through differentiated compilation configurations. *IEEE Transactions on Software Engineering*, pages 1–16, 2025.

[54] Yibiao Yang, Maolin Sun, Jiangchang Wu, Qingyang Li, and Yuming Zhou. Debugger toolchain validation via cross-level debugging. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1, ASPLOS 2025, Rotterdam, The Netherlands, 30 March 2025 - 3 April 2025*, pages 280–294. ACM, 2025.

[55] Qirun Zhang, Chengnian Sun, and Zhendong Su. Skeletal program enumeration for rigorous compiler testing. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, pages 347–361. ACM, 2017.

A Artifact Appendix

Abstract

The artifact contains the code and datasets we used for our experiments, as well as scripts to generate the numbers and tables of our evaluation. Specifically, it includes (a) links and bug-triggering test cases of each reported bug and (b) detailed instruction documentation for using ATLAS. Everything is packaged and pre-built as a docker image.

Scope

The artifact allows validation of the identified bugs shown in Table 1. Additionally, Table 2, Table 3, and Figure 7 are based on a manual analysis of the identified bugs.

Contents

This artifact consists of the following items:

- **The AST Parsing Tool.** This artifact provides the tool to parse the seed programs.
- **Attribute Insertion Tool.** This artifact provides scripts that can be used to insert attributes into the seed programs

- **Documentation.** This artifact provides step-by-step instructions on setting up and using the Docker image, along with usage guidance for the tools above.

Hosting

The artifact is archived on Zenodo and can be downloaded at the following DOI: https://doi.org/10.5281/zenodo.15375789

Requirements

For optimal performance, we recommend running the full evaluation on a machine with the following specifications:

- **CPU:** 16 cores or more
- **Memory:** At least 32 GB
- **Disk Space:** 20 GB free

A.1 Installation

The Docker image is provided in a pre-configured format, obviating the need for any installation. The following commands can be employed to extract the artifact archive and import it into Docker:

```
$ gunzip -c atlas.tar.gz > atlas.tar
$ docker load -i atlas.tar
$ docker run -it -name atlas
compiler_testing:atlas /bin/bash
```

ARTIFACT EVALUATED usenix ASSOCIATION AVAILABLE
ARTIFACT EVALUATED usenix ASSOCIATION FUNCTIONAL
ARTIFACT EVALUATED usenix ASSOCIATION REPRODUCED

Understanding and Detecting Fail-Slow Hardware Failure Bugs in Cloud Systems

Gen Dong, Yu Hua, Yongle Zhang*, Zhangyu Chen, Menglei Chen
Huazhong University of Science and Technology
**Purdue University*
Corresponding Author: Yu Hua (csyhua@hust.edu.cn)

Abstract

Fail-slow hardwares are still running and functional, but in a degraded mode, thus slower than their expected performance. Bugs triggered by fail-slow hardwares cause severe cloud system failures. Existing testing tools fail to efficiently detect these bugs due to overlooking their characteristics. In order to address this problem, this paper provides a bug study that analyzes 48 real-world fail-slow hardware failures from typical cloud systems. We observe that (1) fail-slow hardwares make high-level software components vulnerable, including synchronized and timeout mechanisms; (2) the fine granularity of fail-slow hardwares is necessary to trigger these bugs. Based on these two observations, we propose Sieve, a fault injection testing framework for detecting fail-slow hardware failure bugs. Sieve statically analyzes target system codes to identify synchronized and timeout-protected I/O operations as candidate fault points and instruments hooks before candidate fault points to enable fail-slow hardware injection. To efficiently explore candidate fault points, Sieve adopts grouping and context-sensitive injection strategies. We have applied Sieve to three widely deployed cloud systems, i.e., ZooKeeper, Kafka, and HDFS. Sieve has detected six unknown bugs, two of which have been confirmed.

1 Introduction

Cloud systems have become the backbone of numerous modern applications [29, 57, 61, 65]. However, fault-triggered failures in cloud systems cause severe loss of customer satisfaction and revenues for service providers [9, 19, 53]. To achieve high reliability, cloud systems need to correctly handle various faults [1, 10, 37, 42]. Based on the scope of faults, we classify them into two categories: **coarse-grained** and **fine-grained** faults. Specifically, when only considering I/O operations in a cloud system, let D=$\{D_1,D_2,...,D_n\}$ represent the set of disk I/O operations and N=$\{N_1,N_2,...,N_m\}$ represent the set of network I/O operations. Coarse-grained faults affect all associated I/O operations, e.g., the I/O operations in D fail in fail-stop disks [46] and those in $D \cup N$ fail in node crashes [16, 42]. Fine-grained faults affect a subset of associated I/O operations, e.g., fail-slow Network Interface Cards (NICs) [22] slow down N_{sub} ($N_{sub} \subset N$). Coarse-grained faults have been well studied and handled by existing well-designed fault tolerance mechanisms [5, 18, 52, 56]. However, emerging fine-grained faults propose new challenges to the reliability of cloud systems.

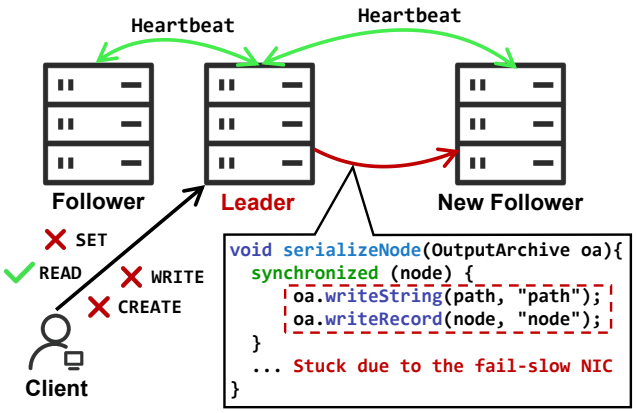


Figure 1: A production ZooKeeper failure due to the fail-slow NIC [11]. The writeRecord operation becomes slow within the synchronized block. The update-type requests are blocked. Developers fixed the bug by copying the node object and serializing the copy out of the synchronization.

Recently, fail-slow hardwares [22, 44] received an increasing amount of attention, which deliver lower-than-expected performance and cause fine-grained faults. For example, the throughput of a 1-Gbps NIC might suddenly drop by several orders of magnitude to 1 Kbps due to partial buffer corruption and retransmission [12], which slows down partial network I/O operations. Lu et al. [44] indicate that the fail-slow NVMe SSD has become a widespread and severe problem. On average, 1.41% of NVMe SSDs (over one million NVMe SSDs) are infected within four months of monitoring. Moreover,

fail-slow hardware incidents always consume hours or even months to detect. Only 1% of the cases are detected in minutes [22]. As a consequence, once fail-slow hardwares bring cloud systems down, it requires large human efforts to detect the root cause. For example, several AWS services become unavailable due to unexpected high network latency [53], which takes developers nine hours to detect the root cause. Figure 1 presents another real-world fail-slow hardware failure from ZooKeeper [11] which is a widely used distributed coordination to tolerate leader and follower crashes. Specifically, an entire cluster becomes a near-freeze status. `write()` and `create()` requests fail, but `read()` requests can be successfully processed. The developers at PagerDuty [50] spend five months to diagnose this failure. In this paper, we refer to the fault caused by the fail-slow hardware as the **FSH fault** and the bug triggered by the FSH fault as the **FSH bug**. We define ***fail-slow hardware failures*** as the failures caused by FSH bugs and refer to these failures as **FSH failures**.

To prevent the FSH fault from breaking cloud systems at runtime, it is essential to develop an FSH fault injection testing framework. Fault injection testing [1, 6, 14, 16, 21, 31, 32, 42, 48] is a commonly used technique for detecting fault-triggered bugs. There are enormous efforts in developing various fault injection testing frameworks. However, existing schemes fail to efficiently detect FSH bugs due to overlooking the characteristics of FSH failures. To address this problem, we conduct a study of 48 real-world FSH failure cases from five large-scale and open-source systems. Furthermore, we observe that there are two key characteristics in efficiently detecting FSH failures like the example in Figure 1.

1) The vulnerable synchronized and timeout mechanisms cause all studied FSH failures. For example, synchronized tasks slowed down by fail-slow hardwares block other tasks for an uncertain time; data race can occur between the timeout task caused by the fail-slow hardware and the corresponding timeout handler with simple timing constraints. This characteristic can significantly prune the huge fault injection space. However, existing bug studies overlook this characteristic since they focus on either the fail-slow software or hardware. Unlike them, our bug study comprehensively analyzes how the fail-slow hardware affects the software.

2) The fine granularity of FSH faults is necessary to trigger FSH failures. One of the main reasons is that fine-grained FSH faults are relatively easy to escape the detection of internal checkers. For example, as shown in Figure 1, there are two network I/O operations, i.e., `serializeNode` and `Heartbeat`. The fail-slow NIC only slows down `serializeNode` so that the heartbeat thread in the leader can maintain the liveness of the cluster. Hence, the entire cluster becomes a *near-freeze status*. In this scenario, the fail-slow NIC escapes the detection of the heartbeat threads in the followers. The fine-grained fault injection is necessary to trigger such failure. Specifically, if a fail-stop NIC is injected in the leader, these two network I/O operations are affected. Subsequently, the heartbeat threads in the followers will detect this coarse-grained fault and start a new leader election, which drives the cluster into a healthy status. Most existing fault injection testing frameworks focus on coarse-grained faults. Although these schemes are enough to detect crash recovery bugs or distributed protocol bugs, they cannot effectively detect FSH bugs.

Inspired by these two key characteristics, we propose Sieve, a fault injection testing framework designed to detect FSH bugs in cloud systems. Sieve treats synchronized and timeout-protected I/O operations as candidate fault points. To identify candidate fault points, we investigate various synchronized and timeout mechanisms in cloud systems and conclude four general patterns from them. Based on these four patterns, Sieve identifies I/O operations in synchronized and timeout-protected domains through lightweight static analysis at the code instruction level. Besides, Sieve automatically instruments hooks before candidate fault points to precisely control the fine-grained FSH fault injection.

To efficiently explore these candidate fault points, Sieve adopts grouping and context-sensitive injection strategies. Specifically, Sieve groups fault points within the same basic block[1]. Because fault points within the same basic block are always under the same high-level system states (e.g., leader election and snapshot synchronization) and protected by the same fault-tolerance handler, which indicates these fault points trigger similar scenarios. Besides, Sieve leverages the context-sensitive injection strategy to (1) avoid exploring the same contexts of fault points and (2) find buggy contexts of the same fault points. During testing, the hook sends the fault injection query to the injection controller. The controller checks the runtime context of the fault point, e.g., call stack and thread ID, and decides whether to inject the fault.

We have implemented a prototype of Sieve and evaluated it on three large-scale cloud systems, i.e., ZooKeeper [70], Kafka [33], and HDFS [2]. Moreover, Sieve has detected six unknown bugs in these systems. Besides, we have released Sieve (including bug study, scripts for reproducibility, and source codes) for public use at https://github.com/RabbitDong-on/Sieve.

In summary, this paper makes the following contributions:

- We conduct a study of 48 real-world FSH failure cases from five cloud systems. Our bug study helps developers to comprehensively understand the characteristics of FSH failures.

- We propose Sieve, a novel approach that enables fine-grained FSH fault injection for detecting FSH bugs. Based on our bug study, Sieve analyzes synchronized and timeout-protected I/O operations as candidate fault points. To efficiently explore these candidate fault points, Sieve adopts grouping and context-sensitive injection strategies.

[1] A sequence of consecutive instructions without any branch instructions.

Table 1: The numbers of FSH Failures.

ZooKeeper	HDFS	HBase	MapReduce	Cassandra
11	18	10	4	5

- We implement a prototype of Sieve and evaluate it on three large-scale real-world cloud systems. Sieve has detected six unknown bugs, two of which have been confirmed.

2 Methodology

To understand the FSH failure, we studied 48 real-world failures in five popular cloud systems shown in Table 1. We leverage the search tool in JIRA [54] to identify reports related to fail-slow hardwares. First, we search reports using the following keywords: "slow disk", "slow storage", "slow network", "slow NIC", and "slow switch". Second, we exclude the issues that have priority of "Minor", "Trivial" and "Low". Third, we identify the issues that are indeed related to fail-slow hardwares. Specifically, for each issue, we carefully read the failure report to obtain an overview of the failure. If the unit test is provided, we run the unit test to reproduce the failure. The unit test provides simplified buggy logic, which is an important guideline for us to read source codes. If the unit test is absent, we directly read source codes based on conversations between the bug reporter and project maintainers. We can identify the root cause in both scenarios. If the root cause is related to the delay and the issue report explicitly mentions the delay is caused by the fail-slow hardware, we preserve this issue.

Threats to Validity. Like all characteristic studies, the results of our study should be interpreted with the following limitations.

Representativeness of the selected systems. The selected systems are diverse, widely-used and open-source: ZooKeeper [70] is a distributed coordination service; MapReduce [58], HDFS [2], and HBase [23] are the cores of the dominant Hadoop data analytics platforms; Cassandra [4] is a highly available peer-to-peer NoSQL database. However, without accurate market information, it is difficult to conclude whether we have chosen the most widely-used cloud systems. Besides, closed-source cloud systems could have different characteristics.

Limitations of the filtering criteria. We clarify that this paper focuses on two types of fail-slow hardwares including the fail-slow storage and network devices. The main reason is that it is relatively easy for developers to confirm the impact of the fail-slow storage and network devices, i.e., slow I/O operations. It is still possible to miss the FSH failures whose issue reports do not contain the selected keywords. The fail-slow hardware is difficult to identify [22]. Developers may not be sure whether the delay is caused by the fail-slow hardware.

Table 2: FSH Failure Symptoms.

Symptom	**%**
Node service unavailable	58.3
Data unavailability	10.4
Data inconsistency	10.4
Job failure	8.3
Performance degradation	6.3
Client stuck	4.2
Node crash	2.1

Hence, the issue report possibly misses the selected keywords. We did try other keywords like "slow", and found that the resulting issue reports contain too many false positives, i.e., non-FSH failures. It is impractical to reduce false positives one by one.

Observer errors. To minimize the possibility of observer errors, each failure is investigated by two inspectors with the same criteria. Any disagreement is discussed in the end to reach a consensus.

3 Understanding FSH Failures

In this section, we present some findings and implications by analyzing collected FSH failures.

3.1 Findings

Finding 1: *Over half (58.3%) of FSH failures cause node service to be unavailable.*

Table 2 shows that FSH failures exhibit various failure symptoms. Over half of FSH failures cause certain software functionality or the entire node to be unavailability. In some cases, even a normal node is removed from the cluster. For example, in HDFS-9178 [27], the upstream datanode is stuck in the fail-slow disk. Hence, the downstream node timeouts when reading the packet from the upstream datanode. Then, the upstream datanode sends an ACK to the client and sets the downstream datanode status to `ERROR`. Finally, the client excludes the downstream datanode from the pipeline, even though the upstream datanode is the abnormal one.

Finding 2: *20.8% of FSH failures are silent (including data unavailability and inconsistency).*

These failures are difficult to detect without the correctness specification. For example, in HBase-26195 [24], synchronizing Hlog to HDFS timeouts due to the fail-slow hardware. The client receives an exception and rolls back the data protected by the Hlog. However, this procedure only rolls back the data in the primary node leading to data inconsistency between the primary and replicated nodes. Besides, internal checkers in HBase do not raise any explicit alarms in system

Table 3: FSH Failure Root Causes.

Root cause		%
Non-concurrency		81.2
◇ Indefinite blocking	41.6	
◇ Buggy internal checker	35.4	
○ Buggy error checking	20.8	
○ Buggy error handling	14.6	
◇ Infinite loop	2.1	
◇ Logic error	2.1	
Concurrency		18.8
◇ Data race	16.7	
◇ Deadlock	2.1	

logs or immediately stop the system, which makes this failure detection difficult.

Finding 3: *The root causes of studied failures are diverse. The top three (total 93.7%) root causes are indefinite blocking, buggy internal checker, and data race.*

Indefinite blocking means that synchronized tasks slowed down by fail-slow hardwares block other tasks for an uncertain time. In Figure 1, `writeRecord` within the `synchronized` block becomes extremely slow due to the fail-slow NIC. This synchronized `writeRecord` blocks all following write requests from clients. The fine-grained fail-slow NIC escapes the detection of `Heartbeat`, which is critical to trigger this failure. Buggy internal checker incorrectly handles timeout tasks caused by fail-slow hardwares. For example, in HDFS-5522 [26], the delay caused by the fail-slow disk is treated as a network error. Hence, `checkDiskError` is not invoked, which fails a lot of client requests. Data race occurs between timeout tasks caused by fail-slow hardwares and timeout handlers. Other root causes include infinite loop, logic error, and deadlock.

Finding 4: *Buggy internal checker fails to correctly handle timeout tasks.*

Buggy internal checker includes buggy error checking and error handling [67], which are confused by the impact of fine-grained FSH faults. Specifically, buggy error checking fails to figure out sources of FSH faults, such as fail-slow disks in HDFS-5522 [26]. Even though error checking is correct, error handling may also incorrectly handle FSH faults. For example, in MapReduce-1800 [47], the reducer fails to fetch the mapper's output due to the reducer-side fail-slow NIC. However, the reducer can still talk to the job tracker(JT). When a sufficient number of reducers fail, the normal mapper is blacklisted by the JT. In this example, although error checking finds the slow network, error handling incorrectly handles this fault. The fine-grained FSH fault is key to trigger this failure. Specifically, if a network partition occurs, the reducer cannot talk to the JT. Hence, the normal mapper is not blacklisted.

Table 4: The percentage of each involved event.

Event type	%
Client/Server connection	12.5
Client request	83.7
One fail-slow hardware	89.6
Multiple fail-slow hardwares	10.4
Node reboot/join cluster	4.2

Finding 5: *Data race can occur between the timeout task caused by the fail-slow hardware and the corresponding timeout handler with simple timing constraints.*

The task slowed down by the fail-slow hardware triggers the timeout mechanism. Moreover, the invoked timeout handler does not end the slow task. Therefore, if the timeout handler finishes before the slow task, this task will corrupt the data written by the timeout handler. For example, in ZooKeeper-710 [72], the client connects with the follower that encounters the fail-slow NIC. This connection request triggers the timeout mechanism. The invoked timeout handler sends another connection request to the leader. After successfully connecting with the leader, the former connection request is processed by the follower and the renewal request is sent to the leader. The renewal request overwrites the metadata of the current connection. As a result, the leader still connects with the client, but cannot process the client request. This data race occurs without the complex thread interleaving control. Besides, the fine granularity of the fail-slow hardware is necessary to trigger data race. Specifically, if coarse-grained faults, e.g., node crashes and network partitions, occur in the follower, the renewal request will not be sent to the leader. Hence, data race does not happen in this scenario. Additionally, our bug study shows that the buggy timeout mechanism is a new root cause for concurrency bugs.

Finding 6: *Most (89.6%) of FSH failures are caused by one fail-slow hardware.*

As shown in Table 4, the triggering conditions of most FSH failures do not require multiple fail-slow hardwares and complex inputs. For example, in HDFS-5341 [25], `DirectoryScanner` encounters a fail-slow disk when scanning the dataset with a lock. As a result, `DirectoryScanner` blocks the heartbeat and all DataXceiver threads that acquire the lock. This example only requires simple client requests as inputs.

3.2 Implications

Overall, our bug study reveals that the FSH failure is a severe problem in cloud systems (Table 2). Most (73.5%) of studied failures were reported in the last decade. Moreover, over half (58.3%) of FSH failures cause node service to be unavailable (Finding 1). Even though existing in-production

detectors [28, 39, 43, 51] can detect part of FSH failures, the detected failures already cause damages, e.g., node unavailability and data inconsistency [28, 39, 43, 51]. Hence, it is necessary to design an FSH fault injection testing framework to detect FSH bugs before releasing production. The fault injection space explosion is a notorious problem in fault injection techniques [1, 6, 14, 16, 21, 31, 32, 42, 48]. However, our bug study reveals that synchronized and timeout-protected I/O operations are error-prone, which significantly reduces the fault injection space. Besides, most FSH failures are caused by one fail-slow hardware (Finding 6). Hence, the fault injection space does not increase exponentially due to the combinations of different fault points, which indicates that the fault injection testing framework can efficiently explore the fault injection space without complex fault injection strategies.

There are four points to avoid most FSH failures:

- Do not perform I/O operations within synchronization structures. For example, the solution of the example in Figure 1 is to copy and serialize data out of the `synchronized` block.
- Use the fine-grained lock to accurately synchronize I/O operations.
- Collect more information from other components of cloud systems to figure out sources of FSH faults, e.g., the delay from upstream datanode in HDFS-9178 [27], and take suitable actions.
- Avoid data race between the timeout task and timeout handler. Developers can use timestamps to avoid buggy execution order.

4 The Sieve Design

In this section, we first discuss Sieve's fault model (§ 4.1) and then describe the workflow (§ 4.2) and main components: Fault Point Analysis (§ 4.3), Injection Controller (§ 4.4), Workload Driver (§ 4.5), and Failure Checkers (§ 4.6).

4.1 Fault Model

Fail-slow hardwares exhibit two types of fine-grained faults. The first type slows down partial I/O operations. The second type incurs exceptions from partial I/O operations. In this paper, we only focus on the first type and leave the second type as future work.

Cloud systems are designed to tolerate various faults, but their fault-tolerance levels are limited. If we inject multiple faults, the accumulated side effects possibly break cloud systems as expected. Moreover, one fault is enough to trigger most FSH failures (Finding 6). Hence, Sieve simulates the first type by injecting one delay for each test run.

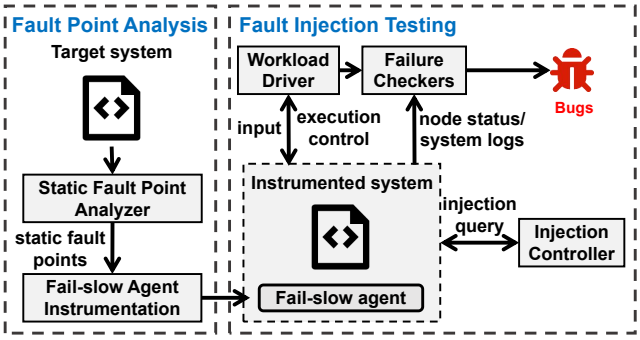


Figure 2: The workflow of Sieve.

4.2 Workflow

In a nutshell, Sieve identifies all fault points and then performs fault injection testing at each fault point. Figure 2 presents the workflow of Sieve, which consists of two phases. The first phase (left half of Figure 2) leverages the static analysis to identify fault points and instruments fail-slow agents within the cloud system to query fault injection decisions. The second phase (right half of Figure 2) treats the fault injection testing as a series of test runs. For each test run, Sieve leverages the workload driver to exercise the cloud system. When a fail-slow agent is executed, it sends an RPC query to the injection controller. The controller decides whether to inject the fault based on the history injection record. If the fail-slow agent receives a positive reply, it injects a delay into the cloud system. During this test run, failure checkers detect abnormal system behaviors, e.g., system crashes.

4.3 Fault Point Analysis

Cloud systems contain a large number of fault points affected by fail-slow hardwares. Moreover, most fault points can be tolerated by cloud systems [6, 16, 39, 42]. Therefore, it is impractical to exhaustively explore all fault points. Based on our bug study, we observe that synchronized and timeout-protected I/O operations are error-prone. Hence, our fault point analysis focuses on these two types of fault points, which can reduce the fault injection space.

Identify the Synchronized I/O Operation. In general, synchronized mechanisms have two effects: (1) avoid multiple threads to simultaneously access critical data; (2) control different tasks to execute in a certain order. There are two corresponding patterns of synchronized mechanisms. The first pattern creates a critical region, e.g., Lines 2-10 in Figure 3. I/O operations (I/O_{1-3}) in the critical region are considered to be synchronized. To identify these synchronized I/O operations, Sieve needs to find the critical region. The critical region is wrapped by entry and exit points at the instruction level. It is non-trivial to identify the critical region by matching the entry point to the correct exit point, since branch and return instructions introduce more than one exit point. For

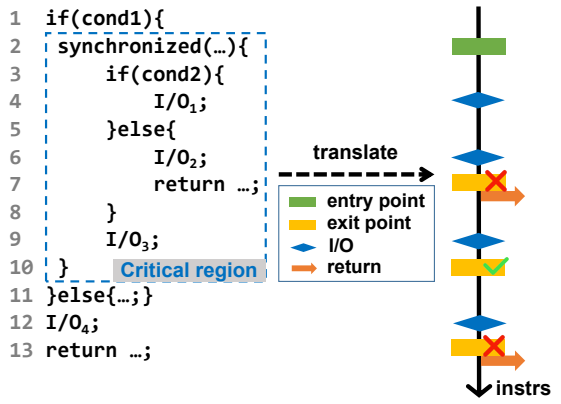


Figure 3: The left part shows an example of creating a critical region (Lines 2-10). The right part shows low-level instructions, which only preserve I/O, return, entry and exit points.

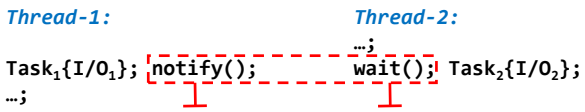


Figure 4: An example of controlling the execution order of $Task_{1,2}$ via the barrier.

example, in Figure 3, there are three exit points. If Sieve matches the entry point to the first or last exit points, it will miss a synchronized I/O operation (I/O_3) or incorrectly treat a non-synchronized I/O operation (I/O_4) to be synchronized.

To match the correct exit point, the intuitive method is to (1) understand how the programming language designs the interaction among the critical region, branch, and return instructions (2) and design a search algorithm that leverages former understanding. However, the first step is difficult and requires large human efforts. Hence, we try another practical method. Specifically, we analyze different combinations of the critical region, branch, and return instructions case by case. In this process, we observe that the correct exit point is the last not followed by the return instruction, i.e., the second exit point in Figure 3. Based on this observation, Sieve can correctly identify the critical region and synchronized I/O operations in practice. Specifically, Sieve traverses the call graph[2] twice. (1) If encountering an entry point, Sieve pushes this entry point into a stack. Subsequently, if encountering an exit point not followed by the return instruction, Sieve pops the top non-entry point from the stack and pushes this entry point into the stack. If the encountered exit point is followed by the return instruction, Sieve just ignores this exit point. Finally, all entry and matched exit points stored in the stack confirm all critical regions. (2) Sieve identifies I/O operations in critical regions.

[2]A directed acyclic graph. Its vertex represents a function and its edge means a function call relation

The second pattern creates a hard barrier that makes sure one task is executed before the other, e.g., `notify()` and `wait()` create a barrier in Figure 4. I/O operations (I/O_1) before the barrier are considered to be synchronized. Identifying these synchronized I/O operations is trivial. Sieve traverses the call graph, finds barriers, and identifies I/O operations before these barriers. Timeout mechanisms also have this pattern that creates a soft barrier (using `notify()` and `wait(timeout value)`) that means the latter task ($Task_2$) can timeout and invoke timeout handler to process the former task ($Task_1$). Sieve uses the same method to deal with this pattern of timeout mechanisms.

Identify the Timeout-protected I/O Operation. Timeout mechanisms are commonly used to handle unexpected faults in cloud systems. After investigating several cloud systems, we conclude two general patterns of timeout mechanisms. One has been discussed in § 4.3.

The other is related to get-time functions, e.g., `nanoTime` and `currentTimeMillis` in Java. Specifically, cloud systems record **startTime** and **endTime** assigned by get-time functions and calculate **elapsedTime = endTime - startTime** to obtain the execution time of the task. If elapsedTime exceeds the predefined timeout value, the timeout handler is invoked to process this abnormal task. We define the timeout scope as the scope between assignments of startTime and endTime. I/O operations in the timeout scope are considered to be timeout-protected. To identify these timeout-protected I/O operations, Sieve needs to match startTime to the corresponding endTime, which is non-trivial. In real-world cloud systems, there are many {**startTime**,**endTime**} pairs interleaving with each other, e.g., $\{startTime_1, endTime_1\}$ interleaves with $\{startTime_2, endTime_2\}$ in Figure 5. Moreover, most {startTime, endTime} pairs usually have irregular variable names. Hence, we cannot distinguish which endTime ($endTime_{1,2}$) is matched to $startTime_1$. When reviewing this pattern, we note that startTime and the matched endTime must appear in the same formula: **elapsedTime = endTime - startTime**. Based on this observation, Sieve can correctly identify timeout-protected I/O operations. Specifically, Sieve (1) collects all variables ($startTime_{1,2}$ and $endTime_{1,2}$) assigned by get-time functions as a get-time set. (2) then checks all expressions whose forms are **A=B-C**. If two operators in the expression are in the get-time set and the result of this expression is used in `If` expression, this expression is the target formula ($elapsedTime_1 = endTime_1 - startTime_1$). The two operators ($startTime_1$ and $endTime_1$) in this expression correctly confirm a timeout scope ($timeoutscope_1$). (3) finally identifies all I/O operations in the timeout scope.

Instrument the Fail-Slow Agent. To enable fine-grained fault injection, Sieve instruments fail-slow agents before candidate fault points, i.e., synchronized and timeout-protected I/O operations. When the fail-slow agent is reached at runtime, it sends a query to the injection controller, which includes the current call stack, instrumented function information, instru-

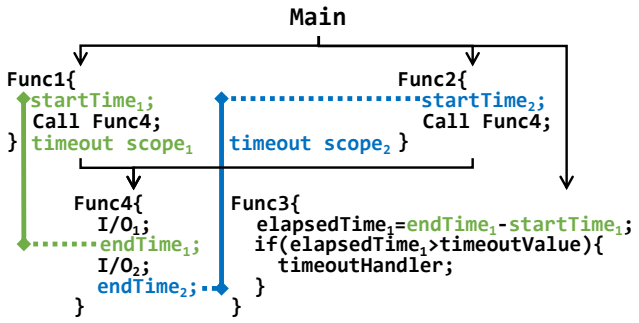


Figure 5: An example of identifying timeout-protected I/O operations.

mented position, and thread ID. If the reply is positive, the fail-slow agent invokes `Thread.sleep`. Additionally, Sieve instruments some hooks to obtain timeout values.

4.4 Injection Controller

In the fault injection phase, Sieve injects one fault for each test run. In each test run, the injection controller receives many injection requests from fail-slow agents, collects the information from injection requests, and decides on the fault injection. Finally, only one of the agents obtains a positive reply. To efficiently explore candidate fault points, Sieve implements grouping and context-sensitive injection strategies.

Grouping Injection Strategy. Through the fault point analysis, Sieve has pruned many fault points with a low possibility of triggering FSH bugs. However, there are still a large number of fault injection choices. It is inefficient to exhaustively explore all choices. We note many fault injections explore similar scenarios when these fault points are within the same basic block. Because different I/O operations within the same basic block are always under the same high-level system states (e.g., leader election and snapshot synchronization) and protected by the same fault-tolerance handler. To further reduce the fault injection space, Sieve groups fault points within the same basic block and selects the last fault point as the representative of the group. The fault point analyzer implements this grouping strategy by instrumenting the fail-slow agent before the last fault point within the same basic block.

Context-sensitive Injection Strategy. In addition to grouping fault points, Sieve implements a context-sensitive strategy (listed in Algorithm 1) to (1) find the buggy context of the fault point (2) and avoid repeatedly injecting faults at the same point under the same context. Specifically, Sieve collects the call stack of the fault point carried by the injection request (Line 20). When an injection request is received, Sieve checks whether the corresponding fault point has already been explored. If the fault point is not explored, Sieve grants this fault injection. Otherwise, Sieve compares call stacks of previous fault injections with that of current fault injection (Lines 15-19). If the current call stack is unique, Sieve injects one fault at this point.

Algorithm 1: Fault injection strategy

```
1  Function scheduleFaultInjection(request):
2      if !checkRequest(request) then
3          return false;
4      end
5      fault ← getFault(request);
6      if fault.type=SYNC then
           delayTime ← 5 * 60 * 1000 ;
7      else
           delayTime ← 3 * getTimeOutValue(fault)/2 ;
8      grant(request,delayTime);
9  Function checkRequest(request):
10     if isInjected then
11         return false;
12     end
13     curContext ← getContext(request);
14     curFault ← getFault(request);
15     foreach preContext ∈ historyInfo.contexts() and
           preFault ∈ historyInfo.faults() do
16         if preFault=curFault and
               preContext.contain(curContext) then
17             return false;
18         end
19     end
20     record(historyInfo,curContext,curFault);
21     return true;
```

Fault Impact. Except for injection strategies, Sieve needs to determine how long the delay lasts. Specifically, Sieve checks the type of the fault point after receiving the injection request (Line 5). If the injection request comes from the synchronized I/O operation, Sieve uses 5 minutes as the default delay which is enough to cause indefinite blocking (Line 6). Hence, the gray failure checker in Sieve can detect this bug. Meanwhile, this delay duration does not significantly extend the evaluation time.

For the timeout-protected I/O operation, the delay duration is two-thirds of the timeout value (Line 7). In this way, Sieve not only triggers timeout mechanisms to mislead internal checkers but also reorders timeout tasks and timeout handlers. Sieve obtains timeout values by analyzing two general patterns shown in § 4.3 and instrumenting hooks. When hooks are reached at runtime, they send timeout values to the injection controller.

4.5 Workload Driver

Sieve leverages the workload driver to exercise a target cloud system. The workload driver can use various sources ranging from simple unit tests to carefully-crafted test cases as workloads. For each cloud system, we either implement several common workloads or select several existing test cases as

workloads shown in Table 5.

The workflow of the workload driver is as follows. First, the workload driver creates multiple clients and each client connects with one system node. Hence, if the client encounters an exception, it can send an RPC request to failure checkers reporting the corresponding system node is suspicious. Second, the client executes the workload. If the client finishes the workload, it sends an RPC request to failure checkers reporting the workload progress. Finally, the workload driver restarts the target system for each test run to avoid the impact of previous injected faults.

4.6 Failure Checkers

To determine whether a bug occurs in cloud systems, Sieve provides two checkers:

- **Log error checker.** It scans the execution log of each system node to check whether there are FATAL, ERROR, and WARN entries or uncommon exceptions.
- **Gray failure checker.** It is a simplified version of Panorama [28] to identify whether differential observability exists. This checker marks the test run suspicious if (1) the system's internal checker indicates a node is healthy, but the node's client reports errors. Specifically, this checker receives an RPC request that reports errors from the client, while obtaining a healthy status of the corresponding node by existing tools, e.g., `./zkServer.sh status` in ZooKeeper. (2) a subset of clients do not finish their workloads. Specifically, this checker does not receive RPC requests that report the workload progress from a subset of clients.

As a testing framework, developers can flexibly add specific checkers to assert system properties they care about, such as performance bug [35] and inconsistency [40].

5 Implementation

We have implemented Sieve in Java with around 8,100 SLOC for core components. The fault point analyzer is built on top of Soot, a Java program analysis framework [59]. Sieve instruments cloud systems using Javassist, a Java bytecode instrumentation toolkit [30]. The injection controller is designed in a client-server architecture via Java RMI for RPCs.

I/O Operation Identification. Sieve identifies all I/O operations by statically analyzing cloud systems. If the function is implemented by general I/O packages, it is treated as an I/O operation. Sieve analyzes five general I/O packages, including `java.io`, `java.nio`, `java.net`, `javax.net` and `io.netty`. Additionally, some cloud systems implement their customized I/O operations, e.g., serialize/deserialize APIs in class Record of ZooKeeper. Sieve also identifies such I/O operations.

Timeout Value Collection. Sieve analyzes two general patterns of timeout mechanisms presented in § 4.3. For the first pattern, Sieve instruments hooks before soft barriers, e.g., `wait` in Java. The hooks intercept barriers and capture their parameters. For the second pattern, Sieve finds elapsedTime through the taint analysis. Sieve instruments a hook before `CMP` instruction that contains elapsedTime. The hook captures the other operator of `CMP` instruction. The captured parameters and operators are timeout values. Finally, the hooks send timeout values to the injection controller.

Table 5: The evaluated systems.

System	Release	Workload
ZooKeeper (ZK)	3.9.0	Create/read/update/delete znode
Kafka (KA)	3.6.0	Producer/consumer performance test
HDFS	3.3.6	Read/write/move/put file

6 Evaluation

Our evaluation aims to answer two questions: (1) How effective is Sieve in detecting bugs? (2) How does Sieve compare with state-of-the-art fault injection approaches?

Evaluated Systems. We evaluated Sieve by using three widely used and open-source cloud systems in Table 5. Note that Kafka [33] is not included in our bug study, which is a popular stream-processing service. To prove that Sieve is not specific to the five studied systems, we selected Kafka as a benchmark. All target systems are the latest versions during the experiments.

Setup. We apply default configurations to all cloud systems and deploy a cluster of cloud systems on a single physical machine using dockers. The physical machine contains two Intel Xeon Gold 6230R CPUs, which include 52 cores, and 192GB DRAM running Ubuntu18.04. In the evaluation, the cluster consists of either three nodes for ZooKeeper and Kafka or four nodes for HDFS. The fault injection experiment for each system consists of 2000 test runs. The time of each test run varies depending on how the system reacts to the injected faults and whether it fails early or not. The experiment time for the three systems is 20.9 hrs, 24.9 hrs, and 29 hrs respectively.

6.1 Effectiveness of FSH Bug Detection

6.1.1 Methodology

We apply Sieve to the target systems and check whether Sieve can detect unknown bugs and reproduce studied bugs.

6.1.2 Detecting Unknown Bugs

As shown in Table 6, Sieve has detected seven FSH bugs, including six unknown bugs and one known bug (HDFS-15869). Moreover, ZK-4836 and KAFKA-16412 have been confirmed by developers. Although HDFS-15869 was reported by developers, it has not been fixed yet. Hence, Sieve can detect

Table 6: Bugs detected by Sieve. The last five columns show whether the bugs are detected by different approaches shown in Table 7.

	Bug ID	Failure Symptoms	Random	FATE	Legolas	Chronos	Sieve-I	Sieve-S	Sieve
Unknown Bugs	ZK-4816	A follower cannot follow the leader for a long time (more than 30 seconds)	✗	✗	✗	✗	✗	✓	✓
	ZK-4817	CancelledKeyException cannot catch the client disconnection exception	✗	✗	✗	✓	✗	✓	✓
	ZK-4844	Fail-slow disk while executing writeLongToFile causes the follower to hang	✗	✗	✗	✗	✗	✓	✓
	ZK-4836	Inconsistent ACL index leads to MarshallingError	✗	✗	✗	✗	✗	✓	✓
	KAFKA-16401	One request consumes all request handler threads	✗	✓	✓	✗	✓	✓	✓
	KAFKA-16412	An uncreated topic is considered as a created one	✗	✗	✓	✓	✗	✓	✓
Known Bugs	HDFS-15869	Namenode hangs due to the slow sendResponse	✗	✓	✓	✗	✓	✓	✓

HDFS-15869. Moreover, we did not know about HDFS-15869 before. These bugs cause node hang, uncaught exceptions, node crash, and semantic violation.

ZK-4816. A follower cannot join the cluster for a long time (more than 30 seconds). Specifically, the cluster consists of three nodes. The first node becomes the leader after the election. The leader is stuck in serializing the snapshot to the local fail-slow disk. The followers disconnect from the leader, since their sync operations are blocked by the snapshot serialization in the leader. These two followers start a new leader election. The second node becomes the new leader. However, the old leader does not change the node status. There have been two leaders in the cluster for a long time. The third node is confused and cannot follow any leader for a long time.

ZK-4817. `CancelledKeyException` cannot catch the client disconnection exception in some cases. Specifically, NIOServerCxn throws `CancelledKeyException` if the client disconnects from the server abnormally. When NIOServerCxn executes `doIO`, the disk becomes fail-slow. The client cannot receive heartbeats and disconnects from the server. If `doIO` is blocked for 30 seconds, NIOServerCxn throws `CancelledKeyException`. However, when `doIO` is stuck for more than 30 seconds, `CancelledKeyException` disappears in system logs. Although the clients disconnect from the server abnormally in two scenarios, `CancelledKeyException` does not work in the latter scenario.

ZK-4844. Fail-slow disk while executing writeLongToFile causes the follower to hang. Specifically, the follower executes `writeLongToFile` slowed down by the fail-slow disk. The internal checker is blocked by `writeLongToFile`. The leader excludes the follower from the cluster, but the follower believes that it still acts as a follower. As a result, all requests sent to the follower cannot be processed. This bug is similar to Zookeeper-4074 [71]. Zookeeper-4074 can be solved by adding `-Dlearner.asyncSending=true`. However, this method cannot solve ZK-4844.

ZK-4836. An inconsistent ACL index leads to MarshallingError. Specifically, when a leader creates a node (N0) with an ACL entry (ACL_0) and deletes it due to the node deletion (assume the current aclindex is 0). A follower starts to synchronize the snapshot (including the ACL table and datatree) with the leader. When synchronizing the ACL table to the follower, the NIC becomes fail-slow, which slows down the ACL table transmission. Meanwhile, a client sends a request that creates a node (N1) with a new ACL entry (ACL_1) to the leader. In the leader, the datatree contains N1->1 (indicates that N1 points to the second slot of the ACL table) and the ACL table contains 1->ACL_1 (indicates that ACL_1 is stored in the second slot of the ACL table). The leader synchronizes the updated datatree to the follower. When the follower deserializes the new datatree, the aclindex is set to the number of elements in the ACL table. The follower contains the old ACL table (aclindex=0) and the new datatree (N1->1). The ZAB protocol synchronizes the client creation request from the leader to the follower. When replaying the client creation request in the follower, ACL_1 will be added into the first slot of the ACL table (0->ACL_1). However, the node N1 still points to the second slot of the ACL table (N1->1). Finally, when executing getAcl N1, MarshallingError arises, which leads to the follower crash.

KAFKA-16401. One request consumes all request handler threads. Specifically, a consumer request is stuck in `storeOffsets` slowed down by the fail-slow disk. The thread response for this request is blocked and holds a group lock. Due to timeouts, the consumer resends requests to the broker. All request handler threads are quickly occupied by these requests and stuck in acquiring the group lock. As a result, new requests cannot be processed.

KAFKA-16412. An uncreated topic is considered as a created one. Specifically, a client sends the request for topic creation to the broker. Another client also sends the same request to the broker. When the first request is not finished, the second one fails and returns `TopicExistsException` to the second client. However, subsequent requests that operate on this unestablished topic from the second client fail, which confuses the second client.

HDFS-15869. Namenode hangs due to the slow `sendResponse`. Specifically, a client sends a write request. When the namenode handles this request and writes the editlog to the disk, the disk becomes fail-slow. The thread response for writing the editlog is stuck, so that it does not process the following write requests from clients.

Except for five detected bugs in ZooKeeper and HDFS, Sieve also detects two bugs in Kafka that are out of the five studied systems. Because (1) Sieve is designed to detect FSH bugs for most cloud systems that contain synchronized and timeout mechanisms, not just the studied systems. (2) There

are buggy synchronized and timeout mechanisms in Kafka.

6.1.3 Reproducing Studied Bugs

To further evaluate Sieve's effectiveness, we try to reproduce all studied bugs. For each bug, we first check whether Sieve can correctly identify its corresponding fault point. If successful, we further modify Sieve and only inject the FSH fault at the corresponding point to detect the bug. If the failure report provides workloads, Sieve directly uses them. Otherwise, the workloads shown in Table 5 are used. When the checkers mark test runs suspicious, we manually check whether their symptoms and system logs are consistent with the original failure report in JIRA. If consistent, the corresponding bug is successfully reproduced.

Sieve successfully reproduces 34 of 48 studied bugs. There are 14 bugs not reproduced. For the five bugs (HBase-11536, HDFS-3493, HDFS-11755, HDFS-16659, and MapReduce-1800), multiple fault injections within one test run are necessary. However, Sieve currently focuses on a single fault injection. To reproduce these five bugs, we need to provide multiple buggy fault points and the correct fault injection order. Moreover, for the first four bugs, we need to provide additional checkers due to their silent symptoms (Finding 2). For the remaining six silent bugs (ZK-417, HDFS-7065, HDFS-10301, HDFS-13111, HBase-26195, and HBase-13430), Sieve fails to reproduce them due to the absence of accurate checkers. For the bug (ZK-4293), the complex thread interleaving causes deadlock. However, Sieve is designed to detect concurrency bugs with simple timing constraints (Finding 5). Although Sieve fails to reproduce the above twelve bugs, it can identify buggy fault points. However, Sieve fails to reproduce the remaining bugs (HBase-12270 and HBase-27947), since the static analysis in Sieve cannot analyze the synchronized data structures, i.e., the FIFO queue in these two bugs. To reproduce these two bugs, we need to extend the static analysis in Sieve to analyze the FIFO queue.

6.2 Comparison with Alternative Fault Injection Approaches

6.2.1 Methodology

We compare Sieve with six alternative fault injection approaches shown in Table 7. These approaches include coarse-grained (FATE) and fine-grained (Random, Legolas, Chronos, and two variants of Sieve) tools.

Random is implemented as the baseline of the fault injection strategy. We reuse the client-server architecture of Sieve and instrument the fail-slow agent before each I/O operation. When the fail-slow agent is reached at runtime, the injection controller randomly grants an injection request.

FATE [21] focuses on node-level faults, making it not directly comparable to Sieve. We re-implement its fault injection strategy in Sieve to attempt meaningful comparisons.

Table 7: Settings for Alternative Approaches. S/T I/O operations represent synchronized and timeout-protected I/O operations.

Approach	Fault Point	Injection Strategy
Random	All I/O operations	Random
FATE [21]	All I/O operations	Context-sensitive
Legolas [62]	All I/O operations	Abstract state and bsrr
Chronos [8]	T I/O operations	Deep-priority
Sieve-I	S/T I/O operations	Context-insensitive
Sieve-S	S/T I/O operations	Context-sensitive
Sieve	S/T I/O operations	Grouping and context-sensitive

Specifically, we define the failure ID like FATE, which comprises the class name, function name, line number, and call stack of the I/O operation. The injection controller can receive injection requests from all I/O operations and grant an injection request whose failure ID is unique.

Legolas [62] is a state-of-the-art fine-grained fault injection tool. Legolas infers abstract states of cloud systems and stores fault points in queues. The fault injection strategy of Legolas is called budgeted-state-round-robin (bsrr). Specifically, Legolas selects a queue in the round-robin way and dequeues a fault point to test.

Chronos [8] is a state-of-the-art fine-grained delay injection tool and adopts the deep-priority guided algorithm to detect timeout bugs. However, its core components are not available. Hence, we re-implement its fault injection strategy in Sieve. Specifically, the injection controller records timeout-protected I/O points, randomly selects six fault points to explore and records new fault points that only appear after injecting delays. If the depths of new fault points are larger than the average depth of selected fault points, Sieve injects delays at new fault points and selected fault points in the next test run. The depth of a fault point means the number of preceding fault points that include tested and untested fault points in the execution path. If there is not a new fault point, the injection controller randomly selects six fault points again.

Sieve-I adopts the context-insensitive injection strategy which means that all identified fault points are explored once. Except for the injection strategy, other components of Sieve-I are the same as Sieve.

Sieve-S is built on top of Sieve-I. Sieve-S explores identified fault points multiple times. We evaluate the effectiveness of grouping and context-sensitive strategies by comparing Sieve-S with Sieve and Sieve-I respectively.

All the above six approaches are performed with the same settings as Sieve, i.e., cluster configurations, failure checkers, workload driver, and 2000 test runs that consume more than 20 hours. Additionally, Chronos injects multiple faults since its strategy is designed to explore the combinations of different delays. Except for Chronos, the remaining five approaches inject one fault for each test run. Legolas uses its fault point configuration, which injects a one-minute delay for each test

Table 8: # Dynamic/Static fault points indicate the number of injected and candidate faults. S/T indicates that Sieve identifies all synchronized and timeout-protected fault points. S/T+Gr indicates that Sieve applies the grouping strategy to S/T fault points.

	# Dynamic fault points							# Static fault points		
	Random	FATE	Legolas	Chronos	Sieve-I	Sieve-S	Sieve	All	S/T	S/T+Gr
ZooKeeper	2000	2000	2000	2000	423	1291	705	1905	1266	856
Kafka	2000	2000	2000	2000	267	1968	967	1953	1090	780
HDFS	2000	2000	2000	2000	383	883	658	4216	1974	1568

run. Random and FATE also inject a one-minute delay for each test run like Legolas.

6.2.2 Comparison Result Analysis

Compared with alternative approaches, we aim to answer one question: is Sieve more effective and efficient in detecting FSH bugs? We answer this question from two aspects: the number of detected bugs and explored fault points.

The Number of Detected Bugs. Table 6 shows that Sieve is more effective in detecting FSH bugs. Random does not detect any bugs since it misses buggy fault points in deeper system states due to injecting faults at the beginning of workloads.

Compared with FATE and Legolas, four out of seven FSH bugs can only be detected by Sieve. FATE exhaustively explores the huge fault injection space, which is impractical and inefficient. Hence, FATE misses the buggy points of ZK-4816, ZK-4844, and KAFKA-16401 within 2000 test runs. Although we can add more test runs, FATE will generate more reports which require large human efforts to confirm true bugs. Legolas fails to detect ZK-4816 and ZK-4844 due to the same reason. For ZK-4817, to observe different behaviors in system logs, injecting fault twice with different delay durations at the buggy fault point is necessary for the checkers. Although FATE and Legolas may explore this point multiple times, the duration of the injected delay does not change. Hence, FATE and Legolas fail to detect ZK-4817. ZK-4836 is a concurrency bug and requires a suitable timing constraint. However, the long delay (1min) injected by FATE and Legolas causes the client creation request to timeout, so that the updated datatree will not be synchronized to the follower. As a result, ZK-4836 does not occur. Therefore, FATE and Legolas fail to detect ZK-4836.

Compared with Chronos, Sieve detects five more bugs. ZK-4816, ZK-4844, and HDFS-15869 are out of the scope of Chronos since these bugs are not related to timeout mechanisms. Chronos fails to detect ZK-4836 since multiple delay injections cause the client creation request to timeout, like a single long delay injection in FATE and Legolas. For KAFKA-16401, Sieve injects a delay before the synchronized I/O point on the server side, which causes the client to frequently retry the timeout request. If Chronos wants to detect KAFKA-16401, it needs to repeatedly inject a delay before the same I/O point on the client side. However, this scenario contradicts the deep-priority guided algorithm of Chronos, which aims to explore the combinations of different delays. Chronos can detect ZK-4817 and KAFKA-16412 thanks to our broad log error checker. The original failure checkers of Chronos described in its paper detect two symptoms: server crash and service hang. However, ZK-4817 and KAFKA-16412 do not cause these two symptoms. Therefore, Chronos should have failed to detect the two bugs.

The Number of Explored Fault Points. We record the number of dynamic and static fault points in Table 8. The number of dynamic fault points is less than 2000, which indicates all fault points exposed under current workloads are explored. Tables 6 and 8 show that Sieve can explore fewer fault points, while detecting more bugs. However, Random, FATE, and Legolas explore too many useless fault points to detect some FSH bugs. In some worse cases, Random even repeatedly explores the same fault under the same runtime context. Compared with FATE, Sieve-S can efficiently detect more FSH bugs, since our bug study reveals that synchronized and timeout-protected I/O operations are error-prone, which significantly prunes the fault injection space. Besides, thanks to the effective context-sensitive strategy, Sieve-S explores more useful contexts of fault points and further detects more FSH bugs than Sieve-I. Moreover, compared with Sieve-S, Sieve explores fewer fault points without compromising the number of detected bugs, which indicates that the grouping strategy is effective. In summary, Sieve is more efficient in detecting FSH bugs thanks to its effective design.

Chronos aims to explore the combinations of different delays, which cause the fault injection space to increase in an exponential manner. Therefore, Chronos implements a deep-priority guided algorithm to efficiently explore such huge fault injection space. However, Sieve focuses on the practical and common scenario, i.e., one delay injection (Finding 6). Chronos and Sieve have different target scenarios. Hence, we do not compare with Chronos in this aspect.

6.3 False Positive

Threats of false positives come from two sources. The first source is injected faults. If the injected fault is not realistic, a false positive arises. Specifically, injecting delays before non-blocking I/O operations causes false positives. To eliminate these false positives, Sieve simulates a 10s delay for the

fail-slow disk and NIC using the device mapper (DM) and traffic control (TC) in Linux (a 10s delay is suitable since it exceeds the time consumed by normal I/O operations and does not break cloud systems). As a result, the consumed time of the non-blocking I/O operation is less than 10 seconds. Because the non-blocking mechanisms are implemented above the layer that simulates the fault using DM and TC in Linux, which indicates the non-blocking I/O operation returns before reaching the simulated fault. Based on this feature, Sieve re-executes the suspicious test run marked by failure checkers and calculates the time consumed by the explored I/O operation. If the consumed time is less than 10 seconds, Sieve abandons the report of the suspicious test run. Overall, the faults injected by Sieve do not cause any false positives. In addition, the fault simulated by DM and TC slows down all I/O operations, which cannot enable fine-grained fault injection.

The second source is failure checkers. The gray failure checker is a simplified version of Panorama [28], so it cannot obtain comprehensive information of components in cloud systems. For example, in Zookeeper, if Sieve injects a delay on the client side when the client connects to the server, the gray failure checker will obtain an error report from the client. Meanwhile, the gray failure checker obtains a healthy status of the server by using `./zkServer.sh status` provided by Zookeeper. Based on different server statuses observed by the client and server, the gray failure checker reports a bug. In fact, the reported bug is a false positive. The log error checker also makes mistakes. For example, our checkers mark a test run suspicious when a fault is injected in `LearnerHandler` since ERROR entries appear in system logs. However, such ERROR entries are expected behaviors in ZooKeeper. To eliminate these false positives, for each suspicious test run, we (1) read system logs to understand system behaviors; (2) check whether system behaviors are consistent with design documentation. (3) report this bug to developers and discuss it with them. In the near future, we plan to enhance our checkers by using more accurate rules. However, our core contribution focuses on our bug study and fault injection methodology.

6.4 Overhead Analysis

Table 9 shows the runtime overhead of Sieve. The **Baseline** column shows the original workload run time without any instrumentation. The **Info Collection** column shows the average fault-free run time of the instrumented system. The **Average Test Time** column shows the average run time with one fault injection.

The results show that Sieve introduces 1.1× to 4.9× baseline time for runtime information collection. On average, testing a fault point consumes around 1.9× to 6.3× baseline time. Each test run includes a series of operations such as initializing the execution environment, executing the workload, collecting runtime information, deciding the fault injection, and checking failure symptoms. Additionally, the static analysis completes in three minutes and consumes at most 8GB memory for each cloud system.

Table 9: Runtime Overhead (in seconds).

System	Baseline	Info Collection	Average Test Time
ZooKeeper	9.41	13.17	37.64
Kafka	7.12	34.89	44.86
HDFS	24.87	27.36	52.23

7 Discussions

Fault Model. Sieve currently injects one delay to simulate the fail-slow hardware. However, the fail-slow hardware may incur other subtle faults such as exceptions and multiple delays. Hence, Sieve misses some bugs caused by these subtle faults. For example, in § 6.1.3, Sieve fails to detect five bugs caused by other subtle faults. Extending Sieve to support more fault models becomes our future work.

Fault Point Analysis. The keywords, e.g., `synchronized`, used in synchronized and timeout mechanisms are necessary for the fault point analysis in § 4.3. Sieve has analyzed several most common keywords: `synchronized`, `lock`, `wait`, `await`, `tryacquire`, `join`, `nanoTime` and `currentTimeMillis`. Our implementation is modular, which makes it convenient for developers to add more keywords and patterns completing the fault point analysis in Sieve.

Workload Driver. Table 8 indicates current workload drivers cannot trigger all candidate fault points since the used workloads are simple, which limit the code coverage. Fuzzing is an automatic input generation technique [3, 7, 13, 20, 49, 60, 63, 64, 69] that can improve the code coverage. Developers can integrate fuzzing tools into Sieve to explore more candidate fault points.

8 Related Work

Fault Injection. In recent years, various fault injection techniques [1, 6, 15–17, 21, 31, 32, 42, 48] have been proposed to detect bugs triggered by various faults in cloud systems. Many existing schemes focus on coarse-grained faults and adopt various fault injection strategies. For example, CrashTuner [42] injects node crashes when meta-info variables are accessed; NEAT [1] provides simple APIs for developers to inject network partitions. Unlike them, Sieve instruments fail-slow agents before I/O operations to enable fine-grained FSH fault injection. Recent fine-grained delay injection tools [8,62] fail to efficiently explore the FSH fault injection space due to overlooking the characteristics of FSH failures. Our work provides a bug study to analyze FSH failures and concludes some findings shown in § 3 Inspired by our bug study, Sieve is proposed to efficiently explore the FSH fault injection space.

CORDS [14] and WASABI [55] inject expections into cloud systems to detect error handling bugs. However, Sieve injects delays to detect FSH bugs. Sieve is orthogonal to CORDS and WASABI.

Distributed System Model Checker. Distributed system model checkers [34, 45, 66] intercept non-deterministic distributed events and permute their ordering. These schemes detect protocol bugs caused by complex interleaving of node-level events, such as message and node crash/reboot, while Sieve aims to detect implementation-level bugs triggered by fine-grained FSH faults. Sieve is complementary to existing schemes.

Distributed Concurrency Bug Detection. Several schemes [36, 38, 41, 68] have been proposed to detect concurrency bugs in cloud systems. For example, DCatch [36] uses specific happen-before rules to identify conflicting operations and reorders them. Sieve is an FSH fault injection framework that aims to detect diverse bugs including concurrency bugs.

Bug Studies on Fail-Slow Incidents. Guanwai et al. [22] present a study of fail-slow hardwares and analyze how fail-slow hardwares occur. Several works [39, 43, 51] provide studies of fail-slow failures at the software level and analyze how to accurately detect fail-slow failures by in-production monitoring. However, unlike them, our bug study aims to study how fail-slow hardwares affect the software so that we can design an effective and efficient fault injection tool to detect FSH bugs before releasing production. Besides, our bug study shows that FSH failures caused by fail-slow hardwares include fail-slow and fail-stop failures. Our bug study is complementary to existing bug studies.

Fail-Slow Failure Detection. Several works [28, 39, 43, 51] develop advanced detectors for fail-slow failures. Panorama [28] detects fail-slow failures by enhancing the observability of different system components. IASO [51] detects fail-slow nodes based on timeout signals and peer evaluation. OmegaGen [39] generates customized watchdogs for detecting and localizing fail-slow failures. PERSEUS [43] leverages machine learning techniques to detect fail-slow failures. These detectors focus on in-production monitoring and require a significantly long time to detect failures, which are not suitable for testing before releasing production. For example, ISAO has been deployed for more than 1.5 years to catch fail-slow failures. However, Sieve focuses on detecting FSH bugs before releasing production by injecting FSH faults.

9 Conclusion

Fail-slow hardwares cause severe failures in cloud systems. This work presents a study of 48 real-world FSH failures in cloud systems to analyze their characteristics. We propose Sieve, a novel fault injection testing framework that enables fine-grained FSH fault injection for detecting FSH bugs. Sieve treats synchronized and timeout-protected I/O operations as candidate fault points that are likely to trigger FSH bugs. Sieve implements grouping and context-sensitive injection strategies to efficiently explore candidate fault points. Our evaluation shows that Sieve is effective and efficient in detecting FSH bugs in real-world cloud system.

Acknowledgments

This work was supported in part by National Natural Science Foundation of China (NSFC) under Grant No. 62125202 and U22B2022. We are grateful to our shepherd, Xiang Ren, and anonymous reviewers for their comments and suggestions.

References

[1] Ahmed Alquraan, Hatem Takruri, Mohammed Alfatafta, and Samer Al-Kiswany. An analysis of network-partitioning failures in cloud systems. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 51–68, 2018.

[2] HDFS architecture. Online. Available: https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html.

[3] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1032–1043, 2016.

[4] Apache Cassandra. Online. Available: https://cassandra.apache.org.

[5] K Mani Chandy and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems (TOCS)*, 3(1):63–75, 1985.

[6] Haicheng Chen, Wensheng Dou, Dong Wang, and Feng Qin. Cofi: consistency-guided fault injection for cloud systems. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, pages 536–547, 2020.

[7] Yang Chen, Alex Groce, Chaoqiang Zhang, Weng-Keen Wong, Xiaoli Fern, Eric Eide, and John Regehr. Taming compiler fuzzers. In *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*, pages 197–208, 2013.

[8] Yuanliang Chen, Fuchen Ma, Yuanhang Zhou, Ming Gu, Qing Liao, and Yu Jiang. Chronos: Finding timeout bugs in practical distributed systems by deep-priority fuzzing with transient delay. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 1939–1955. IEEE, 2024.

[9] Google compute engine incident 17008. Online. Available: hhttps://status.cloud.google.com/incident/compute/17008.

[10] Ting Dai, Jingzhu He, Xiaohui Gu, and Shan Lu. Understanding real-world timeout problems in cloud server systems. In *2018 IEEE International Conference on Cloud Engineering (IC2E)*, pages 1–11. IEEE, 2018.

[11] D.Nadolny. Debugging distributed systems. In *SREcon*, 2016.

[12] Thanh Do, Mingzhe Hao, Tanakorn Leesatapornwongsa, Tiratat Patana-anake, and Haryadi S Gunawi. Limplock: Understanding the impact of limpware on scale-out cloud systems. In *Proceedings of the 4th annual Symposium on Cloud Computing*, pages 1–14, 2013.

[13] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. Collafl: Path sensitive fuzzing. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 679–696, 2018.

[14] Aishwarya Ganesan, Ramnatthan Alagappan, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Redundancy does not imply fault tolerance: Analysis of distributed storage reactions to file-system faults. *ACM Transactions on Storage (TOS)*, 13(3):1–33, 2017.

[15] Yu Gao, Wensheng Dou, Feng Qin, Chushu Gao, Dong Wang, Jun Wei, Ruirui Huang, Li Zhou, and Yongming Wu. An empirical study on crash recovery bugs in large-scale distributed systems. In *Proceedings of the 2018 26th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*, pages 539–550, 2018.

[16] Yu Gao, Wensheng Dou, Dong Wang, Wenhan Feng, Jun Wei, Hua Zhong, and Tao Huang. Coverage guided fault injection for cloud systems. In *Proceedings of IEEE/ACM SIGSOFT International Conference on Software Engineering (ICSE)*, 2023.

[17] Yu Gao, Dong Wang, Qianwang Dai, Wensheng Dou, and Jun Wei. Common data guided crash injection for cloud systems. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*, pages 36–40, 2022.

[18] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 29–43, 2003.

[19] Gocardless:API and Dashboard outage on 10 October 2017. Online. Available: https://aws.amazon.com/cn/message/12721/.

[20] Patrice Godefroid, Adam Kiezun, and Michael Y Levin. Grammar-based whitebox fuzzing. In *Proceedings of the 29th ACM SIGPLAN conference on programming language design and implementation*, pages 206–215, 2008.

[21] Haryadi S Gunawi, Thanh Do, Pallavi Joshi, Peter Alvaro, Joseph M Hellerstein, Andrea C Arpaci-Dusseau, Remzi H Arpaci-Dusseau, Koushik Sen, and Dhruba Borthakur. Fate and destini: A framework for cloud recovery testing. In *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11)*, pages 238–252, 2011.

[22] Haryadi S Gunawi, Riza O Suminto, Russell Sears, Casey Golliher, Swaminathan Sundararaman, Xing Lin, Tim Emami, Weiguang Sheng, Nematollah Bidokhti, Caitie McCaffrey, et al. Fail-slow at scale: Evidence of hardware performance faults in large production systems. *ACM Transactions on Storage (TOS)*, 14(3):1–26, 2018.

[23] Apache HBase. Online. Available: https://hbase.apache.org.

[24] HBase-26195. Online. Available: https://issues.apache.org/jira/browse/HBASE-26195.

[25] HDFS-5341. Online. Available: https://issues.apache.org/jira/browse/HDFS-5341.

[26] HDFS-5522. Online. Available: https://issues.apache.org/jira/browse/HDFS-5522.

[27] HDFS-9178. Online. Available: https://issues.apache.org/jira/browse/HDFS-9178.

[28] Peng Huang, Chuanxiong Guo, Jacob R Lorch, Lidong Zhou, and Yingnong Dang. Capturing and enhancing in situ system observability for failure detection. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 1–16, 2018.

[29] INS. Online. Available: https://www.instagram.com/.

[30] Javassist. Online. Available: https://www.javassist.org/, 1999.

[31] Jepsen. Online. Available: https://github.com/jepsen-io/jepsen, 2016.

[32] Pallavi Joshi, Haryadi S Gunawi, and Koushik Sen. Prefail: A programmable tool for multiple-failure injection. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, pages 171–188, 2011.

[33] Apache Kafka. Online. Available: https://kafka.apache.org.

[34] Tanakorn Leesatapornwongsa, Mingzhe Hao, Pallavi Joshi, Jeffrey F Lukman, and Haryadi S Gunawi. Samc: Semantic-aware model checking for fast discovery of deep bugs in cloud systems. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 399–414, 2014.

[35] Jiaxin Li, Yuxi Chen, Haopeng Liu, Shan Lu, Yiming Zhang, Haryadi S Gunawi, Xiaohui Gu, Xicheng Lu, and Dongsheng Li. Pcatch: Automatically detecting performance cascading bugs in cloud systems. In *Proceedings of the Thirteenth EuroSys Conference*, pages 1–14, 2018.

[36] Haopeng Liu, Guangpu Li, Jeffrey F Lukman, Jiaxin Li, Shan Lu, Haryadi S Gunawi, and Chen Tian. Dcatch: Automatically detecting distributed concurrency bugs in cloud systems. *ACM SIGARCH Computer Architecture News*, 45(1):677–691, 2017.

[37] Haopeng Liu, Shan Lu, Madan Musuvathi, and Suman Nath. What bugs cause production cloud incidents? In *Proceedings of the Workshop on Hot Topics in Operating Systems*, pages 155–162, 2019.

[38] Haopeng Liu, Xu Wang, Guangpu Li, Shan Lu, Feng Ye, and Chen Tian. Fcatch: Automatically detecting time-of-fault bugs in cloud systems. *ACM SIGPLAN Notices*, 53(2):419–431, 2018.

[39] Chang Lou, Peng Huang, and Scott Smith. Understanding, detecting and localizing partial failures in large system software. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 559–574, 2020.

[40] Haonan Lu, Kaushik Veeraraghavan, Philippe Ajoux, Jim Hunt, Yee Jiun Song, Wendy Tobagus, Sanjeev Kumar, and Wyatt Lloyd. Existential consistency: Measuring and understanding consistency at facebook. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 295–310, 2015.

[41] Jie Lu, Feng Li, Lian Li, and Xiaobing Feng. Cloudraid: hunting concurrency bugs in the cloud via log-mining. In *Proceedings of the 2018 26th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering*, pages 3–14, 2018.

[42] Jie Lu, Chen Liu, Lian Li, Xiaobing Feng, Feng Tan, Jun Yang, and Liang You. Crashtuner: detecting crash-recovery bugs in cloud systems via meta-info analysis. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 114–130, 2019.

[43] Ruiming Lu, Erci Xu, Yiming Zhang, Fengyi Zhu, Zhaosheng Zhu, Mengtian Wang, Zongpeng Zhu, Guangtao Xue, Jiwu Shu, Minglu Li, et al. Perseus: A fail-slow detection framework for cloud storage systems. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*, pages 49–64, 2023.

[44] Ruiming Lu, Erci Xu, Yiming Zhang, Zhaosheng Zhu, Mengtian Wang, Zongpeng Zhu, Guangtao Xue, Minglu Li, and Jiesheng Wu. Nvme ssd failures in the field: the fail-stop and the fail-slow. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 1005–1020, 2022.

[45] Jeffrey F Lukman, Huan Ke, Cesar A Stuardo, Riza O Suminto, Daniar H Kurniawan, Dikaimin Simon, Satria Priambada, Chen Tian, Feng Ye, Tanakorn Leesatapornwongsa, et al. Flymc: Highly scalable testing of complex interleavings in distributed systems. In *Proceedings of the Fourteenth EuroSys Conference 2019*, pages 1–16, 2019.

[46] Ao Ma, Rachel Traylor, Fred Douglis, Mark Chamness, Guanlin Lu, Darren Sawyer, Surendar Chandra, and Windsor Hsu. Raidshield: characterizing, monitoring, and proactively protecting against disk failures. *ACM Transactions on Storage (TOS)*, 11(4):1–28, 2015.

[47] MapReduce-1800. Online. Available: https://issues.apache.org/jira/browse/MAPREDUCE-1800.

[48] Chaos Monkey. Online. Available: https://netflix.github.io/chaosmonkey, 2012.

[49] Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. Semantic fuzzing with zest. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 329–340, 2019.

[50] PagerDuty. Online. Available: https://www.pagerduty.com/.

[51] Biswaranjan Panda, Deepthi Srinivasan, Huan Ke, Karan Gupta, Vinayak Khot, and Haryadi S Gunawi. Iaso: A fail-slow detection and mitigation framework for distributed storage services. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 47–62, 2019.

[52] Daniel J Scales, Mike Nelson, and Ganesh Venkitachalam. The design of a practical system for fault-tolerant virtual machines. *ACM SIGOPS Operating Systems Review*, 44(4):30–39, 2010.

[53] AWS service disruption. Online. Available: https://aws.amazon.com/cn/message/12721/.

[54] Jira Software. Online. Available: https://www.atlassian.com/software/jira.

[55] Bogdan Alexandru Stoica, Utsav Sethi, Yiming Su, Cyrus Zhou, Shan Lu, Jonathan Mace, Madanlal Musuvathi, and Suman Nath. If at first you don't succeed, try, try, again...? insights and llm-informed tooling for detecting retry bugs in software systems. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, pages 63–78, 2024.

[56] Jeff Terrace and Michael J Freedman. Object storage on craq: High-throughput chain replication for read-mostly workloads. In *USENIX Annual Technical Conference*, 2009.

[57] TikTok. Online. Available: https://www.tiktok.com/.

[58] MapReduce tutorial. Online. Available: https://hadoop.apache.org/docs/current/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html.

[59] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot: A java bytecode optimization framework. In *CASCON First Decade High Impact Papers*, pages 214–224. 2010.

[60] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. Skyfire: Data-driven seed generation for fuzzing. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 579–594, 2017.

[61] WeChat. Online. Available: https://weixin.qq.com/.

[62] Haoze Wu, Jia Pan, and Peng Huang. Efficient exposure of partial failure bugs in distributed systems with inferred abstract states. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 1267–1283, 2024.

[63] Mingyuan Wu, Kunqiu Chen, Qi Luo, Jiahong Xiang, Ji Qi, Junjie Chen, Heming Cui, and Yuqun Zhang. Enhancing coverage-guided fuzzing via phantom program. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1037–1049, 2023.

[64] Mingyuan Wu, Yicheng Ouyang, Minghai Lu, Junjie Chen, Yingquan Zhao, Heming Cui, Guowei Yang, and Yuqun Zhang. Sjfuzz: Seed and mutator scheduling for jvm fuzzing. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1062–1074, 2023.

[65] X. Online. Available: https://x.com/.

[66] Junfeng Yang, Tisheng Chen, Ming Wu, Zhilei Xu, Xuezheng Liu, Haoxiang Lin, Mao Yang, Fan Long, Lintao Zhang, and Lidong Zhou. Modist: Transparent model checking of unmodified distributed systems. In *NSDI'09*, pages 213–228, 2009.

[67] Ding Yuan, Yu Luo, Xin Zhuang, Guilherme Renna Rodrigues, Xu Zhao, Yongle Zhang, Pranay U Jain, and Michael Stumm. Simple testing can prevent most critical failures: An analysis of production failures in distributed {Data-Intensive} systems. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 249–265, 2014.

[68] Xinhao Yuan and Junfeng Yang. Effective concurrency testing for distributed systems. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1141–1156, 2020.

[69] M. Zalewski. American fuzzy lop. Online. Available: https://lcamtuf.coredump.cx/afl/, 1999.

[70] Apache ZooKeeper. Online. Available: https://zookeeper.apache.org.

[71] ZooKeeper-4074. Online. Available: https://issues.apache.org/jira/browse/ZOOKEEPER-4074.

[72] ZooKeeper-710. Online. Available: https://issues.apache.org/jira/browse/ZOOKEEPER-710.

CONVEROS: Practical Model Checking for Verifying Rust OS Kernel Concurrency

Ruize Tang[1]*, Minghua Wang[2], Xudong Sun[3], Lin Huang[2], Yu Huang[1]†, Xiaoxing Ma[1]

[1]*State Key Laboratory for Novel Software Technology, Nanjing University*
[2]*Ant Group* [3]*University of Illinois Urbana-Champaign*

Abstract

ASTERINAS is an open-source, general-purpose operating system written in Rust, compatible with the Linux ABI, and designed with a focus on reliability and security.

We developed a practical model-checking methodology, CONVEROS, to verify the correctness of ASTERINAS concurrency modules such as synchronization primitives and critical thread-safety components. CONVEROS leverages the rigor of formal specifications and introduces a multi-layered, multi-grained specification approach to make writing scalable specifications practical, demonstrated in our case by writing PlusCal specifications for Rust code. It also makes conformance checking incremental and more automated to detect specification-code discrepancies. While many formal methods are challenging to apply due to complexity and the expertise required, CONVEROS makes model checking cost-effective, accessible, and adaptable to evolving specifications and code. We applied CONVEROS to 12 critical concurrency modules, uncovering 20 bugs that led to issues such as data races, deadlocks, livelocks, and kernel panics. With a specification-to-code ratio ranging from 0.3 to 2.3 and a verification effort of only four person-months, our results demonstrate the practicality and effectiveness of CONVEROS.

1 Introduction

ASTERINAS [2, 64] is a new general-purpose OS kernel written in Rust, compatible with the Linux ABI and designed for improved reliability and security. It is being developed with the goal of supporting future deployment within Ant Group. Although still in its early stages, ASTERINAS has a substantial codebase of 100K lines of Rust, supports 200 commonly used Linux syscalls, and can run large-scale applications such as the JVM and MySQL. ASTERINAS is open-source and has built an active community, attracting many contributors.

While ASTERINAS is rapidly evolving, we have encountered numerous low-level concurrency bugs that tests cannot reliably rule out. Although Rust greatly enhances memory safety and prevents data races in safe code, OS kernel development inevitably requires `unsafe`, which may still introduce concurrency-related undefined behaviors such as data races. Logical errors and flawed designs can lead to deadlocks, livelocks, and functional violations. These concurrency bugs are notoriously difficult to detect, reproduce, and fix correctly.

We decide to apply formal methods by model checking the ASTERINAS implementation because we want to verify the correctness of complex concurrency code to prevent severe bugs from reaching production, inspired by recent successful industrial experiences [31, 38, 47, 61] and efforts to enhance the practicality of model checking [31, 34, 38, 63].

However, applying model checking to complex OS implementations is hard. First, writing specifications can be challenging, as it requires domain knowledge with formal methods and poor specifications can hinder model checking scalability. Second, discrepancies between specifications and implementations–caused by modeling errors or the rapid evolution of code–can compromise verification results, leading to both false positives and false negatives.

Contributions. To address these challenges, we present CONVEROS, a practical methodology for model checking OS concurrency modules. CONVEROS consists of three steps: (1) writing formal specifications (§4), (2) checking conformance between implementation traces and specifications (§5), and (3) model checking the refined specifications. We focus on steps (1) and (2), which make model checking results effectively reflect code correctness. To make CONVEROS practical, each step is small, incremental and provides rapid feedback.

For writing specifications, we introduce a multi-layered, multi-grained specification approach (§4), which divides each module into manageable components for incremental modeling at different granularities. The multi-layered specifications consist of high-level and low-level specifications (§4.1). The high-level specifications represent the correct design, abstracting away implementation details. The low-level specifications reflect code details and incorporate potential code bugs. This ensures model checking can identify those bugs effectively.

*Work done during an internship at Ant Group.
†Corresponding author.

We use a multi-grained specification approach [63] for low-level specifications (§4.2), enabling incremental verification by verifying modules separately and coarsening actions if the modules do not interact. To make writing specifications more accessible, CONVEROS adopts the imperative-like language PlusCal[54], which is compiled into TLA$^+$[53]. PlusCal offers a syntax similar to C, making it familiar to programmers. Writing PlusCal specifications is applicable across programming languages; in our case, we model Rust code.

For conformance checking, we employ trace validation [34] to check specification-code conformance (§5). Trace validation executes the implementation to generate traces and checks whether these traces are allowed in the specification's state space (§5.1 and §5.2). We enhanced trace validation to improve automation and usability by introducing a new algorithm that automatically infers missing events (§5.3). This improvement addresses issues such as inaccurate timestamps where recorded event times do not precisely reflect actual occurrences and enables quick and incremental establishment of specification-code conformance, even when logging is incomplete for all actions, significantly reducing logging effort. Checking high-level specification conformance helps identify code bugs. Checking low-level specification conformance increases confidence that the specifications accurately reflect the code's behavior. When model checking low-level specifications reveals violations, we write test cases to confirm the bugs and often uncover by-product bugs during testing.

Key Results. We have applied CONVEROS to ASTERINAS by verifying 12 critical concurrency modules including spinlocks, mutexes, file locks, futex, SysV semaphore IPC, a lock-free ring buffer, a locking protocol for page tables, and lock usage in TTY. We have detected 20 new bugs in ASTERINAS: 9 were automatically detected by CONVEROS, and 11 were manually found when diagnosing bugs reported by CONVEROS. All bugs have been confirmed, and 14 have been fixed. The effort required for writing specifications and checking conformance has proven to be manageable and cost-effective. The specification-to-code ratio ranges from 0.3 to 2.3, with approximately 43.5 person-days spent writing all specifications and 21.5 person-days on conformance checks–amounting to a total of about four person-months.

Summary. This paper presents four main contributions:

- We designed a practical model checking methodology, CONVEROS, for verifying safety and liveness of OS concurrency modules, featuring a structured, incremental process for specification, trace validation, and model checking.
- We developed a multi-layered, multi-grained specification approach that improves practicality and enables incremental modeling and scalable verification by composing modules at different granularities.
- We proposed a novel algorithm to enhance trace validation for spec-code conformance by automatically searching for missing events, and improved automation and usability for granularity alignment and discrepancy debugging.
- We applied CONVEROS to ASTERINAS, modeling and verifying 12 concurrency modules. CONVEROS revealed 20 bugs and helped improve code correctness, demonstrating its practicality for real-world OS verification.

2 Background and Motivation

2.1 ASTERINAS

ASTERINAS [2, 64] is an open-source collaborative project between academic researchers and industry practitioners aimed at creating a production-grade operating system entirely in Rust. It aims to support general-purpose usage scenarios such as data centers and trusted execution environments, where thread safety is essential. Although still in the early stages, ASTERINAS is positioned as a serious industry effort intended to serve as a drop-in replacement for Linux, providing Linux-compatible system call interfaces and replicating key Linux features. To date, it currently comprises approximately 100K lines of code, has implemented 200 of the 336 most commonly used Linux system calls and is capable of running large-scale applications such as JVM and MySQL.

Rust's ownership model and strict borrowing rules prevent memory safety issues like dangling pointers and use-after-free errors. An example of Rust ownership and concurrency is shown in Figure 1, illustrating the use of Mutex for concurrently updating shared data. Line 6 is detailed in Figure 9, while line 7 involves dereferencing mutable data, which internally uses `unsafe` as shown in Figure 14. Rust's ownership model ensures proper resource management, such as automatically unlocking the mutex when the guard goes out of scope. However, these features do not eliminate all concurrency issues. They can still occur in unsafe code or result from design flaws and improper use of low-level synchronization primitives [66]. Concurrency issues are still frequently reported [3], particularly in custom synchronization primitives designed specifically for the OS. This motivates the adoption of formal methods to rigorously verify the concurrency correctness of these modules and increase confidence in their reliability.

```
1 // Arc shares ownership across threads via atomic ref counting
2 let m = Arc::new(Mutex::new(0));
3 for _ in 0..10 {
4   let c = Arc::clone(&m); // refcount++; `c` owns a reference
5   thread::spawn(move || { // Ownership of `c` moves in
6     let mut guard = c.lock(); // Blocks until lock acquired
7     *guard += 1; // Critical section: mutable access
8     // `guard` dropped: lock released; `c` dropped: refcount--
9   });} // `m` dropped: refcount goes to zero, Mutex is dropped
```

Figure 1: Rust ownership and concurrency example

2.2 TLA$^+$ and PlusCal

TLA$^+$. TLA$^+$ [53] is a formal specification language widely used in both academia and industry to verify concurrent and distributed systems [32, 34, 38, 47, 61, 63].

```
CONSTANTS PS \* Process set
(*--algorithm AbstractLock {
variables locked = FALSE;
procedure acquire_lock() {
lock: \* A label
  await locked = FALSE;
  locked := TRUE;
  return;
}
procedure release_lock() {
unlock:
  locked := FALSE;
  return;
}
fair process (p \in PS) {
start:
  while (TRUE) {
    call acquire_lock();
  cs:
    skip;
    call release_lock();
  }
}}*)
```

```
VARIABLES locked, pc
\* Some variables are omitted
vars == <<locked, pc>>
lock(p) ==
  /\ pc[p] = "lock"
  /\ locked = FALSE
  /\ locked' = TRUE
  /\ pc' = [pc EXCEPT ![p]="cs"]
acquire_lock(p) == lock(p)
start(p) ==
  /\ pc[p] = "start"
  /\ pc' = [pc EXCEPT ![p]="lock"]
  /\ UNCHANGED locked
\* Some actions are omitted
Init ==
  /\ locked = FALSE
  /\ stack = [p \in PS |-> <<>>]
  /\ pc = [p \in PS |-> "start"]
Next == \E p \in PS:
  \/ acquire_lock(p)
  \/ release_lock(p)
  \/ start(p)
  \/ cs(p)
```

Figure 2: A simple lock specification. The PlusCal specification (left) includes two procedures (e.g., `acquire_lock`) and four atomic actions/labels (e.g., `lock`) with `PS` processes executing the `process` statement concurrently. The `lock` action can be executed only when the `await` condition is true. In the compiled TLA$^+$ specification (right), `pc` is a generated variable that stores the next labels to execute for each process. "`\E`" and "`\A`" represent existential and universal quantifiers.

```
1  struct RangeLockItem {lock: RangeLock, waitqueue: Arc<WaitQueue>}
2  struct RangeLockList {inner: RwMutex<Vec<RangeLockItem>>}
3  impl RangeLockList {
4  fn set_lock(&self, req_lock: &RangeLockItem) -> Result<()> {
5    loop { // Loop until `req_lock` is inserted
6      let conflict_lock;
7      { let mut list=self.inner.write();// Access `list` for write
8        if let Some(existing_lock) = list
9          .iter().find(|l| req_lock.conflict_with(l))
10       { // Find a conflict lock, clone for waiting
11         conflict_lock = existing_lock.clone();
12       } else { // No conflict lock, insert and return
13         Self::insert_lock_into_list(&mut list, req_lock);
14         Ok(()) } // Implicit return
15     } // `list` (write guard) is dropped at the end of the scope
16     conflict_lock.waitqueue.wait_until(|| {// Wait: no conflicts
17       let list = self.inner.read(); // Access `list` for read
18       if list.iter().any(|l| req_lock.conflict_with(l)) {
19         None // Continue waiting
20       } else { Some(()) } // Condition satisfied, break waiting
21     }); // `conflict_lock` is dropped at the end of loop
22   }}}
```

Figure 3: Buggy range lock code causing lost wakeup. When a waiter (line 16) is woken up, it checks for conflicts, and if found, it continues waiting (highlighted code) on `conflict_lock.waitqueue`, which is shared with `existing_lock`'s (line 11). When `existing_lock` is removed from `list` by another thread, that thread wakes waiters on `existing_lock.waitqueue`–the only opportunity for them to be woken up. If a waiter rechecks and waits again, no thread can wake it, resulting in lost wakeup.

TLA$^+$ models a system as a state machine comprising variables, initial states, and state transitions. A typical specification follows the form `Init /\ [Next]_vars /\ L`. "/\" and "\/" represent conjunction and disjunction. `Init` specifies the initial state by assigning values to all variables. `Next` specifies state transitions as a disjunction of actions, modeling nondeterminism by allowing multiple possible next steps. `vars` contains all variables in the specification. `[Next]_vars` expands to `Next \/ vars' = vars`, representing that either `Next` holds or all variables remain unchanged (i.e., stuttering), where primed variables represent their next state. `L` is a temporal formula used to assert liveness and fairness assumptions.

To ensure correctness, safety properties (asserting that nothing bad ever happens) and liveness properties (asserting that something good eventually happens) must also be specified.

PlusCal. PlusCal is a formal specification language designed to describe algorithms using imperative-like structures, with simple syntax and constructs similar to widely-used programming languages (e.g., `while` loops and `procedure` calls), making it more accessible to programmers. PlusCal specifications are compiled into TLA$^+$ and verified using TLC [1].

A PlusCal specification consists of one or more processes, declared with the `process` keyword. Each process executes sequentially, while multiple processes run concurrently and can interleave. In the `process` and `procedure` bodies, labels are defined to mark atomic actions. Figure 2 illustrates a simple lock specification in PlusCal and its compiled TLA$^+$.

Industrial experiences show that programmers find PlusCal more productive to start with than TLA$^+$ [61]. Since low-level specifications need to closely reflect the code's structure. PlusCal offers an imperative style that is more intuitive for programmers, making writing, maintaining, and incrementally improving specifications significantly easier.

2.3 Motivation Example

Concurrency bugs, especially in OS kernels, are difficult to detect, reproduce and fix. Addressing concurrency bugs through expert reviews and CI testing is unreliable and may overlook corner cases. Formal methods provide rigorous verification of safety and liveness properties, helping identify bugs, generate violation traces, and ensure the correctness of fixed code.

We demonstrate the effectiveness of CONVEROS in identifying a hard-to-find, difficult-to-fix concurrency bug in range lock implementation [8] (Figure 3). This bug was introduced by a bug fix, but the fix was still incorrect [28]. We fixed it correctly after multiple failed fix attempts.

Range locks (i.e., file record locks) allow processes to place multiple read and write locks on different regions of the same file via the `fcntl` system call. This involves complex semantics for merging and splitting locks when new locks overlap with existing regions. The implementation maintains a list (i.e., `RangeLockList`) of active locks for the file inode associated with the file descriptor. When a requested lock conflicts with an existing one (i.e., their ranges overlap and at least one is a write lock), the process waits for the conflict to be resolved (lines 16–21) before inserting the new lock.

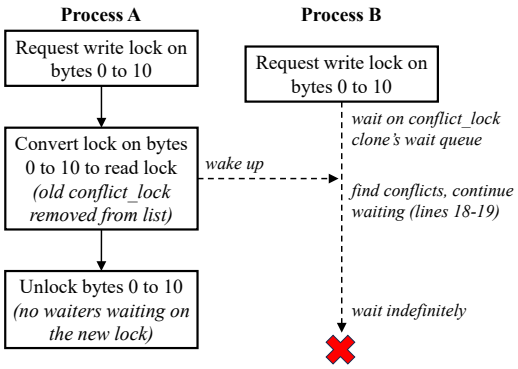


Figure 4: Range lock bug trace. Process A requests a write lock, blocking Process B's conflicting write lock request. Process A then converts its lock type to read atomically, replacing the old write lock with a new read lock in `list` and waking up the waiters on the old write lock. Process B is woken up but finds conflicts with the new read lock and re-enters waiting. Process A unlocks the read lock but wakes no process because no waiters are on the read lock's `waitqueue`.

The model checker revealed a deadlock bug cause by lost wakeup of the waiting process. The root cause lies in how the code manages wait queues for conflicting locks. When a requested lock `req_lock` conflicts with an existing one (lines 8–9), the thread clones `existing_lock` into `conflict_lock` (line 11), also cloning its `Arc`-wrapped `waitqueue` (line 1). This creates a shared `waitqueue` between threads. The thread then blocks on this `waitqueue` (line 16), expecting another thread operating on the same queue to wake it up when the conflict may have been resolved. However, if all locks associated with that `waitqueue` are removed from the list while some threads are still waiting on it, those threads will block indefinitely. Figure 4 shows how the bug can occur. Several attempts to fix the bug in the specification were identified as incorrect through model checking, highlighting the challenge of correctly fixing concurrency bugs.

- **First fix attempt.** The fix transferred the `wait_queue` to the new lock when replacing the old lock (scenario in Figure 4). However, the model checker revealed another lost wakeup scenario: when a `conflict_lock` is removed by `unlock`, it wakes up the waiting process. But before the waiter checks for conflicts, another process inserts a conflicting lock, causing the waiter to re-enter the wait state even when the old `conflict_lock` has been removed from `list`.
- **Second fix attempt.** The previous attempt demonstrated the need to ensure the `conflict_lock` remains in the list before entering the wait state. To avoid a costly loop check, we tried optimizing by checking if the reference count of the `conflict_lock.waitqueue` would reach 0. However, the model checker revealed that multiple processes could be waiting on the same `conflict_lock`, causing multiple clones and a non-zero reference count even when the lock was no longer in the list.

```
1  fn set_lock(&self, req_lock: &RangeLockItem) -> Result<()> {
2    loop {
3      let (waiter, waker); // Dedicated waiter/waker pair
4      { let mut list=self.inner.write();// Access `list` for write
5        if let Some(existing_lock) = list
6          .iter().find(|l| req_lock.conflict_with(l))
7        { (waiter, waker) = Waiter::new_pair();
8          existing_lock.waitqueue.enqueue(waker);
9        } else {
10         Self::insert_lock_into_list(&mut list, req_lock);
11         Ok(()) }
12     } // `list` (write guard) is dropped at the end of the scope
13     waiter.wait(); // Upon wakeup, enter the next iteration
14   }}
```

Figure 5: Correct bug fix code for range lock. Uses a dedicated waiter/waker pair instead of cloning `existing_lock` (lines 7–8), and upon wakeup, enters the next loop iteration to recheck for conflicts (line 13).

- **Third fix attempt.** We then attempted to wait on an empty condition; once woken up, the process would start the next loop for conflict detection. The model checker identified a livelock scenario: at the end of each loop, the `conflict_lock` clone would be dropped, waking up all waiting processes. These processes would then drop their `conflict_lock` clones, re-check for conflicts, and re-enter the wait state, potentially being awakened by a slower drop of `conflict_lock` clones. This resulted in an infinite loop under specific timing conditions, which became increasingly likely under high contention, causing the exclusive write access to `list` to be held almost continuously.
- **Final fix.** Finally, we abandoned cloning `existing_lock` to `conflict_lock` and instead directly enqueued the waiter upon detecting a conflict (Figure 5). Similar to the third fix attempt, the process starts the next loop upon waking up, but this time it waits directly on a dedicated waiter/waker pair, which won't wake up other processes on drop.

All fix attempts were conducted using model checking, which quickly verified each method and provided counterexamples for incorrect fixes. Once the bug was correctly fixed in the specification, we applied the corresponding fixes to the code and verified specification-code conformance.

3 CONVEROS Overview

We present CONVEROS, a practical methodology for model checking OS concurrency modules. CONVEROS draws inspiration from recent advancements in TLA$^+$ for writing multi-grained specs [63] and validating traces for conformance [34].

The effectiveness of CONVEROS lies in addressing spec-code discrepancies through conformance checking, which enables model checking to detect real code bugs. Applying CONVEROS involves three main steps: (1) formal specification (§4), (2) conformance checking (§5), and (3) model checking. Figure 6 illustrates the overall workflow for verifying a concurrency module using CONVEROS. For step (1), developers write a PlusCal specification derived from

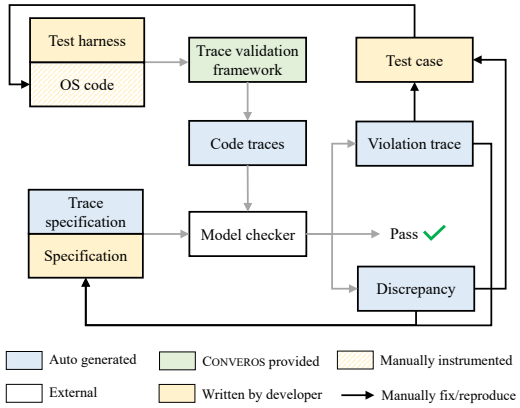


Figure 6: An overview of CONVEROS's workflow

the code (e.g., Figure 2). For step (2), They manually instrument the module and create a test harness to run it for trace collection (e.g., Figure 7). When discrepancies are found during conformance checking, developers manually revise the specification or code. Step (3) involves model checking the revised specification and manually writing test cases to reproduce any reported violations. We automated the generation of trace specifications, code traces, and discrepancy traces for conformance checking, while writing specifications, test harnesses, and test cases, and fixing bugs remain manual tasks. We summarize the overall developer burden in §6.2.

CONVEROS is designed to be practical by making each step small, incremental, and quick to execute, ensuring rapid feedback throughout the process. Consider a lock module that provides `lock` and `unlock` public APIs. The process begins with writing a simple specification that models the public APIs while abstracting away code details (Figure 2). Next, a corresponding test harness is written to use the lock's public APIs in a loop, recording logs for each action[1] (Figure 7). Each log corresponds to a specification's action.

CONVEROS's trace validation framework then automatically runs the test harness to generate code traces. Each trace line represents a code action, recording the action name, process ID and (partial) variable updates. CONVEROS automatically generates a trace specification that constrains the model-checking state transitions by the trace's action name and variable updates. For example, as logged on line 7 in Figure 7, the `start` action constrains the next value of `pc` to `lock`; if it evaluates to a different value, the state is pruned.

CONVEROS subsequently runs the model checker for trace validation (step 2) and property verification (step 3). If the model checker finds discrepancies, developers address the issue in the code or specification, based on whether the specification reflects the intended design (as in this simple example) or detailed code logic. For property violations, both the spec and code must be fixed if the bug is confirmed in the code.

[1] In the code, we use "action" and "event" interchangeably, while in PlusCal specifications, "action" and "label" are used interchangeably.

```
1  pub struct MutexModel;
2  impl TlaModel for MutexModel {
3    fn run(&self, procs: usize, loops: usize) {
4      let mutexlock = Arc::new(Mutex::new(0));
5      let proc_func = move || {
6        for _i in 0..loops {
7          TlaLogger::new("start").next_pc("lock").record();
8          let guard = mutexlock.lock();
9          TlaLogger::new("lock").next_pc("cs").record();
10         TlaLogger::new("cs").next_pc("unlock").record();
11         drop(guard);
12         TlaLogger::new("unlock").record();
13       }
14     };
15     run_procs(procs, proc_func); // Provided by trace framework
16   }}
```

Figure 7: Simplified test harness for a mutex lock model. It implements the `run` method of the `TlaModel` trait, defines a closure using Mutex's public APIs, and logs each action corresponding to a specification's action. `run_procs` executes the closure in parallel and post-processes the log output.

To ensure practicality, CONVEROS addresses two key challenges. First, it makes specifications incremental and scalable through a multi-layered, multi-grained specification approach (§4). Second, it improves automation of trace validation by introducing a new algorithm that allows missing events (§5).

4 Formal Specification

We design multi-layered and multi-grained specifications to make them easy to write, incremental, and scalable for model checking. Additionally, we define correctness properties to verify whether the system behaves as required.

4.1 Multi-Layered Specifications

Multi-layered specifications serve distinct purposes at each layer. High-level specifications capture the design intent (primarily for basic synchronization primitives) and are straightforward to write and verify (e.g., Figure 2). Low-level specifications are developed from the code and reflect implementation details (e.g., Figure 8 corresponds to the Mutex module shown in Figure 9). We do not construct formal refinement mappings [5] between high-level and low-level specifications, as this imposes substantial burdens on developers. Instead, we directly check the properties on the low-level specification.

High-Level Specifications. High-level specifications capture design requirements and are straightforward to write for their public APIs. These specifications are primarily used for basic synchronization primitives like spin and mutex locks, which are simple to write and easy to verify. For example, the PlusCal specification for an abstract lock, shown in Figure 2, models two public APIs: `acquire_lock` and `release_lock`, written in a style resembling imperative procedural code. These specifications ensure the APIs satisfy common locking properties, such as mutual exclusion, and absence of deadlocks and livelocks (as discussed in §4.3).

```
1  variables locked = FALSE, has_woken_of_wakers = <<>>,
2    num_wakers = 0, wakers_lock = FALSE, wakers = <<>>;
3  procedure acquire_lock()
4  variable waiter_ref; (* Local variable *) {
5  fast_path:
6    if (locked = FALSE) { locked := TRUE; return; };
7  slow_path:
8    has_woken_of_wakers := Append(has_woken_of_wakers, FALSE);
9    waiter_ref := Len(has_woken_of_wakers);
10 loop:
11   while (TRUE) {
12 \*enqueue: skip;
13     acquire_wakers_lock:
14       await wakers_lock = FALSE; wakers_lock := TRUE;
15     enqueue_waker:
16       wakers := Append(wakers, waiter_ref);
17     inc_num_wakers:
18       num_wakers := num_wakers + 1;
19     release_wakers_lock:
20       wakers_lock := FALSE;
21   try_lock:
22     if (locked = FALSE) { locked := TRUE; goto drop_waiter; };
23   do_wait:
24     await has_woken_of_wakers[waiter_ref] = TRUE;
25     has_woken_of_wakers[waiter_ref] := FALSE;
26   };
27 drop_waiter:
28   has_woken_of_wakers[waiter_ref] := TRUE; return; }
```

Figure 8: A low-level Mutex PlusCal specification

```
1  pub struct Waker { has_woken: AtomicBool } // has_woken_of_wakers
2  pub struct Waiter { waker: Arc<Waker> }
3  pub struct WaitQueue { num_wakers: AtomicU32, // num_wakers
4    wakers: SpinLock<VecDeque<Arc<Waker>>>} // wakers_lock, wakers
5  pub struct Mutex<T> {lock: AtomicBool, queue: WaitQueue, val: T}
6  impl<T> Mutex<T> {
7    pub fn lock(&self) -> MutexGuard<T> { // acquire_lock in spec
8      self.queue.wait_until(|| self.try_lock())
9    }}
10 impl WaitQueue {
11   pub fn wait_until(&self, cond: ...) -> ... {
12     if let Some(res) = cond() { return res; }     // fast_path
13     let (waiter, _) = Waiter::new_pair();         // slow_path
14     let cond = || { self.enqueue(waiter.waker()); // enqueue
15                     cond() };                     // try_lock
16     loop {                                       // loop
17       if let Some(res)=cond() {/* drop(waiter); */ return res;};
18       waiter.waker.do_wait();                     // do_wait
19     }
20   }
21   pub fn enqueue(&self, waker: Arc<Waker>) {
22     let mut wakers = self.wakers.lock(); // acquire_wakers_lock
23     wakers.push_back(waker);             // enqueue_waker
24     self.num_wakers.fetch_add(1);        // inc_num_wakers
25     /* drop(wakers); */                  // release_wakers_lock
26   }}
27 impl Drop for Waiter {                   // drop_waiter
28   fn drop(&mut self) { self.waker.has_woken.swap(true); }}
```

Figure 9: Simplified Mutex code in ASTERINAS. `try_lock` performs `compare_exchange` and returns a guard on success. `do_wait` blocks until `has_woken.swap(false)` returns true. Comments in yellow correspond to labels in Figure 8's spec.

High-level specifications are designed to provide a quick startup for our approach, enabling incremental development of more detailed low-level specifications based on them. They also support early conformance checking (§5), facilitating efficient application of CONVEROS (§3). Since our focus is on identifying real code bugs in specifications, high-level specifications may be omitted if they are difficult to define (e.g., for modules with complex lock usage).

Low-Level Specifications. Unlike high-level specifications aiming to represent a correct design specification, low-level specifications should closely reflect the code implementation. When bugs exist, the low-level specification enables detection through model checking against correctness properties.

A low-level specification can build on its high-level specification and model code details within the public APIs. For example, Figure 8 presents the `lock` procedure of a mutex's specification, while Figure 9 shows the corresponding code.

Translating Rust code into a PlusCal specification is relatively straightforward because PlusCal supports many programming constructs and concepts, such as `while` loops and procedure calls. However, some Rust-specific characteristics require special handling: (1) Rust automatically invokes `drop` when a variable's goes out of scope, which must be explicitly modeled in PlusCal (e.g., line 27 in Figure 8). (2) Closures in Rust do not have direct support in PlusCal and must be expanded into explicit steps (e.g., the closure on line 8 in Figure 9 is expanded in PlusCal to lines 6 and 22 in Figure 8).

4.2 Multi-Grained Specifications

A key challenge in specifying low-level code is determining the appropriate level of granularity. Fine-grained specifications can lead to state-space explosion, while coarse-grained ones risk missing concurrency bugs. To address this, we adopt the methodology of multi-grained specifications from Remix [63]. Multi-grained specifications use different levels of granularity for composable modules [6, 7, 35, 50]. They include fine-grained modules for detailed code-level behavior and coarse-grained modules that preserve interactions while omitting internal details to reduce state space.

A mixed-grained specification combines fine-grained and coarse-grained modules and aims to mitigate state explosion by verifying selective concurrency interleavings without compromising the ability to detect bugs. This is achieved through the interaction-preserving principle [63], i.e., for each module, only the internal part can be omitted while the interactions with other modules must be preserved. It enables verifying the complex concurrency code module by module and abstract fine-grained actions in verified modules to coarse-grained actions. For example, the `acquire_lock()` procedure in the Mutex module (Figure 8) uses five variables. When the Mutex is integrated into another module, only the `locked` variable, which represents the lock guard, interacts externally and is preserved, while other internal variables are abstracted. Similarly, the SpinLock module is coarsened by preserving only the `wakers_lock` variable (Figure 8), which interacts with the Mutex module, while omitting internal details.

Following the mixed-grained specification approach, we analyze dependencies for each module and adopt a stepwise, modularized process. We first specify and verify basic synchronization primitives, such as SpinLock and RwLock, using fine-grained specifications. For modules built on top of these verified primitives, we coarsen their specifications to abstract

away internal details, while continuing to specify and verify the module's own logic with fine-grained actions.

Fine-grained specifications are written based on high-level specifications, capturing the low-level code logic. A key challenge is determining atomic actions. In PlusCal, labels define atomic actions and can be placed in any statement. In our case, the guiding principle is that each modification to a shared variable should be treated as an atomic action because such modifications may be immediately observed by other threads, and lead to interleavings. For example, in Figure 8, we insert labels such as `enqueue_waker` and `inc_num_wakers`, corresponding to the operations at lines 23 and 24 in Figure 9. Additionally, we abstract unrelated low-level scheduling mechanisms. For instance, the `do_wait` code that parks the kernel thread (line 18 in Figure 9) is assumed correct and modeled as a single atomic action (line 23 in Figure 8).

Once fine-grained specifications are verified, their actions can be composed into coarse-grained actions. This typically requires minimal rewriting compared to Remix [63], as high-level specification actions can often be reused if they follow the interface-preserving principle. PlusCal and TLA$^+$ natively support action composition, allowing multiple fine-grained actions to be composed into a single coarse-grained action. For example, in PlusCal, removing labels merges consecutive actions into one atomic action, while TLA$^+$'s action composition [53] composes multiple actions in a single step.

By employing multi-grained specifications, we can achieve incremental modeling and scalable verification, without missing bugs thanks to the interaction-preserving principle.

4.3 Correctness Properties

We define correctness properties to verify whether the system behaves as required. These properties are classified into two categories: *safety* and *liveness*. Safety properties specify that the system never reaches undesirable states (i.e., "something bad never happens"), and liveness properties specify that the system eventually reaches desirable states (i.e., "something good eventually happens").

For concurrency modules, several correctness properties are commonly checked, as demonstrated in Figure 10. Safety properties include mutual exclusion for locks, which asserts that no two processes are in the critical section simultaneously. Liveness properties include absence of deadlock, livelock, and starvation. Deadlock/livelock freedom specifies whenever a process is in the code attempting to acquire a lock, some process must eventually enter the critical section. Starvation freedom specifies that any process attempting to acquire a lock will eventually succeed in entering the critical section.

In addition to common properties, many modules require domain-specific correctness properties. For example, semaphores must ensure that the semaphore count never drops below zero. These specific properties are typically functional safety properties that require domain knowledge.

```
1 MutualExclusion     == \* Safety
2   \A i, j \in Procs: (i /= j) => ~(pc[i] = "cs" /\ pc[j]="cs")
3 DeadAndLiveLockFree == \* Liveness
4   \E i \in Procs: pc[i]="lock" ~> (\E j \in Procs: pc[j]="cs")
5 StarvationFree      == \* Liveness
6   \A i \in Procs: pc[i]="lock" ~> (pc[i]="cs")
```

Figure 10: Common safety and liveness properties

5 Conformance Checking

Verifying conformance between high- and low-level specifications and code increases confidence in verification correctness. In TLA$^+$, conformance checking can be performed using two approaches: top-down [38, 63, 68] and bottom-up [34, 38, 47]. The top-down approach generates specification traces with the model checker and replays them in the code, requiring deterministic control over event interleaving. In contrast, the bottom-up approach captures code execution traces and verifies whether these traces are allowed by the specification. Controlling thread interleavings in OS kernels is challenging, so we adopt and extend the bottom-up trace validation method outlined in §3. This section describes the trace validation technique used to increase confidence that implementation traces conform to specifications with different granularities.

5.1 Trace Validation

To increase confidence that the implementation's behavior conforms to the spec, we build on trace validation [34] by collecting trace logs and creating a trace specification that reuses actions from the high-level or low-level specification to match trace logs. Unlike CCF [47], which checks properties during trace validation, we focus on conformance at this stage, with property checking as optional. Figure 11 shows an example trace log and the trace specification. Trace logs are generated by instrumenting the code to log each specified action, with each log recording a timestamp, event name, process ID, and optional variable updates. To validate traces, the trace specifications execute the corresponding actions (`SpecActions`) under constraints set by the trace log and enforced by `IsEvent`. `IsEvent` advances the trace line number `l`, enforces event constraints, and applies variable update constraints. A trace is valid once `l` exceeds the trace length.

5.2 Trace Collection

To collect traces for concurrency modules, we execute the modules with logging inserted after each code region corresponding to a labeled action. The executions align with the `process` blocks defined in the specification (e.g., lines 15–23 in Figure 2). Since existing unit tests generally do not meet these requirements, we develop lightweight test harnesses (similar to unit tests) to run concurrency modules. One example test harness is shown in Figure 7. The `run_procs` function launches a specified number of kernel threads, each pinned

```
{"timestamp":100,"event":"start","p":2,"pc":[{"op":"Update",
  ↪  "path":[2],"args":["lock"]}]}
{"timestamp":110,"event":"start","p":1,"pc":[{"op":"Update",
  ↪  "path":[1],"args":["lock"]}]}
```

```
\* Simplified actions of the high-level lock specification
start(self) ==
  /\ pc[self] = "start" /\ pc' = [pc EXCEPT ![self] = "lock"]
  /\ UNCHANGED locked
lock(self) ==
  /\ pc[self] = "lock" /\ pc' = [pc EXCEPT ![self] = "cs"]
  /\ locked = FALSE /\ locked' = TRUE
\* Trace specification (below)
VARIABLES l \* Initialized to 1
Trace == LoadTrace("trace.ndjson") \* Loaded from trace file
cur_line == Trace[l]
UpdateVariables(line) ==
  "pc" \in DOMAIN line => pc' = UpdateVar(pc, line.pc) /\ ...
IsEvent(e) ==
  /\ l \in 1..Len(Trace) /\ l' = l + 1
  /\ pc[cur_line.p] = e \* Use PlusCal's `pc` as event constraint
  /\ UpdateVariables(cur_line)
SpecActions(p) == start(p) \/ lock(p) \/ ...
TraceNext == IsEvent(cur_line.event) /\ SpecActions(cur_line.p)
```

Figure 11: Trace log (top) and trace specification (bottom)

```
MissingEvent(Actions(_)) == \* Actions: operator with 1 argument
  /\ UNCHANGED l    \* ps (below): allowed process set
  /\ \E p \in cur_line.ps: UpdateVariables(cur_line) /\ Actions(p)
IsNextEvent(e) ==
  /\ l \in 1..(Len(Trace)-1) /\ l' = l + 2
  /\ pc[next_line.p] = e
  /\ UpdateVariables(next_line)
ExecuteEvent(Actions(_)) ==
  IF cur_line.event /= "MissingEvent"
  THEN IsEvent(cur_line.event) /\ Actions(cur_line.p)
  ELSE \/ IsNextEvent(next_line.event) /\ Actions(next_line.p)
       \/ MissingEvent(Actions)
TraceNext == ExecuteEvent(SpecActions)
```

Figure 12: Missing event algorithm in trace specification

```
TlaLogger::new_missing().cur_pc("lock").record();
let guard = mutexlock.lock();
TlaLogger::new_missing().cur_pc("lock").record();
TlaLogger::new("cs").next_pc("unlock").record();
TlaLogger::new_missing().cur_pc("unlock").record();
drop(guard);
TlaLogger::new_missing().cur_pc("unlock").record();
```

Figure 13: Logging for missing events

to a specific CPU core, executing the defined closure functions. After generation, logs are sorted by timestamps, post-processed (e.g., adding common constraints; improvements such as merging consecutive missing events are discussed in §5.3), and then output. Finally, the configuration (e.g., process set) and trace files required by the trace specification are automatically produced based on the generated logs.

5.3 Our Improvements

Allow Missing Events. The success of trace validation relies on two key assumptions. First, the order of concurrent events is either strictly logged or does not impact the results. Second, the trace length precisely matches the length of an accepted behavior inferred by the model checker, as in the trace validation method proposed by Cirstea et al. [34], which advances the trace line number `l` after matching an event.

However, the first assumption poses challenges in synchronization with shared memory. Modifications to shared variables become immediately visible to other threads, but timestamps may not accurately reflect the actual event timings. This misalignment can cause the incorrect reordering of concurrent events. For instance, in a spinlock scenario, if thread A unlocks and thread B immediately locks the spinlock, the correct trace should show thread A unlocking before thread B locks. Due to timestamp inaccuracies, the order might be reversed in the logs, making the trace invalid. Introducing a global lock to serialize events has been shown to cause problems [38], as it changes system behavior and is prone to deadlock. For example, if thread A holds the global lock and executes the `acquire_wakers_lock` (Figure 8, line 13), which attempts to lock `wakers_lock`, while thread B already holds `wakers_lock` and attempts to acquire the global lock in `inc_num_wakers` (line 17), the two threads can deadlock.

The second assumption requires users to align granularity by composing actions in the trace specification and logging every corresponding specification label. This is burdensome, particularly for low-level specifications with numerous labels (e.g., the `RwMutex` specification contains 47 labels).

We addressed these issues by introducing a new algorithm for inferring missing events, as illustrated in Figure 12, with usage shown in Figure 13. During the instrumentation process for logging, missing events are inserted before and after specific code regions, representing a time window during which some events might have occurred. These missing events can also set constraints, such as allowed events or expected variable updates to guide state exploration. In log post-processing, consecutive missing events are merged into a single event. This ensures that the next event after a missing event is either a normal event (i.e., non-missing) or EOF. When a missing event is encountered, state exploration diverges:

- One branch attempts to match the next (normal) event following the missing event, advancing `l` by two (skipping the missing event and consuming the matched event).
- The other branch explores any enabled actions allowed by the missing event without advancing `l`, so that the missing event remains active in the next step, enabling exploration of additional paths.

As a result, a missing event can correspond to zero or more inferred actions. Figure 13 shows how missing events address timestamp inaccuracies by replacing lines 8–12 in the test harness of Figure 7. To illustrate how missing events reduce logging burden for low-level specifications, one can omit the `cur_pc` constraint that limits the allowed events based on the current `pc`. This allows users to avoid logging all labels at once and instead log them incrementally, as discussed in §5.4.

While this approach allows for greater flexibility, it significantly increases the state space to search, as missing events introduce additional nondeterminism during exploration. To address this, we introduced two pruning constraints:

- **Maximum exploration depth.** A per-event constraint that limits exploration depth for inferred missing events.
- **Convergence points.** A subsequent trace line is marked as a convergence point during post-processing, instructing the model checker to stop exploring states for the corresponding missing event (implemented by storing a constraint globally via `TLCSet`). Between the divergence (the missing event) and the convergence point, all processes allowed by the missing event must match at least one normal event.

These constraints can lead to false positives by prematurely pruning valid paths. To mitigate this risk, We set the maximum exploration depth slightly above the estimated limit. For convergence points, adding more detailed constraints (e.g., more variable updates) for the normal event following the missing event, helps improve the accuracy of convergence points. This prevents overly relaxed conditions from creating false convergence points that prune valid traces.

Cirstea et al. [34] implemented depth-first search (DFS) for trace validation in TLC, which is orders of magnitude faster than breadth-first search (BFS). DFS prioritizes deepening paths that advance `l`, backtracking only when no successors are enabled. Consequently, convergence pruning offers no performance gain in DFS mode, as DFS does not backtrack on genuinely converged paths. However, incorrect pruning at false convergence points may discard valid states and cause false positives. To avoid this, we disable convergence pruning in DFS mode. Conversely, convergence pruning is beneficial for BFS when inferring minimal traces. Without it, BFS becomes impractical, even with a few missing events.

Constrain Variable's Current State. In Cirstea et al. [34], the `UpdateVariables` operator only sets constraints for variables' next states. To constrain the current state of a variable, users previously had to manually modify the trace specification by adding conjunctions in `IsEvent`, as demonstrated in [47]. We simplify this by incorporating support for current state constraints directly into the logging mechanism and the TLA$^+$ trace library. These constraints can check not only equality but also other conditions, such as inequality, greater-than, or subset relationships. Our test framework automatically generates methods to set these constraints, such as `cur_pc` in Figure 13, which constrains the current `pc` state.

Automatic Generating Trace Specifications. CONVEROS automatically generates trace specifications by parsing TLA$^+$ specifications compiled from PlusCal. These specifications follow a standardized format where each action includes an event name stored in `pc` and a single argument representing the process, and each variable is defined using consistent keywords. These elements are easily extracted and incorporated into the trace specification (e.g., lines 13 and 18 in Figure 11). By using missing events for aligning granularities (§5.4) and setting current-state constraints during the logging process, we eliminate the need to manually modify trace specifications, enabling fully automatic generation.

Automatic Generating Debugging Trace for Discrepancies. Incorrect logging, unsound pruning, or genuine discrepancies can cause trace rejection, necessitating debugging to identify the issues. The work [34, 47] leverages the TLA$^+$ debugger to debug discrepancies by allowing users to step through formula evaluations and generate all unsatisfied states for trace comparison, but it still needs manual intervention when running batches of trace validations that frequently fail.

CONVEROS automates discrepancy trace generation by performing a second run with the longest unmatched trace line number. When the first run fails, the line number is set as a breakpoint in the improved trace specification for the next run. We use the `ENABLED` operator in TLA$^+$ to determine if an event can be matched under constraints before applying undiscardable constraints (which assign the variable's next state). If an event cannot be matched and the trace line number reaches or exceeds the breakpoint, the trace specification searches for a possible missing event without applying trace line constraints and triggers an assertion failure. This causes TLC to produce a counterexample consisting of the longest inferred trace with the unsatisfied state as the final state. Additionally, we periodically print the longest matched trace line number and automatically visualize the failed trace validation state space graph when it is small (e.g., fewer than 100 states).

Although we can construct multiple unsatisfied states after the longest matched trace line, a single constructed unsatisfied state is typically sufficient for debugging. This is because the missing event limits the exploration to the process specified in the trace line, and within a single process, execution is usually deterministic with no multiple successor states.

5.4 Granularity Alignment

Our specification approach is multi-layered and multi-grained, producing specifications with varying granularities. Users must ensure that each label corresponds to a specific code location during development. Trace validation requires adding logging for all these code locations, which supports all granularities. However, logging for low-level specifications with numerous labels can be burdensome and error-prone. To address this, we adopted an iterative and incremental process.

High-level specifications model public APIs and abstract internal implementations, allowing logging in the test harness without instrumenting the APIs themselves (Figure 7). Due to timestamp inaccuracies, events involving atomic variables and concurrent updates are marked as missing to let TLC infer a valid execution order (Figure 13).

For low-level specifications, we can reuse the high-level test harness for preliminary conformance checks by marking public API events as missing in logging. For example, in Figure 7, the `lock` and `unlock` (i.e., `drop(guard)`) events need to be marked as missing without constraining the event (i.e., the current `pc`), enabling trace validation without instrumenting these APIs. This helps detect major discrepancies early, such

as missing implicit logic like drop in the specification. We then iteratively log internal API labels, using missing events as placeholders for labels that are not yet fully logged. As more labels are logged, trace validation can uncover increasingly subtle discrepancies, such as incorrect condition checks, as illustrated in Figure 16 and discussed in §6.3.

After logging, we aggregate trace logs to assess event coverage. For uncovered labels, we analyze why the corresponding code paths are not triggered and add assertions to the specification so that if those paths are reached, TLC generates traces that help diagnose and resolve the issues.

5.5 Bug Confirmation

After trace validation, we conduct model checking to detect property violations. When violations are identified, we create test cases to reproduce the bugs. The violation trace provides serialized sequence and event timing, making test case development straightforward. Typically, this involves adding delays to events to reconstruct the violation path in the code.

If a test case fails to reproduce the bug, it indicates a discrepancy that requires further specification adjustments. Conversely, successfully reproducing the bug confirms its existence. We then fix the specification, verify it, apply the fix to the code, and rerun trace validation to check conformance.

Additionally, running test cases may uncover by-product bugs. Each test is executed repeatedly in a debugging environment to confirm the fix, which can occasionally reveal additional issues such as kernel panics and hangs. When such by-product bugs are identified, they are addressed to enhance overall system robustness.

6 Evaluation

To demonstrate CONVEROS's practicality, we applied it to 12 concurrency modules in ASTERINAS, evaluating bugs discovered, verification effort, and discrepancy detection. We dedicated 4 person-months to model checking about 4,000 lines of ASTERINAS code and identified 20 bugs. This includes 9 bugs discovered automatically through model checking and 11 by-product bugs discovered during diagnosis, and they were all confirmed by the developers. Among these, 8 model-checking bugs and 6 by-product bugs were fixed, with the remaining unfixed due to low priority. All bugs were detected under a model checking configuration with 2-5 processes, and each one was detected within 3 minutes on a laptop equipped with a 16-core CPU, 32GB of RAM, and 22 parallel threads.

6.1 Case Study

Table 1 summarizes the bugs we found. We examine two case studies to demonstrate how we apply CONVEROS to identify and fix these bugs. Analyses of the other bugs are provided in Appendix A.

Case Study 1: Mutex. We found a mutual exclusion bug in the Mutex module that caused severe data races, allowing multiple threads to enter the critical section and obtain multiple mutable references through unsafe. Figure 14 shows the buggy code. The root cause was an eagerly evaluated MutexGuard in the then_some argument, which was unintentionally dropped when the lock was not acquired (line 3). The drop method for MutexGuard then unlocks the mutex (line 7). It was introduced following a compiler linter suggestion to replace then with then_some. Trace validation using the high-level specification successfully found this bug. We fixed it by restoring then and wrapping the MutexGuard construction using a new method inside a lazily evaluated closure (line 4).

```
1  impl<T> Mutex<T> {
2    pub fn try_lock(&self) -> Option<MutexGuard<T>> {
3  -   self.acquire_lock().then_some(MutexGuard { mutex: self })
4  +   self.acquire_lock().then(|| MutexGuard::new(self))
5    }}
6  impl<T> Drop for MutexGuard<T> {
7    fn drop(&mut self) { self.mutex.unlock(); }}
8  impl<T> DerefMut for MutexGuard<T> {
9    fn deref_mut(&mut self) -> &mut T {
10     unsafe { &mut *self.mutex.val.get() }
11   }}
12 impl bool {
13   pub fn then_some<T>(self, t: T) -> Option<T> {
14     if self { Some(t) } else { None }
15   }}
```

Figure 14: Mutex mutual exclusion bug and fix. MutexGuard is incorrectly dropped when acquire_lock fails.

Case Study 2: RwLock. We found a livelock/starvation bug in the RwLock module that can cause temporary kernel hangs under high contention (Figure 15). The RwLock is a spinning lock with its state represented by an AtomicUsize, encoding readers, an upgradable reader, and a writer. A writer can downgrade to an upgradable reader, or vice versa. The root cause was that the try_read method for acquiring a read lock was non-atomic. This method uses fetch_add (line 2) to increase the reader count, and if failed to acquire the read lock, it reverts with fetch_sub (line 6). The try_downgrade method fails at the intermediate state created by try_read, as it requires the lock state to match exactly that of a writer.

```
1  pub fn try_read(&self) -> Option<RwLockReadGuard<T>> {
2    let lock = self.lock.fetch_add(READER);
3    if lock & WRITER == 0 {
4      Some(RwLockReadGuard { inner: self })
5    } else {
6      self.lock.fetch_sub(READER);
7      None
8    }}
9  fn try_downgrade(mut self)->Result<RwLockUpReadGuard<T>,Self> {
10   let inner = self.inner.clone();
11   let res = self.inner.lock
12     .compare_exchange(WRITER, UPREADER);
13   if res.is_ok() {
14     drop(self);
15     Ok(RwLockUpReadGuard { inner })
16   } else {
17     Err(self)
18   }}
```

Figure 15: RwLock livelock/starvation bug

ID	Module	Violation/Manifestation	Description
Found by CONVEROS			
#1	RangeLock [8]	Deadlock-free	Lost wakeup due to waiting on an outdated wait queue
#2	Mutex [9]	Mutual exclusion	Incorrect guard construction in `lock` causing unintended `unlock` on `drop`
#3	RwLock [10]	Livelock/Starvation-free	Non-atomic `read()` operation racing with `downgrade()`
#4	RwMutex [11]	Deadlock-free	Wrong wakeup condition causing a dropped upreader to never wake up a waiter
#5	Semaphore [12]	Semaphore count ≥ 0	Semaphore count decreases below 0 due to unhandled multi-operation `semop()`
#6	Futex [13]	Deadlock-free	Signal or timeout leaving outdated futex item uncleared, causing lost wakeup
#7	Flock [14]	Deadlock-free	Incorrect wait condition causing permanent wait
#8	Pipe [15]	Write atomicity	Writes smaller than `PIPE_BUF` size are not guaranteed to be atomic
#9	TTY [16]	Deadlock-free	Circular dependencies from incorrect locking order causing TTY hang
By-product bugs			
#10	Semaphore [17]	(Found during modeling)	Incorrect initialization of the waiter count, leading to wrong lookup results
#11	Futex [18]	(Found during modeling)	Poor hash algorithm leads to uneven key distribution, overloading a few buckets
#12	Atomic mode [19]	Kernel panic	Misuse of spinlock/mutex breaks atomic mode
#13	IRQ [20]	Stack overflow	Improper IRQ enabling during the bottom half leads to nested IRQs
#14	TLB [21]	Kernel hang	Deadlock caused by a spinlock used in IRQ without disabling IRQ outside it
#15	Trap [22]	Kernel panic and hang	Uncleared direction flag (`rflags.DF`) leads to memory corruption on traps
#16	Task switch [23]	User process crash	Missing FPU state save/restore causes memory corruption on task switch
#17	Sendfile syscall [24]	User data truncation	Short writes cause data truncation when input is read but not fully written
#18	Kernel tests [25]	Kernel test hang	Assertion in kernel test threads leads to infinite error message unwraps
#19	Kernel threads [26]	Kernel panic	A new scheduler feature breaks CPU affinity setting
#20	Logging [27]	Kernel hang	Logging allocating memory during low-memory rescue triggers recursive rescues

Table 1: Bugs found. "Violation" indicates violated properties, while "Manifestation" indicates by-product bug manifestations.

Both `try_read` and `try_downgrade` are repeatedly invoked in surrounding loops, resulting in a livelock where neither operation succeeds. Under high contention, the probability of encountering this issue rises sharply. In our test case, the `downgrade` method took 26 seconds to complete with 30 readers. This type of liveness bug is challenging to detect through testing but can be effectively identified by CONVEROS.

6.2 Verification Effort

Table 2 summarizes key verification effort metrics for the modeled concurrency modules, including lines of code, specification size, number of variables and actions, and the estimated effort in person-days based on Git history.

Our multi-grained specification approach allows coarsening of verified components, and non-essential code such as logging is excluded as well, resulting in a low specification-to-code ratio ranging from 0.3 to 2.3 across 12 modules. Meanwhile, PlusCal effectively supports fine-grained modeling in an imperative style. These factors make modeling each module cost-effective and relatively straightforward, averaging 330 lines of code and 3.6 person-days per module. Variations in effort are mainly due to module complexity and required domain knowledge; for example, the SysV semaphore module took 9 person-days because of its intricate semantics.

Trace validation efforts benefited from improvements to our validation framework, including automation and usability enhancements such as allowing missing events. Developing the trace validation framework took 12 person-days, which included 681 lines of Rust and 396 lines of Python code. On average, trace validation took 2 days per module. Early efforts required more time (e.g., SpinLock took 5 days), as we debugged and refined the framework. Refining the approach into manageable steps significantly sped up later validations.

Writing test cases for confirming and fixing bugs typically took less than a day. Overall, one developer with TLA$^+$ expertise contributed four full-time person-months to writing specifications, developing the trace validation framework, performing trace validation and model checking, and debugging, demonstrating CONVEROS's cost-effectiveness.

6.3 Discrepancies

We identified around 15 modeling errors that caused discrepancies throughout the workflow, especially during trace validation. Trace validation proved effective in uncovering subtle issues that did not trigger obvious property violations. Other discrepancies were easier to detect, as they caused violations during early model checking in specification development.

For example, a discrepancy in RwMutex is illustrated in Figure 16. The `try_upread` method uses `fetch_or` to set the `UPREADER` bit and returns the previous lock value, retaining only the `WRITER` and `UPREADER` bits. In our low-level specification, the lock state was divided into variables like `upreader_lock` and `writer_lock`. We incorrectly modeled the returned lock to retain only the previous `upreader_lock` state (line 4) and mistakenly checked the current `writer_lock` state (line 8) to determine the success condition.

Trace validation identified this discrepancy in a trace where another process released the write lock between the `fetch_or` and `check` actions. In this trace, the `check` action reported unsuccessful lock acquisition because the `WRITER` bit was set in the previous lock value. However, the last unmatched state in the discrepancy trace indicated that the upread lock was successfully acquired, revealing the modeling error.

Code		Specification			Est. Effort	
Module	**#LOC**	**#LOC**	**#Var.**	**#Act.**	**Spec.**	**Conf.**
SpinLock	147	49	1	5	1.5	5
Mutex	113	89	4	14	4	3
RwLock	404	261	5	19	5	2.5
RwMutex	202	460	9	47	3.5	2.5
CondVar	238	199	6	25	3.5	1.5
Semaphore	622	490	10	37	9	2
PageCursor	468	297	10	21	5	-
Pipe	392	143	7	17	1.5	1
RangeLock	457	529	9	24	4	2
Flock	144	176	7	12	1.5	0.5
Futex	469	201	5	21	4	1
TTY	309	138	4	20	1	0.5
Total	3965	3032	77	262	43.5	21.5

Table 2: Verification effort. "#Var." and "#Act." denote the number of variables and actions respectively. "Spec." and "Conf." represent specification and conformance efforts (in person-days). Conformance checking was skipped for PageCursor as the module is undergoing refactoring.

```
pub fn try_upread(&self) -> Option<RwMutexUpReadGuard<T>> {
  let lock = self.lock.fetch_or(UPREADER) & (WRITER | UPREADER);
  if lock == 0 {
    Some(RwMutexUpReadGuard { inner: self })
  }
```

(a) RwMutex code

```
 procedure try_upread()
 variable prev_lock; {
 fetch_or:
-  prev_lock := upreader_lock;
+  prev_lock := <<upreader_lock, writer_lock>>;
   upreader_lock := TRUE;
 check:
-  if (~prev_lock /\ ~writer_lock) {
+  if (~prev_lock[1] /\ ~prev_lock[2]) {
   success:
     role[self] := UPREADER;
     return;
   }
```

(b) Discrepancy and fix in the RwMutex specification

Figure 16: RwMutex discrepancy: code and specification

7 Discussions

Assurance. CONVEROS provides assurance that, within the explored state space, the model-checked code satisfies specified safety and liveness properties, assuming it conforms to its low-level specification. To improve this conformance, CONVEROS performs trace validation between code and specification. In addition, high-level specifications help scale to larger systems by enabling compositional verification in mix-grained specifications, where each component can be verified assuming the correctness of the components it interacts with.

False Positives and False Negatives. The final bugs reported by CONVEROS's model checking stage did not include false positives, all detected bugs were reproduced in the bug confirmation. However, the trace validation stage can report false positives, mainly due to incorrect logging (e.g., incorrect logging of a variable's value) and unsound pruning (e.g., stopping exploration prematurely due to a maximum depth constraint). To ease diagnosis, discrepancy traces are automatically generated (§5.3). False negatives may result from unmodeled components or incomplete exploration of the state space, which are inherent limitations of model checking. In addition, CONVEROS may miss bugs, as trace validation does not prove refinement and may accept traces that do not conform to the specification. CONVEROS mitigates these issues through multi-layered and multi-grained specifications. Writing specifications also enforces clear thinking, precise designs, and unambiguous documentation, increasing confidence in correctness. To further reduce the risk of undetected discrepancies, we compute trace event coverage (§5.4).

Generalizability. The design of CONVEROS, including its formal specification approach and trace validation techniques, is generic and can be extended beyond OS concurrency modules. It is suitable for verifying systems with clear correctness properties, such as crash safety in file systems or fault tolerance in distributed systems. For example, applying CONVEROS to a distributed system follows a similar workflow as shown in Figure 6. Core components can be reused, including the multi-layered, multi-grained specification approach, incremental trace validation with support for missing events, and automatic generation of trace specifications and discrepancy traces. In contrast, system-specific elements must be developed anew, including domain-specific specifications (e.g., modeling both intra-node and inter-node concurrency such as faults and messages as separate `processes` in PlusCal), logging instrumentation, and test harness.

8 Related Work

CONVEROS model-checks ASTERINAS's concurrency modules using multi-layered, multi-grained specifications, building on recent advances in specification development [63] and trace validation [34, 38, 47]. Remix [63] model-checks distributed systems using multi-grained specifications and checks conformance by replaying specification traces against the implementation. In contrast, CONVEROS targets shared-memory OS concurrency, employs both high-level and low-level specifications, and validates code traces within the specification. It further improves coarse-grained specification construction by leveraging TLA$^+$'s action composition. We discuss related work on conformance checking in TLA$^+$, and on model checking, deductive verification, and testing for OS.

Conformance Checking. Prior work has explored conformance checking in TLA$^+$ using trace validation [34, 38, 41, 47, 65, 69] and test case generation [38, 63, 68, 70]. Tasiran et al. [69] first validated execution traces against TLA$^+$ specifications. Pressler [65] formalized trace validation as a refinement check, and Davis et al. [38] applied it to MongoDB (MBTC). However, they found MBTC impractical for conformance, as refinement requires complete state logging and a strict mapping between code and specification actions–both

are often infeasible. Cirstea et al. [34] improved trace validation by leveraging TLA$^+$'s nondeterminism to infer missing states and implementing action composition to align granularity, proving effective in CCF [47]. Building on these techniques, CONVEROS introduces a new missing-event algorithm that enables three key improvements. First, it allows inference of event ordering in shared-memory concurrency, where exact timings are hard to trace. Second, it adopts an iterative and incremental workflow for aligning granularity, and improving usability and automation. Unlike prior work requiring manual trace specifications, CONVEROS automatically generates them and only requires incremental code logging. Finally, it enhances discrepancy debugging by automatically reconstructing traces leading to the next unmatched event.

Davis et al. [38] also check conformance by generating test cases from the TLC state space, and attribute its success to the specification being written close to the implementation, namely a low-level specification in our terms. This observation is further supported by SandTable [68] and Remix [63], which check conformance in distributed systems using low-level specifications. Mocket [70], by contrast, uses a high-level specification for conformance but encounters challenges aligning granularities. They report two conformance issues caused by generalizations in common TLA$^+$ modeling patterns, which led them to revise the specification to more closely reflect the implementation. These works suggest that high-level specifications may not always be sufficient to fully guide concrete executions. CONVEROS instead uses trace validation to check code traces against the specification, supporting logging missing events for flexible granularity alignment, which works for both high-level and low-level specifications.

OS Model Checking. Model checking has been extensively studied for detecting bugs in various OS components [4, 29, 30, 39, 42, 52, 58–60, 62, 73, 74]. The most closely related work is C2TLA+[58], which statically translates concurrent C code to TLA$^+$ for property checking, and uses refinement checking to ensure the correctness of a manually written abstraction that helps reduce the state space. In contrast, CONVEROS dynamically checks conformance through trace validation, which additionally can uncover by-product bugs in the process. Moreover, CONVEROS uses multi-layered specifications to ease the writing burden and multi-grained specifications to reduce the model checking search space.

CMC [60] is an explicit-state model checker capable of verifying code extracted from OS kernels. It was later extended to support running the Linux kernel directly within its environment [59, 73], effectively detecting numerous bugs in network protocol implementations and file systems. EXPLODE [74] applies in-situ model checking to validate file systems by running their implementations as-is, reducing both the effort needed to adapt code for verification and false positives introduced by modifications. VSync[62] extends stateless model checking with await-checking to verify OS synchronization primitives under weak memory models. Kani [4] is a bounded model checker for Rust programs, effective in verifying unsafe code and memory safety, but it lacks support for concurrency and functional verification. Compared to previous approaches that directly model check the system implementations, CONVEROS improves the efficiency of model checking by first model checking multi-layered, multi-grained specifications and then confirming bugs at the implementation level. To make it practical, CONVEROS provides an approach for writing high-quality specifications using trace validation.

OS Deductive Verification. Recent work has made great progress in formally verifying OS components using deductive verification [33, 36, 37, 44–46, 51, 55–57, 67, 71, 75]. These approaches offer very strong correctness guarantees but often require significant manual proof effort–commonly 5X to 20X the size of the implementation. CONVEROS takes a different approach: it uses model checking to verify an existing, concurrent OS without requiring any manual proofs. While CertiKOS also adopts a multi-layered strategy, it uses refinement proofs to ensure that low-level code meets top-level specifications. In contrast, CONVEROS uses multi-layered specifications to reduce specification effort by modeling top-down, and multi-grained specifications to reduce the search space of model checking.

OS Testing. Recent research has proposed systematic testing techniques for OS concurrency by exploring different interleavings of kernel events [40, 43, 48, 49, 72]. Despite being effective in detecting concurrency bugs, it is hard for testing to exhaustively explore the (bounded) space of system states. CONVEROS provides a systematic and efficient way to explore the system state space using model checking.

9 Conclusion

This work presented CONVEROS, a practical methodology for verifying OS concurrency modules using model checking. CONVEROS employs a multi-layered, multi-grained specification approach and enhanced trace validation to improve usability and automation, enable incremental conformance checking, and scale verification. We show that CONVEROS is both practical and effective: in just four person months, we verified 12 concurrency modules and uncovered 20 bugs, including some that were difficult to detect and fix. We hope that CONVEROS leads to a practical step toward more accessible and efficient verification of real-world systems.

Acknowledgments

We thank the anonymous reviewers, and our shepherd, Murat Demirbas, for their insightful comments. We thank Hongliang Tian, Junyang Zhang, Yuwei Liu and Ruihan Li for valuable assistance in resolving reported issues. Ruize Tang, Yu Huang, and Xiaoxing Ma were supported by the National Natural Science Foundation of China (62025202, 62372222); Ruize Tang was also supported by Ant Group Research Intern Program.

References

[1] TLC and TLA+ Toolbox. https://github.com/tlaplus/tlaplus, 2024.

[2] Asterinas. https://github.com/asterinas/asterinas/, 2025.

[3] Asterinas Deadlock Issues. https://github.com/asterinas/asterinas/issues?q=deadlock+sort%3Acreated-asc, 2025.

[4] Kani Rust Verifier. https://github.com/model-checking/kani, 2025.

[5] M. Abadi and L. Lamport. The Existence of Refinement Mappings. In *Proceedings of the 3rd Annual Symposium on Logic in Computer Science (LICS)*, July 1988.

[6] M. Abadi and L. Lamport. Composing Specifications. *ACM Transactions on Programming Languages and Systems*, 15(1):73–132, Jan. 1993.

[7] M. Abadi and L. Lamport. Conjoining Specifications. *ACM Transactions on Programming Languages and Systems*, 17(3):507–535, May 1995.

[8] Asterinas-01. [BUG] Range lock lost wakeup due to waiting on an outdated wait queue. https://github.com/asterinas/asterinas/pull/1466, 2024.

[9] Asterinas-02. [BUG] Incorrect Mutex guard construction in lock() causing unintended unlock(). https://github.com/asterinas/asterinas/pull/1279, 2024.

[10] Asterinas-03. [BUG] RwLock livelock caused by non-atomic read() operation racing with downgrade(). https://github.com/asterinas/asterinas/issues/1297, 2024.

[11] Asterinas-04. [BUG] RwMutex lost wakeup due to wrong wakeup condition. https://github.com/asterinas/asterinas/issues/1303, 2024.

[12] Asterinas-05. [BUG] Semaphore count decreases below 0 due to unhandled multi-operation semop(). https://github.com/asterinas/asterinas/issues/1370, 2024.

[13] Asterinas-06. [BUG] Futex wait lost-wakeup due to uncleared futex item when failure. https://github.com/asterinas/asterinas/issues/1587, 2024.

[14] Asterinas-07. [BUG] Flock permanent wait due to incorrect wait condition. https://github.com/asterinas/asterinas/issues/1474, 2024.

[15] Asterinas-08. [BUG] Pipe write atomicity is not guaranteed. https://github.com/asterinas/asterinas/issues/1554, 2024.

[16] Asterinas-09. [BUG] TTY hang due to circular lock dependencies. https://github.com/asterinas/asterinas/issues/1588, 2024.

[17] Asterinas-10. [BUG] Wrong semctl() lookup results due to incorrect initialization of the waiter count. https://github.com/asterinas/asterinas/issues/1330, 2024.

[18] Asterinas-11. [BUG] Uneven futex key distribution due to poor hash algorithm. https://github.com/asterinas/asterinas/issues/1641, 2024.

[19] Asterinas-12. [BUG] Misuse of spinlock/mutex breaks atomic mode. https://github.com/asterinas/asterinas/issues/1483, 2024.

[20] Asterinas-13. [BUG] Stack overflow caused by nested IRQs. https://github.com/asterinas/asterinas/issues/1648, 2024.

[21] Asterinas-14. [BUG] Deadlock in TLB caused by a spinlock used in IRQ without disabling IRQ outside it. https://github.com/asterinas/asterinas/issues/1602, 2024.

[22] Asterinas-15. [BUG] Uncleared direction flag (rflags.DF) leads to memory corruption on traps. https://github.com/asterinas/asterinas/issues/1606, 2024.

[23] Asterinas-16. [BUG] Missing FPU state save/restore causes memory corruption on task switch. https://github.com/asterinas/asterinas/issues/1619, 2024.

[24] Asterinas-17. [BUG] Sendfile() short writes cause data truncation when an input file descriptor is read but not fully written. https://github.com/asterinas/asterinas/issues/1542, 2024.

[25] Asterinas-18. [BUG] Assertion in kernel threads leads to infinite error message unwraps in test environment. https://github.com/asterinas/asterinas/issues/1584, 2024.

[26] Asterinas-19. [BUG] A new scheduler feature breaks kernel thread CPU affinity setting. https://github.com/asterinas/asterinas/pull/1697, 2024.

[27] Asterinas-20. [BUG] Deadlock due to logging in low-memory rescue triggering rescue again. https://github.com/asterinas/asterinas/issues/1738, 2024.

[28] Asterinas-PR. Fix lock acquiring and releasing of fcntl. https://github.com/asterinas/asterinas/pull/1217, 2024.

[29] T. Ball, E. Bounimova, R. Kumar, and V. Levin. SLAM2: Static Driver Verification with Under 4% False Alarms. In *Formal Methods in Computer Aided Design (FMCAD)*, Oct. 2010.

[30] T. Ball, E. Bounimova, V. Levin, R. Kumar, and J. Lichtenberg. The Static Driver Verifier Research Platform. In *Proceedings of the 22nd International Conference on Computer Aided Verification (CAV)*, July 2010.

[31] J. Bornholt, R. Joshi, V. Astrauskas, B. Cully, B. Kragl, S. Markle, K. Sauri, D. Schleit, G. Slatton, S. Tasiran, J. Van Geffen, and A. Warfield. Using Lightweight Formal Methods to Validate a Key-Value Storage Node in Amazon S3. In *Proceedings of the 28th ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 2021.

[32] M. Brooker. Fifteen Years of Formal Methods at AWS. In *TLA+ Conference*, Apr. 2024. https://youtu.be/HxP4wi4DhA0.

[33] H. Chen, D. Ziegler, T. Chajed, A. Chlipala, M. F. Kaashoek, and N. Zeldovich. Using Crash Hoare Logic for Certifying the FSCQ File System. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 2015.

[34] H. Cirstea, M. A. Kuppe, B. Loillier, and S. Merz. Validating Traces of Distributed Programs against TLA+ Specifications. In *International Conference on Software Engineering and Formal Methods (SEFM)*, Nov. 2024.

[35] E. M. Clarke, D. E. Long, and K. L. McMillan. Compositional Model Checking. In *Proceedings of the 4th Annual Symposium on Logic in Computer Science (LICS)*, June 1989.

[36] E. Cohen, M. Dahlweid, M. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. VCC: A Practical System for Verifying Concurrent C. In *Theorem Proving in Higher Order Logics (TPHOLs)*, Aug. 2009.

[37] Z. Dai, S. Liu, V. Sjoberg, X. Li, Y. Chen, W. Wang, Y. Jia, S. N. Anderson, L. Elbeheiry, S. Sondhi, Y. Zhang, Z. Ni, S. Yan, R. Gu, and Z. He. Verifying Rust Implementation of Page Tables in a Software Enclave Hypervisor. In *Proceedings of the 29th International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS)*, Apr. 2024.

[38] A. J. J. Davis, M. Hirschhorn, and J. Schvimer. eXtreme Modelling in Practice. In *Proceedings of the VLDB Endowment (VLDB)*, May 2020.

[39] P. N. Devyanin, A. V. Khoroshilov, V. V. Kuliamin, A. K. Petrenko, and I. V. Shchepetkov. Formal Verification of OS Security Model with Alloy and Event-B. In *Abstract State Machines, Alloy, B, TLA, VDM, and Z (ABZ)*, June 2014.

[40] P. Fonseca, R. Rodrigues, and B. B. Brandenburg. SKI: Exposing Kernel Concurrency Bugs through Systematic Schedule Exploration. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Oct. 2014.

[41] D. Foo, A. Costea, and W.-N. Chin. Protocol Conformance with Choreographic PlusCal. In *International Symposium on Theoretical Aspects of Software Engineering*, June 2023.

[42] P. Godefroid. Model Checking for Programming Languages using VeriSoft. In *Proceedings of the 24th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, Jan. 1997.

[43] S. Gong, D. Altinbüken, P. Fonseca, and P. Maniatis. Snowboard: Finding Kernel Concurrency Bugs through Systematic Inter-thread Communication Analysis. In *Proceedings of the 28th ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 2021.

[44] R. Gu, Z. Shao, H. Chen, X. N. Wu, J. Kim, V. Sjöberg, and D. Costanzo. CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Nov. 2016.

[45] R. Gu, Z. Shao, J. Kim, X. N. Wu, J. Koenig, V. Sjöberg, H. Chen, D. Costanzo, and T. Ramananandro. Certified Concurrent Abstraction Layers. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, June 2018.

[46] C. Hawblitzel, J. Howell, J. R. Lorch, A. Narayan, B. Parno, D. Zhang, and B. Zill. Ironclad Apps: End-to-End Security via Automated Full-System Verification. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Oct. 2014.

[47] H. Howard, M. A. Kuppe, E. Ashton, A. Chamayou, and N. Crooks. Smart Casual Verification of the Confidential Consortium Framework. In *Proceedings of the 22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Apr. 2025.

[48] D. R. Jeong, K. Kim, B. Shivakumar, B. Lee, and I. Shin. Razzer: Finding Kernel Race Bugs through Fuzzing. In *2019 IEEE Symposium on Security and Privacy (S&P)*, 2019.

[49] D. R. Jeong, Y. Choi, B. Lee, I. Shin, and Y. Kwon. OZZ: Identifying Kernel Out-of-Order Concurrency Bugs with In-Vivo Memory Access Reordering. In *Proceedings of the 30th ACM Symposium on Operating Systems Principles (SOSP)*, Nov. 2024.

[50] B. Jonsson. Compositional Specification and Verification of Distributed Systems. *ACM Transactions on Programming Languages and Systems*, 16(2):259–303, Mar. 1994.

[51] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. SeL4: Formal Verification of an OS Kernel. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 2009.

[52] M. Kokologiannakis and K. Sagonas. Stateless Model Checking of the Linux Kernel's Read–Copy Update (RCU). *Int. J. Softw. Tools Technol. Transf.*, 21(3): 287–306, June 2019.

[53] L. Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Longman Publishing Co., Inc., Aug. 2002.

[54] L. Lamport. The PlusCal Algorithm Language. In *International Colloquium on Theoretical Aspects of Computing (ICTAC)*, July 2009.

[55] A. Lattuada, T. Hance, J. Bosamiya, M. Brun, C. Cho, H. LeBlanc, P. Srinivasan, R. Achermann, T. Chajed, C. Hawblitzel, J. Howell, J. R. Lorch, O. Padon, and B. Parno. Verus: A Practical Foundation for Systems Verification. In *Proceedings of the 30th ACM Symposium on Operating Systems Principles (SOSP)*, Nov. 2024.

[56] S.-W. Li, X. Li, R. Gu, J. Nieh, and J. Z. Hui. A Secure and Formally Verified Linux KVM Hypervisor. In *2021 IEEE Symposium on Security and Privacy (S&P)*, May 2021.

[57] H. Mai, E. Pek, H. Xue, S. T. King, and P. Madhusudan. Verifying Security Invariants in ExpressOS. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Mar. 2013.

[58] A. Methni, M. Lemerre, B. Ben Hedia, S. Haddad, and K. Barkaoui. Specifying and verifying concurrent C programs with TLA+. In *International Workshop on Formal Techniques for Safety-Critical Systems (FTSCS)*, Nov. 2014.

[59] M. Musuvathi and D. R. Engler. Model Checking Large Network Protocol Implementations. In *Proceedings of the 1st USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Mar. 2004.

[60] M. Musuvathi, D. Y. Park, A. Chou, D. R. Engler, and D. L. Dill. CMC: A Pragmatic Approach to Model Checking Real Code. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI)*, Dec. 2002.

[61] C. Newcombe, T. Rath, F. Zhang, B. Munteanu, M. Brooker, and M. Deardeuff. How Amazon Web Services Uses Formal Methods. *Communications of the ACM*, 58(4):66–73, Mar. 2015.

[62] J. Oberhauser, R. L. d. L. Chehab, D. Behrens, M. Fu, A. Paolillo, L. Oberhauser, K. Bhat, Y. Wen, H. Chen, J. Kim, and V. Vafeiadis. VSync: Push-Button Verification and Optimization for Synchronization Primitives on Weak Memory Models. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Apr. 2021.

[63] L. Ouyang, X. Sun, R. Tang, Y. Huang, M. Jivrajani, X. Ma, and T. Xu. Multi-Grained Specifications for Distributed System Model Checking and Verification. In *Proceedings of the 20th European Conference on Computer Systems (EuroSys)*, Mar. 2025.

[64] Y. Peng, H. Tian, Z. Junyang, R. Li, C. Chen, J. Jiang, J. Xian, Y. Luo, X. Wang, C. Xu, D. Zhou, S. Yan, and Y. Zhang. Asterinas: A Linux ABI-Compatible, Rust-Based Framekernel OS with a Small and Sound TCB. In *Proceedings of the 2025 USENIX Annual Technical Conference (ATC)*, July 2025.

[65] R. Pressler. Verifying Software Traces Against a Formal Specification with TLA+ and TLC. https://pron.github.io/files/Trace.pdf, 2018.

[66] B. Qin, Y. Chen, Z. Yu, L. Song, and Y. Zhang. Understanding Memory and Thread Safety Practices and Issues in Real-World Rust Programs. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, June 2020.

[67] H. Sigurbjarnarson, J. Bornholt, E. Torlak, and X. Wang. Push-Button Verification of File Systems via Crash Refinement. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Nov. 2016.

[68] R. Tang, X. Sun, Y. Huang, Y. Wei, L. Ouyang, and X. Ma. SandTable: Scalable Distributed System Model Checking with Specification-Level State Exploration. In *Proceedings of the 19th European Conference on Computer Systems (EuroSys)*, Apr. 2024.

[69] S. Tasiran, Y. Yu, and B. Batson. Using a Formal Specification and a Model Checker to Monitor and Direct Simulation. In *Proceedings of the 40th Annual Design Automation Conference (DAC)*, June 2003.

[70] D. Wang, W. Dou, Y. Gao, C. Wu, J. Wei, and T. Huang. Model Checking Guided Testing for Distributed Systems. In *Proceedings of the 18th European Conference on Computer Systems (EuroSys)*, May 2023.

[71] F. Xu, M. Fu, X. Feng, X. Zhang, H. Zhang, and Z. Li. A Practical Verification Framework for Preemptive OS Kernels. In *International Conference on Computer Aided Verification (CAV)*, July 2016.

[72] M. Xu, S. Kashyap, H. Zhao, and T. Kim. Krace: Data Race Fuzzing for Kernel File Systems. In *2020 IEEE Symposium on Security and Privacy (S&P)*, 2020.

[73] J. Yang, P. Twohey, D. Engler, and M. Musuvathi. Using Model Checking to Find Serious File System Errors. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI)*, Dec. 2004.

[74] J. Yang, C. Sar, and D. Engler. EXPLODE: A Lightweight, General System for Finding Serious Storage System Errors. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*, Nov. 2006.

[75] Z. Zhou, Anjali, W. Chen, S. Gong, C. Hawblitzel, and W. Cui. VeriSMo: A Verified Security Module for Confidential VMs. In *Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, July 2024.

A Descriptions of Bugs Found by CONVEROS

RwMutex. One deadlock bug was found in the RwMutex module that caused kernel hangs. The RwMutex is a blocking version of RwLock, designed with wakeup mechanisms triggered when a lock holder drops. The root cause of the bug was that the wakeup condition for dropping an upgradable reader as the last reader was incorrectly checked. It assumed that the `fetch_sub` method's return value represented the state after the change, whereas it actually represented the state before the change. As a result, the condition indicating the last reader was never satisfied.

Semaphore. One safety bug was found in the SysV semaphore IPC. The bug was related to the complex design of SysV semaphores, which allows a single `semop` syscall to operate on multiple semaphore numbers with multiple operations. The root cause was the failure to handle cases where multiple operations affected the same semaphore number within a single `semop` call. In these cases, each operation could proceed (e.g., decreasing the count by 1 when the semaphore count was 1), but the overall result required waiting. The precheck to determine if the syscall could proceed failed to detect this case, causing it to proceed without waiting, which led to the semaphore count dropping below zero.

Futex. One deadlock bug was found in the Futex system call, a critical component for constructing user-space synchronization mechanisms like `pthread_mutex_lock`. The bug caused user process hangs due to lost wakeups. When a process waiting on a futex system call was interrupted by a timeout or signal, the futex item was not properly removed from the wait queue. As a result, a subsequent wake operation targeted the leftover futex item instead of a legitimately waiting one, leading to lost wakeups and user process hangs.

Flock. One deadlock bug was found in the Flock module. Similar to RangeLock, which locks specific file regions, Flock locks the entire file. The root cause of the bug was the incorrect use of the `WaitQueue` API to wait for an invalid condition. Specifically, the locking method attempted to wait on a `None` condition when encountering conflicts, intending to resume execution once woken up. However, the `WaitQueue` API does not support such conditions, resulting in an infinite wait.

Pipe. One safety bug was found in the Pipe module, which is implemented using a lock-free ring buffer. While verifying the ring buffer, we found no issues. To ensure correctness, we also wrote test cases to simulate real-world usage. However, a short write issue was identified in the `sendfile` system call, causing user data truncation. Although short writes are allowed in the ring buffer, the POSIX standard requires that pipe writes of sizes up to `PIPE_BUF` be atomic. After learning about this atomicity requirement, we verified it and, unsurprisingly, found it violated. We then implemented a fix and confirmed its correctness through model checking.

TTY. One deadlock bug was found in the TTY driver, which handles input from the keyboard and outputs to the terminal. During testing, we observed that the TTY occasionally became unresponsive or hung completely. Debugging revealed a circular dependency in the spinlock usage. The TTY driver employed four spinlocks, and the circular dependency led to deadlocks. To ensure the correctness of our fix, we modeled the spinlock usage, reproduced the circular dependency issue, and verified the fix.

BIN2WRONG: a Unified Fuzzing Framework for Uncovering Semantic Errors in Binary-to-C Decompilers

Zao Yang
University of Utah

Stefan Nagy
University of Utah

Abstract

Binary *decompilation* is central to many systems tasks that rely on analyzing or modifying closed-source software, such as debugging, performance tuning, and security hardening. Decompilers translate executables into C code with the goal of reconstructing a semantically-equivalent form of the original program's source. Unfortunately, when challenged by intricate program logic, data structures, and diverse executable layouts, decompilers often produce semantically-*wrong* code. Proactively detecting such decompilation defects is critical for ensuring the success of downstream tasks that depend on precise binary analysis. Yet, current methods for assessing decompiler correctness only narrowly explore the variety of source constructs, compilers, optimization levels, executable formats, and *combinations thereof* that influence binary code. Fully guaranteeing decompilation precision—and, by extension, supporting all tasks that hinge on accurate binary-to-source recovery—demands a testing approach that unifies *all* factors affecting binary code, extending practical, systematic correctness testing to all decompilers today.

To accelerate discovery of decompilation defects, this paper introduces BIN2WRONG: a general-purpose decompiler fuzzer combining systematic binary mutation with practical, decompiler-agnostic support. Our approach coalesces *all* factors of binary generation—source, compiler, optimization, and executable format—into a novel, unified testcase structure for mutation. Beyond enabling deeper exploration along these individual dimensions, BIN2WRONG finds unique *combinations* exposing complex, multi-dimensional errors that elude prior decompiler testing approaches. In evaluating BIN2WRONG alongside state-of-the-art decompiler fuzzers Cornucopia and DecFuzzer across seven free and commercial decompilers, BIN2WRONG achieves upwards of **10.39**× and **17.18**× higher binary diversity and **1.16**× and **1.32**× more decompiler code coverage, respectively, whilst uncovering **the most** decompilation bugs. Beyond finding **48 *new* bugs**, **with 30 confirmed**, BIN2WRONG spurred a major redesign of the commercial decompiler Binary Ninja—showing its utility in **uncovering critical defects in mainstream decompilers**.

1 Introduction

Binary *decompilers* are crucial for analyzing, maintaining, and modifying modern systems software, translating raw machine code into more human-readable higher-level representations, most commonly C. By bridging this gap, decompilers become indispensable for tasks that must proceed without a program's original source, such as performance tuning of closed-source components [37], malware analysis [136], security assessment of proprietary kernels [34], and third-party patching of production systems [40]. Popular decompilers today include Binary Ninja [108], Angr [132], Ghidra [78], and RetDec [6].

Decompilers' ultimate goal is recovering code identical or semantically-equivalent to the program's original source. Recent advancements are vastly improving decompilation accuracy through neural networks [30, 141], LLMs [5, 135], and heuristic code restructuring [15, 33]. Yet in practice, decompilers face steep challenges due to the lossy nature of compilation, which strips source-level semantics like type information, function and variable names, and high-level program structures. Moreover, diverse executable formats (e.g., ELF, PE, Mach-O), compiler-specific code layout patterns (e.g. variable arrangement, function ordering), and optimizations (e.g., inlining, loop jamming, dead code elimination) further impede precise decompilation, requiring significant specialized handling. As even subtle mistakes (e.g., a misrecovered branch condition) cause trickle-down failures (e.g., incorrect control flow) in downstream binary analysis tasks, **vetting decompilers' correctness is critical to the success of many systems engineering and reliability efforts today**.

Inspired by the success of automated testing—namely fuzzing—in other software domains, recent works propose testing strategies specially tailored to finding semantic errors in decompilers [20, 54, 89, 142]. While such approaches have historically uncovered bugs, they are overwhelmingly ineffective at finding the vast majority of decompiler defects today. Executables are influenced by four key dimensions: (1) *source*, the fundamental blueprint behind binary generation; (2) *compilers*, which emit and structure the bi-

nary's code; (3) *optimizations*, which restructure, expand, or trim the binary accordingly; and (4) the *executable format*, which encapsulates platform-specific code layout and calling conventions. Yet, despite the equal importance of each of these aspects, current decompiler fuzzers fail to explore them comprehensively—hardcoding themselves to only specific, narrow subsets that ultimately highly overlap with those already-covered by past testing efforts. Worse yet, these restrictive approaches cannot reveal the many decompiler bugs caused by specific source and compilation parameter *combinations*—leaving these edge-case errors unlikely to be found until they inevitably confront decompilers' own users. Thus, preventing the failure of critical efforts reliant on precise decompilation demands **a systematic correctness-testing approach spanning *all* aspects of binary generation—whilst supporting all of today's diverse decompiler ecosystem**.

To accelerate discovery of decompiler errors producing semantically-wrong code, this paper introduces BIN2WRONG: a general-purpose fuzzer that explores *all* aspects of binary generation—source, compiler, optimizations, and executable formats—in a synergistic *and* decompiler-agnostic manner. Unlike prior decompiler fuzzers that view these dimensions individually, BIN2WRONG unifies and mutates them *altogether*, facilitating more flexible and thorough exploration of diverse binary semantics as well as the unique interplay arising from distinct *combinations* of all four factors. When applied to state-of-the-art decompilers, **BIN2WRONG reveals many more decompilation errors** than prior overly-restrictive approaches, assisting decompiler developers in strengthening their tools against complex edge-cases that otherwise derail their users with real-world inaccuracies.

We evaluate BIN2WRONG alongside the state-of-the-art decompiler fuzzers DecFuzzer [54] and Cornucopia [89] on seven free and industrial x86 decompilers: Binary Ninja, Angr, Radare2/Ghidra, Reko, Relyze, RetDec, and Rev.Ng. Overall, BIN2WRONG's holistic, binary-centric mutation strategy vastly outperforms DecFuzzer and Cornucopia in generated binary diversity by upwards of **10.39**× and **17.18**×, respectively. When used to test decompilers, BIN2WRONG's higher-diversity binaries achieve a mean **1.16**× and **1.32**× decompiler code coverage relative to DecFuzzer and Cornucopia—while uncovering **38** and **48** more decompilation semantic errors, respectively. After analyzing BIN2WRONG's **48 newly-found bugs** and reporting them to their decompilers' developers, **30 are so far confirmed or fixed**. Importantly, BIN2WRONG *uniquely* discovered many critical bugs, including one that prompted **a total overhaul** of the industrial decompiler Binary Ninja's core code structuring techniques.

In summary, our paper makes the following contributions:

- **Insights:** We analyze and categorize 64 publicly-reported decompiler semantic defects; and are the first to shed light on the importance of *all four* factors behind binary generation—source, compiler, optimization, and executable format—in facilitating decompiler defect discovery.
- **Challenges:** We weigh the limitations of prior decompiler testing efforts in considering our aforementioned four factors of binary generation; as well as their capability of supporting *all* of today's diverse decompiler ecosystem spanning both free and industrial decompilers.
- **Solutions:** We design BIN2WRONG: the first general-purpose decompiler fuzzer to balance automated, large-scale exploration of diverse binary semantics stemming from distinct source code and compilation parameter combinations—in a fully decompiler-agnostic manner. Beyond supporting virtually all of C's control and data constructs (e.g., floating-point and string literals, `switch-case` statements, `goto` branches, etc.), BIN2WRONG integrates six different C compilers, three binary executable formats, and 5,183 distinct compiler optimizations.
- **Results:** In an evaluation of BIN2WRONG versus state-of-the-art decompiler fuzzers Cornucopia and DecFuzzer, we show that BIN2WRONG achieves the highest-overall binary diversity, code coverage, and semantic bug discovery across seven mainstream binary-to-C decompilers: Binary Ninja, Angr, Radare2/Ghidra, Reko, Relyze, RetDec, and Rev.Ng. Moreover, while prior decompiler testing works explore only limited sets of compilers and optimizations, we present the *first* comprehensive study of the distinct impacts of varying source code—combined with the largest-ever tested set of compilers, executable formats, optimizations, and *their combinations*—on the discovery of semantic errors within contemporary binary decompilers.
- **Impacts:** We report all discovered bugs to their respective decompiler developers. So far, **30** of **48** BIN2WRONG-found bugs are confirmed, with many since fixed. Several BIN2WRONG-uncovered bugs motivated significant changes to mainstream decompilers, including a major restructuring of the commercial decompiler Binary Ninja.

We will publicly release BIN2WRONG, as well as all evaluation artifacts and uncovered bugs at the following URL: `https://github.com/FuturesLab/Bin2Wrong`.

2 Background

This section provides a brief overview of the process of binary decompilation, as well as the most common classes of correctness defects affecting today's various binary decompilers.

2.1 An Overview of Binary Decompilation

Decompilers' goal is *inverting* program compilation: consuming a pre-built executable and recovering its source code [22]. For systems, decompilers critically facilitate human understanding and reasoning about software whose source is unavailable, such as legacy or proprietary components, and obfuscated malware. In the following, we outline and discuss the four main steps central to virtually all decompilers today: **(1) Control Flow Recovery**, **(2) Control Flow Analysis**, **(3) Data Flow Analysis**, and **(4) Code Generation**.

1. **Control Flow Recovery.** Control Flow Recovery aims to understand programs' underlying flow of execution. Instruction disassemblers (e.g., Capstone [3], Intel XED [39]) are used to translate raw binary code into machine instructions—and further—group them into *basic blocks*: sequences of consecutive instructions both targeted-by and terminating-in a *control-flow transfer* (i.e., branches, fall-throughs, calls, or returns). Additional analyses are performed to group basic blocks into functions, culminating in the overall program Control-flow Graph (CFG), where nodes and edges denote basic blocks and control-flow transfers between them, respectively.
2. **Control Flow Analysis.** With the CFG in-hand, decompilers begin abstracting basic block groupings into higher-level *control structures* like loops, switch-cases, and dispatch tables. Critical to CFG analysis is recognizing high-level structures via known patterns: for example, standard techniques for loop identification leverage control-flow dominance between basic blocks to determine the loop's overarching header [90]. Ultimately, recovery of higher-level control structures offers a richer understanding of program semantics, forming the backbone of decompilers' subsequent data flow recovery and code generation steps.
3. **Data Flow Analysis.** Following control structure recovery, decompilers shift focus to examining how *data* is read, written, and propagated throughout program execution. Primary goals of these analyses include identifying distinct data objects, such as function arguments, return values, and both local and global variables, along with determining their associated data *types*. These elements are critical to decompilers' final code generation phase as they directly impact virtually all aspects of program semantics: control-flow logic, data expressions, and more.
4. **Code Generation.** Decompilers' final stage leverages all collected control- and data-flow artifacts to generate the program's final *source* representation. Yet, as all prior steps are inherently undecidable, decompilers' generated code is unlikely to be identical to the program's original source. To this end, decompilers still aim to yield semantically-equivalent source representations which—while visually different from the original—*behaves* the same.

2.2 Semantics-affecting Decompilation Defects

Decompiler defects range from cosmetic inconsistencies affecting human readability of decompiled code—for example, haphazard ordering of `switch-case` labels (Binary Ninja issue [113])—to **insidious *semantics-affecting* errors that completely misrepresent program behavior**. Semantics-affecting defects uniquely burden decompiler users with *incorrect* results, thus impeding the success of downstream systems-oriented binary analysis tasks. To better understand these obstacles, we review public issue trackers of mainstream decompilers and distill reported bugs into four high-level categories of semantics-affecting defects shown in Table 1: **(1) Data Expressions**, **(2) Control-flow Logic & Structures**, **(3) Data Objects, Types, & Values**, and **(4) Compilation-specific Defects**. We briefly discuss these below.

Category	Origin(s)	Decompilers and Examples of Relevant Defects
Data Expression	Instructions	**Ghidra** [75], **Radare2** [82], **Reko** [48], **RetDec** [7, 13]
	Refinement	**Angr** [95, 106], **Binary Ninja** [120, 121, 125], **RetDec** [12]
Control Logic, Structures	If / Else	**Binary Ninja** [114, 126, 130], **Ghidra** [65], **RetDec** [8, 9]
	Loops	**Angr** [94, 99, 103], **Binary Ninja** [112, 116], **Reko** [47]
	Goto	**Angr** [104, 105], **Binary Ninja** [122], **Reko** [44], **RetDec** [11]
	Switches	**Binary Ninja** [111, 115, 116], **Radare2** [84], **RetDec** [10]
Objects, Types, Values	Arguments	**Angr** [93, 101, 102], **Binary Ninja** [117, 123, 127, 129]
	Variables	**Angr** [97, 100], **Binary Ninja** [124], **Ghidra** [72–74]
	Literals	**Angr** [92], **Binary Ninja** [119], **Ghidra** [66, 77], **Reko** [46]
Compile Configs	Code Opts	**Angr** [91, 93, 98], **Binary Ninja** [110], **Ghidra** [71, 76]
	Format	**Angr** [96], **Binary Ninja** [118, 128, 131], **Ghidra** [68, 70]

Table 1: Reported semantic bugs in mainstream decompilers.

- **Defects in Recovering Data Expressions.** Errors in instruction disassembly or handling of instruction semantics often lead to data inconsistencies in decompilers' higher-level code generation. For example, in Ghidra [75], incorrect handling of x86's `FCOMP` (floating-point comparison) results in comparison operands being *swapped*; and as this comparison governs a conditional branch, the code guarded by this branch ends up only being reachable under conditions totally different from the original program's. Additionally, decompilers often refine data expressions using techniques such as constant folding and propagation to enhance code readability [33], conciseness [137], or better align with real-world source code patterns [15]. However, subtle errors in these optimizations also lead to incorrect data. For example, in Binary Ninja [121], constant propagation for x86's `ROL` (rotate-left) and `SHL` (shift-left) instructions inaccurately simulates 32-bit registers, yielding divergent expression results when analyzed on 64-bit systems.
- **Defects in Recovering Control Logic & Structures.** In addition to data expressions, we identify numerous instances of semantics-affecting defects in decompilers related to the recovery of control logic and structures. These include missing bodies for conditional branches (e.g., RetDec [9]); incorrect recovery of loop starting conditions (e.g., Reko [47]); and missing, spurious, or misdirected `goto` branches (e.g., Angr [105]). Handling of `gotos` is particularly contentious—and error-prone—in decompilation research and design, with current decompilers employing vastly different strategies for their recovery and structuring (e.g., [15, 137]). Furthermore, we see that `switch-case` statements remain one of the most universally-challenging control structures across all decompilers, leading to downstream errors that result in unrecoverable code blocks (e.g., Radare2 [84], RetDec [10]), or complicating recovery of other intermingled control logic (e.g., Binary Ninja [116]).
- **Defects in Recovering Data Objects, Types, & Values.**

While recovery and refinement of data expressions pose challenges for decompilers, even identifying and reconstructing their underlying *data objects* (e.g., string, integer, and floating-point variables and literals)—along with their corresponding types and values—presents significant difficulties that directly impact the accuracy of decompiled code semantics. We observe many instances of functions missing arguments (e.g., Angr [101], Binary Ninja [127]) or return values (e.g., Angr [100]), as well as general type incompatibilities on local variables (e.g., Binary Ninja [124], Ghidra [74]). Strings and floating-point literals also present distinct challenges for decompilers, which commonly miss (e.g., Binary Ninja [119]), truncate (e.g., Ghidra [77]), or otherwise erroneously perturb them (e.g., Reko [45]).

- **Defects from Specific Compilation Configurations.** Lastly, we see many issues stemming from *compilation* diversity, such as function recovery defects specific to MacOS Mach-O (e.g., Angr [96], Ghidra [68]) and Windows PE binaries (e.g., Binary Ninja [128]). Additional challenges arise under certain compiler optimizations; for instance, Ghidra [71] and Angr [93] face difficulties with calling convention recovery under MSVC's `/LTCG` (link-time code generation) and GCC's `-Ofast` optimizations, respectively. Furthermore, we see source constructs present unique semantic recovery challenges depending on the executable format they are compiled into, as demonstrated by Binary Ninja's and Ghidra's errors on floating-point function arguments from PE binaries ([69] and [70], respectively).

As binary-oriented systems tasks such as malware analysis, vulnerability discovery, and security hardening all depend on *accurate* decompilation, decompiler developers are working aggressively to fix errors and prevent misleading their users with inaccuracies or failures. However, the lack of a systematic testing approach leaves today's decompiler developers overwhelmingly relying on *manual* bug discovery (e.g., Table 1), limiting their ability to proactively identify critical, semantics-affecting defects. Enhancing the reliability of decompilers thus demands moving beyond cumbersome manual bug-finding and adopting **automated, *practical* testing techniques that thoroughly explore the complex interplay between source *and* compilation diversity** in binary code.

3 Challenges of Testing Decompilers

To automate discovery of semantics-affecting decompiler defects, a handful of decompiler testing efforts have emerged over the years (Table 2): Cornucopia [89], DecFuzzer [54], D-Helix [142], and DSmith [20]. Unfortunately, as we show below, these approaches are fundamentally limited in achieving systematic, practical decompiler defect discovery. In the following, we weigh their trade-offs with respect to addressing the three main challenges of automated decompiler defect discovery: **(1) source construct diversity**, **(2) compilation diversity**, and **(3) supporting *all* decompilers**.

Approach	Source Diversity			Compilation Diversity					Decomp.
	Exprs	Ctrl	Data	Comp	Opts	ELF	PE	Mach-O	Agnostic
DecFuzzer	✔	👎	👎	1	✗	✔	✗	✗	✔
Cornucopia	✔	✔	✔	2	✔	✔	✗	✗	✔
D-Helix	✔	✔	👎	2	👎	✔	✗	✗	✗
DSmith	✔	👎	👎	1	👎	✔	✗	✗	✔
BIN2WRONG	✔	✔	✔	6	✔	✔	✔	✔	✔

Table 2: Survey of state-of-the-art decompiler testing works. ✔ = full support; ✗ = no support; 👎 = limited or poor support.

3.1 Challenge 1: Source Construct Diversity

Varying source constructs pose unique hurdles for decompilation, complicating reconstruction of high-level program logic and structure. For example, both `switch-case` and `if-else` statements are often compiled-down into jump tables [80], challenging decompilers' differentiation of the two. To this end, modern decompilers devote significant analyses and heuristics to recovery and refinement of distinct source constructs [50]. Moreover, decompilers frequently differ in handling specific constructs (e.g., `gotos` [15]), meaning that the same source often produces disagreeing results among decompilers. Thus, maximizing diversity of input source code is crucial to scrutinizing decompilers' underlying capabilities and accuracy in handling real-world binaries.

Limitations: In examining today's leading automated decompiler testing efforts (Table 2), we observe varying support for many of today's most common source code constructs. Although each considers control-flow like loops, conditional branches, function calls and returns, and `gotos`, both DSmith and DecFuzzer entirely omit `switch-case` statements, limiting their ability to thoroughly evaluate decompilers' control-flow recovery. Further, almost all exhibit limitations in supporting various types of literals: DecFuzzer, DSmith, and D-Helix all fail to support floating-point literals; while DecFuzzer and DSmith additionally exclude string literals because their underlying source code generation tooling does not yet support them [139]. Omitting common source constructs drastically reduces the scope of previous methods, **leaving them unable to catch many critical decompiler defects likely to emerge in practice**.

3.2 Challenge 2: Compilation Diversity

Modern compilers have unique code generation strategies and internal algorithms, creating vast differences in their resulting binary code. For example, virtual dispatch tables take on fundamentally disparate forms when compiled by MSVC versus GCC [14]. As a result, identical source overwhelmingly yields *different* binaries when compiled by multiple compilers. This entropy is further increased by compiler *optimizations*, which additionally obscure precise disambiguation of control and data constructs; for example, MSVC's aggressive data and code inlining [60]. Furthermore, platform-specific executable file formats—Windows' PE, Linux's ELF, and MacOS's Mach-O—have fundamentally unique binary layouts,

necessitating decompilers' specialized handling of each [25]. Because reverse engineering—and thus, decompilers—targets *every* software ecosystem, it is thus crucial to identify and characterize the kinds of errors that decompilers encounter across the vast space of possible compilation configurations.

Limitations: While prior decompiler testing works vary in the diversity of the source constructs they examine, they are even more restricted in their exploration of binary compilation parameters—**hardcoding themselves specific, hand-chosen compilation configurations, making them unable to automatically explore *others***. All four cover no more than *two* compilers: D-Helix and Cornucopia focus exclusively on GCC and Clang, while DSmith and DecFuzzer limit themselves to only GCC. In terms of optimizations, Cornucopia examines nearly all possible GCC and Clang optimization levels, whereas the others are more constrained: D-Helix is limited to `O0` and `O2-O3`, D-Smith explores `O0-O3` and `Os`, and DecFuzzer omits optimizations altogether. Additionally, these approaches consider only the Linux ELF executable format, entirely neglecting PE and Mach-O binaries. Moreover, their designs are rigid and hardcoded to specific compilation parameters, **preventing them from automatically exploring the diversity of binaries generated from a single input source code**—and the decompilation defects these variations might introduce—without additional effort and re-engineering.

3.3 Challenge 3: Broad Decompiler Support

Modern binary decompilers exhibit considerable diversity in their implementations, encompassing different programming languages, internal code representations, and degrees of transparency. While many are open-source (e.g., Angr [132], Ghidra [78], and Rev.Ng [24]), some of today's most widely-used decompilers are commercial products and hence remain fully or partially closed-source, limiting insight into their underlying internals (e.g., Binary Ninja [108], IDA Pro [36], and JEB3 [81]). Therefore, a comprehensive vetting approach is required that is supportive of *all* decompilers—regardless of their architectural design variations or transparency.

Limitations: Most previous approaches prioritize scalability by treating decompilers as black-box systems, analyzing only their output C code. In contrast, D-Helix operates directly on the *non-C* Intermediate Representations (IRs) used internally by decompilers prior to final code generation. Yet, such approach requires costly, *per-instruction semantic modeling*—further burdened by the vast differences between decompiler-specific IRs (e.g., Angr's `VEX` vs. Binary Ninja's `LLIL`, `MLIL`, and `HLIL` IRs). Currently, D-Helix only supports Ghidra's `P-code` and Angr's `VEX` IRs, leaving others (e.g., Rev.Ng's LLVM IR [24]) **beyond its reach**. According to its developers [142], extending D-Helix to *other* decompilers' IRs—which often differ significantly in semantics [42]—demands considerable manual effort and domain expertise on the order of **40 days' time per unique decompiler integration**.

Motivation: Thoroughly vetting binary decompilers requires a multifaceted testing approach that accommodates ***all* decompilers** whilst accounting for every factor influencing executable code generation. We propose a holistic view of binary mutation—**integrating diverse source code, multiple compilers, and varying optimizations and executable formats *simultaneously***—to maximize the detection of semantic decompilation defects across *different* decompiler platforms, irrespective of their internal differences or level of transparency.

4 BIN2WRONG: Systematic, Practical Fuzzing for Binary Decompilers

To tackle the limitations of prior decompiler testing approaches, we present BIN2WRONG (Figure 1): a general-purpose fuzzer for uncovering decompilation semantic bugs whilst supporting *any* decompiler—irrespective of internals or transparency. We detail our high-level design of BIN2WRONG's key components below.

4.1 Mutation of Binary-affecting Dimensions

Recent work shows [57, 133] fuzzing is most effective when mutation is *maximized*—randomly distributing mutations across a target program's *entire* input space rather than only along a small portion. Yet, while binaries are distinctly influenced by source, compilers, optimizations, and executable formats, **current decompiler testing approaches limit exploration to only a *small* part of this vast input space**. DecFuzzer [54] focuses on source alone, eluding optimizations entirely; and similarly, prior approaches [20, 89, 142] consider only GCC and/or Clang, excluding other compilers—and consequently—other binary formats like PE and Mach-O.

To overcome these limitations, we propose a holistic approach to decompiler-oriented fuzzing, where source code and compilation parameters are treated and mutated as a *single, unified* testcase. Below, we describe BIN2WRONG's overarching **unified binary testcase construction** (Figure 2), and explain how its components work together to enable a deeper exploration of more diverse binary semantics—and more effective decompiler fuzzing—than previously possible.

4.1.1 Unifying Source and Compilation Mutation

With AFL++ [29] came new possibilities in implementing domain-specific mutators through its versatile *custom mutator API* (e.g., [86, 140]). Building on these capabilities, we develop BIN2WRONG's decompiler-oriented binary mutation via a suite of custom AFL++ mutators targeting both source- and compilation-level modifications, detailed below.

4.1.2 Mutating Compilation

As § 3.2 shows, different compilers, formats, and optimizations produce unique binaries even from identical source, chal-

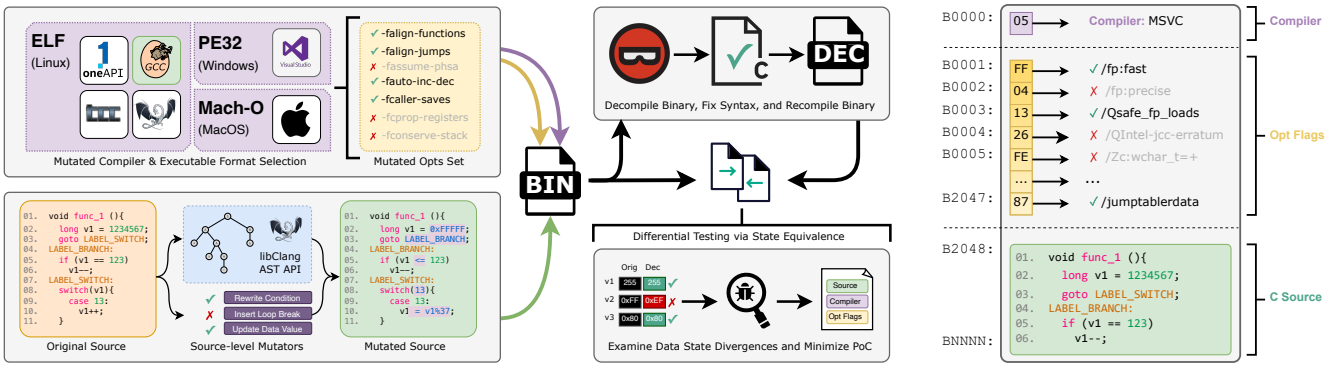


Figure 1: BIN2WRONG's high-level fuzzing workflow in a setup targeting Binary Ninja. Figure 2: Our unified testcase design.

lenging decompilers' recovery of higher-level code from varied binary structures (Table 1). Rather than restrictively *hard-code* compilation parameters like prior approaches (Table 2), BIN2WRONG reshapes compilation diversity as a fuzzing *mutation* technique: randomly selecting the compiler—and further—randomly toggling-on its available optimizations. Accordingly, BIN2WRONG's unified testcase construction (Figure 2) dedicates **the first 2048 bytes** to encode the fuzzer-mutated compilation parameters: byte `0` specifies the compiler, while bytes `1-2047` are each mapped to individual optimization flags specific to the chosen compiler. Upon selecting a testcase from its queue, BIN2WRONG modifies the compiler byte to one of its six supported compilers; while mutating optimization bytes to either *odd* or *even* numbers, denoting whether that optimization is enabled or disabled, respectively.

Compiler and Executable Format: Since BIN2WRONG is built atop the Linux-based fuzzer AFL++ v4.09c [29], it naturally supports Linux-native compilers such as GCC and Clang. In further maximizing compilation diversity across Linux's ELF binary format, we incorporate several mainstream Linux ELF compilers **unexplored by prior decompiler testing approaches**: Tiny C (TCC) [17], which aims to speeds-up compilation performance; as well as Intel's oneAPI DPC++ (ICX) [38], which uses advanced x86-specific optimizations.

Unfortunately, no Linux-based compilers support *non-*Linux formats MacOS's Mach-O and Windows' PE, leaving such approach unable to find their related bugs (e.g., Angr [96], Ghidra [71]). Yet, we see opportunities in cross-platform compatibility layers that run non-Linux code *on Linux* (e.g., WINE [2], Darling [26]). We leverage these tools to extend BIN2WRONG to non-ELF compilers—Apple's Xcode compiler (AppleClang) [4] for Mach-O, and Microsoft's Visual Studio Compiler (MSVC) [61] for PE—making BIN2WRONG **the first decompiler fuzzer spanning ELF *and* non-ELF formats *altogether***.

Optimization Selection: After selecting a compiler for the eventual binary, BIN2WRONG mutates its choice of the optimization options specific to that compiler, spanning a total of **1,029** total options for GCC v11.4.0, **1,378** for Clang v18.0.0, **7** for TCC v0.9.27, **1,174** for ICX v2024.0.2, **217** for MSVC v19.38.33134, and **1,378** for AppleClang v18.0.0. Should BIN2WRONG's compilation mutation later select a *different* compiler (e.g., MSVC→ ICX), BIN2WRONG remaps its 2,047 optimization bytes to the corresponding set for the *new* compiler. If incompatible optimizations result that prevent compilation, BIN2WRONG defaults to one of the compiler's standard optimizations (e.g., `-O1` to `-O3` for ICX), thus resolving 100% of optimization-related compile issues.

While BIN2WRONG's high-level optimization mutation is inspired by Cornucopia's [89], Cornucopia is built to explore only a single compiler's optimization suite at a time, and therefore cannot generalize across different compilers and formats in any one campaign. Comparatively, BIN2WRONG's ability to explore *multiple* compilers' optimizations simultaneously enables it to encompass **5,183** optimizations in total versus Cornucopia's 892 explored Clang-based optimizations—resulting in a far greater binary diversity from **BIN2WRONG's inclusion of both ELF and non-ELF compilers *and* their optimizations**.

Our Approach: Framing compilation diversity as a fuzzing mutation allows for **dynamic exploration of varied, compilation-dependent binary semantics**—enabling broader exploration than prior decompiler testing approaches that rely on hardcoded compilation parameters.

4.1.3 Mutating Source Code

Following compilation mutation, BIN2WRONG explores varied code semantics through its source-level mutators. We model BIN2WRONG's testcases after those in prior compiler [139] and decompiler fuzzers [54], each consisting of a single function targeted for source mutation. To minimize risk of syntax-breaking mutations, we use the LLVM [52] framework's libClang abstract syntax tree (AST) API [56] alongside our own suite of AFL++ custom mutators.

Informed by our survey of real-world semantic bugs (Table 1), we implement the following *expression*-level mutators to probe decompilers' recovery of complex data-affecting instruction sequences (e.g., Ghidra [75], Binary Ninja [121]):

- **Expressions:** Deleting an entire expression; or duplicating it immediately after the original one:
  e.g., `v0 = v1++;` → `v0 = v1++; v0 = v1++;`
- **Sub-expressions:** Deleting or duplicating sub-expressions; or expanding with an existing one:
  e.g., `v1/v4;` → `v1/v4 + (v1%v2);`
- **Operators:** Randomly replacing arithmetic/bitwise operators; or flipping increments/decrements:
  e.g., `(v1*v2)-(v3++);` → `(v1/v2)*(v3-);`

To test decompilers' recovery of complex control-flow logic (e.g., RetDec [9], Angr [105]), we additionally implement the following *control structure* mutators:

- **Loops:** Inserting a conditional `break` or `continue` with a random threshold at a random location:
  e.g., `for (i=0){}` → `for (i=0){ if(i>37) break; }`
- **Conditional Branches:** Replacing the condition with a always-`true` or always-`false` condition:
  e.g., `if(v2/v3 == 0){}` → `if(1){}`
- **Unconditional Branches:** Replacing a `goto` destination with a randomly-chosen existing label:
  e.g., `goto LABEL_BRANCH;` → `goto LABEL_SWITCH;`
- **Switch-Cases:** Replacing the `switch`'s expression with a randomly-selected existing case value:
  e.g., `switch(v1){ case 2: ... }` → `switch(2){...}`

Lastly, to find bugs stemming from imprecise data object, type, and value recovery (e.g., Binary Ninja [119], Ghidra [74]), we implement the following *data-level* mutators:

- **String Literals:** Replacing with a randomly-chosen string literal; or flipping one or more bytes:
  e.g., `char *v0="Global";` → `char *v0="0xFFFFal";`
- **Numerical Literals:** Replacing with a random number; or flipping its sign or one or more bytes:
  e.g., `(v1/55)+(0xFF*v2);` → `(v1/-55)+(0x00*v2)`
- **Numerical Typecasts:** Replacing with a random numerical type (e.g., `int`, `short`, `float`, `long`):
  e.g., `(short) v3/25);` → `(float) v3/25);`

Preventing Semantically-invalid Mutations: To avoid the risk of semantically-invalid mutations that lead to runtime crashes when executing generated binaries, BIN2WRONG's source mutators are designed to operate **conservatively**—deliberately avoiding injecting, deleting, or moving variable *declarations*. Without such safeguards, BIN2WRONG rapidly encounters semantics-related errors from missing or erroneously-placed declarations. We further build our mutators around AST-node-specific guardrails, such as placing `continue` or `break` statements only within loop bodies. While we see opportunities for more guardrails—for example, deliberately injecting source code that will not be optimized-out by the corresponding optimization flags selection—we posit that the runtime costs of additional checks likely outweighs their benefits, and that mutation quality ultimately does not hinder BIN2WRONG's overall performance, as shown by its strong results in our evaluation (§ 5).

Supported Constructs and Flexibility: While BIN2WRONG supports common source constructs—expressions, control logic and structures, and data objects, types, and values—we see room for incorporating additional ones (e.g., `union`, `struct`, and arrays). Whereas other fuzzers inherit limitations from their bespoke, non-exhaustive AST manipulation tooling (§ 3.1), an advantage of BIN2WRONG's design is that its source mutators operate **directly on Clang's robust AST** interface [56]. Adding new mutators targeting other AST node types requires minimal engineering, as Clang's AST already provides access to these and all other AST nodes.

Our Approach: By bridging AST-level source mutations with compilation mutations, general-purpose fuzzing can **explore the expansive range of diverse source and compilation *combinations***—achieving unprecedented binary diversity for *systematic* decompiler testing.

4.2 Decompiler-agnostic Semantic Testing

Following testcase mutation, BIN2WRONG compiles the binary according to the specified compilation parameters and forwards it to the targeted decompiler. Drawing inspiration from previous works [54, 139], BIN2WRONG employs *differential testing* to detect cases where decompiler-emitted code semantics *diverge* from the original program's. Below, we outline the core techniques enabling BIN2WRONG's discovery of semantics-affecting decompiler bugs.

4.2.1 Recompilation-oriented Syntax Patching

Since small syntactical errors prevent entire programs from being recompilable, we perform minor syntax corrections on decompiled code. Our high-level recompilation-oriented syntax patching spans the following basic strategies:

- Resolving compilation-blocking variable type mismatches:
  e.g., `void* const str;` → `char *str;`
- Renaming objects that differ from their intended names;
  e.g., `void __cdecl func_1()` → `void func_1()`
- Injecting macros to resolve decompilers' custom types:
  e.g., `word32 v0;` → `typedef int32_t word32;`
- Culling, replacing, or adding externs for decompiler-introduced macros, symbols, and functions:
  e.g., `movss(a);` → `extern float movss(float a);`

While some recompilation failures are not easily solved via basic syntax patching, we see a major improvement in post-patching recompilation success, rising from 11.60% to **47.8%** on average—comparable to D-Helix's reported 45% recompilation rate for Angr [142]. Importantly, such failures do not prevent BIN2WRONG's finding of *the most* bugs in our evaluation (Table 6); and moreover, no developers reported any false-positives caused by our syntax patching. We weigh additional strategies for improving recompilation success in § 6.2.

4.2.2 Bug-finding via Differential Execution

To uncover subtle semantic bugs (e.g., Table 1), we adopt Csmith [139]'s approach of *variable*-level differential execution checking. Like other fuzzers', BIN2WRONG's generated programs serve as the *decompilers'* inputs and are designed to be fully self-contained, requiring no external inputs. Each program deterministically exercises its logic based solely on internal control and data flows, ensuring reliable and repeatable semantic comparison across original and decompiled forms. Accordingly, we modify BIN2WRONG-generated testcases' source functions to initialize a suite of "placeholder" globals, which eventually assume their values from the mutated testcase's *local* variables. Post-decompilation, we instrument all globals in the original and recompiled programs to self-report value-dependent checksums. After executing both, if the recompiled program's checksum *differs* from the original's, we flag it as a *semantics-affecting* defect in the decompiled code.

By focusing solely on decompiler-emitted C code, BIN2WRONG extends support to *any* C-targeting decompiler. In contrast, D-Helix's [142] symbolic-execution-based approach needs ***per-decompiler*, *per-instruction* semantic modeling** for decompilers' internal Intermediate Representations (IRs), requiring considerable retooling to support more semantics, as evidenced by their lack of *floating-point* instructions (Table 2), and moreover, its estimated **40 days' time** to support *new decompilers*. BIN2WRONG avoids invasive bug-detection mechanisms, extending semantic correctness testing to *all* C decompilers—**both open- *and* closed-source**—whilst being the first to achieve systematic, unified mutation across all major dimensions of binary generation.

Our Approach: Utilizing the original program as *the correctness oracle* allows for decompiler-*agnostic* testing, eliminating any need for impractical tailoring to decompilers' dissimilar internal IRs—whilst enabling **practical testing of open- *and* closed-source decompilers**.

4.2.3 Post-fuzzing Manual Bug Analysis

Like other fuzzers [29], BIN2WRONG offloads bug post-processing to manual analysis. We spent about 10 minutes manually minimizing each bug-exposing testcase, in-line with the duration reported by DecFuzzer's authors [54], before deduplicating them into distinct *unique* bugs. While conventional fuzzers benefit from a rich ecosystem of automated fault localization tools (e.g., [87]), no analogous tools exist for binary decompilers; hence, we relegate fault localization to the best-equipped domain experts—decompilers' own developers—and promptly report all bugs we find.

5 Evaluation

Our evaluation of BIN2WRONG is guided by the following fundamental questions:

- **Q1. Testing Depth:** Is BIN2WRONG more effective at exercising decompilers' internals?
- **Q2. Bug Discovery:** Does BIN2WRONG reveal more decompiler semantic correctness bugs?
- **Q3. New Results:** Where do decompilers struggle in handling source and compilation diversity?

5.1 Experimental Setup

We evaluate BIN2WRONG alongside state-of-the-art decompiler fuzzers Cornucopia and DecFuzzer across seven highly popular decompilers: the free and open-source Angr [132], R2Ghidra (Radare2+Ghidra) [1], Reko [49], Relyze [85], RetDec [6], and Rev.Ng [24]; as well as the commercially-sold decompiler Binary Ninja [108]. We omit D-Helix [142] as it unfortunately only supports two of our seven decompilers; as well as DSmith [20] as its source diversity is far more constrained than DecFuzzer's, excluding basic increment/decrement operators, jumps, and many common numerical data types [19]. As BIN2WRONG is built atop AFL++ v4.09c [29], we configure it in grey-box mode for the five decompilers supportive of its QEMU-based coverage tracing [16]—Binary Ninja, R2Ghidra, Reko, RetDec, and Rev.Ng—and coverage-agnostic black-box mode for Angr and Relyze.

We configure DecFuzzer's and Cornucopia's compilation parameters according to their respective papers: DecFuzzer with GCC and `-O0`, and Cornucopia with Clang and its full optimization suite. Per DecFuzzer's procedure [54], we seed all fuzzers with a corpus of Csmith-generated C programs (10 in total). Because Csmith omits certain constructs—most notably string literals—we manually augment its generated seed programs to include these elements. Following [43], we repeat all experiments for five trials; and compute Mann-Whitney U's statistical test at p=0.05 significance in comparing BIN2WRONG to Cornucopia and DecFuzzer (with statistically-significant p-values shown **bolded**). We perform all experiments on Ubuntu 22.04 machines, each with an Intel Core i9-12900K CPU and 64GB RAM.

5.2 Q1: Decompiler Testing Depth

In the following sections, we weigh BIN2WRONG's effectiveness in enabling in-depth decompiler testing through its impact on binary diversity and decompiler code coverage.

5.2.1 Binary Diversity

Rigorously testing decompilers demands binaries that *maximize* internal diversity. To assess BIN2WRONG's diversity, we measure binary similarity scores against Cornucopia and DecFuzzer using three industry-standard diffing algorithms:

DiffTool	BIN2WRONG Binary Diffing Scores per Dimension(s) of Source, Compiler, Optimization							BIN2WRONG-ALL Relative Mean Diffing Score Δ			
	ALL THREE	SRC+COM	SRC+OPT	COM+OPT	ONLY SRC	ONLY COM	ONLY OPT	over Cornucopia	MWU p	over DecFuzzer	MWU p
BinDiff	0.627	0.544	0.332	0.623	0.321	0.539	0.326	10.398 ×	**0.009**	17.189 ×	**0.011**
Radiff2-M	0.490	0.368	0.096	0.489	0.086	0.366	0.094	9.119 ×	**0.010**	16.941 ×	**0.007**
Radiff2-L	0.572	0.437	0.120	0.571	0.107	0.436	0.116	7.131 ×	**0.009**	16.089 ×	**0.011**

Table 3: Comparison between BIN2WRONG's mean binary diffing scores across all mutation dimension(s) for each diffing algorithm; and BIN2WRONG's overall scores relative to state-of-the-art decompiler fuzzers Cornucopia and DecFuzzer.

(1) Zynamics' BinDiff [143]; **(2)** Radiff2's Eugene W. Myers' $O(ND)$ Diffing Algorithm [62] (**Radiff2-M**); and **(3)** Radiff2's Levenshtein's Edit Distance [53] (**Radiff2-L**). We configure each to generate 1,000 testcases for each of their 10 Csmith-generated seed programs, totaling 10,000 testcases per competitor. We compute all three diff scores per generated binary relative to its seed binary, and report mean score across all comparisons. Table 3 shows the five-trial mean diversity per BIN2WRONG's supported mutation configurations—source, compiler, optimization, and all combinations thereof—alongside those of Cornucopia and DecFuzzer.

Results: Overall, BIN2WRONG-generated programs achieve **7.13–10.39**× and **16.08–17.18**× binary diversity relative to Cornucopia and DecFuzzer, respectively. In our ablation study of BIN2WRONG's full mutation modes, we see that mutating across *all* dimensions—source, compiler, *and* optimizations (column "ALL THREE" in Table 3)—yields the highest binary diversity across all three diffing algorithms, underscoring that **systematic, unified mutation *maximizes* binary diversity**.

5.2.2 Decompiler Code Coverage

As prior work shows [27], higher code coverage increases the likelihood of finding semantics-affecting defects. To determine whether BIN2WRONG's high-diversity inputs yield greater decompiler code coverage, we measure the coverage of BIN2WRONG, Cornucopia, and DecFuzzer on all coverage-tracing-compatible decompilers: Binary Ninja, R2Ghidra, Reko, RetDec, and Rev.Ng. For tracing coverage, we utilize the AFL-QEMU-Cov [28] tool, as it supports both open- and closed-source decompilers; however, because AFL-QEMU-Cov does not support Reko, we instead use AFL++'s built-in AFL-Showmap [29] utility to trace Reko's code coverage. We report the mean edge coverage results across five 24-hour trials per competitor, as shown in Table 4.

Decompiler	BIN2WRONG's Relative Increase over Cornucopia				over DecFuzzer			
	Δ COV	MWUp	Δ BIN	MWUp	Δ COV	MWUp	Δ BIN	MWUp
Bin.Ninja	1.091 ×	**0.008**	8.189 ×	**0.008**	1.215 ×	**0.008**	123.694 ×	**0.011**
Reko	0.989 ×	**0.008**	4.035 ×	**0.008**	0.973 ×	**0.008**	18.335 ×	**0.012**
R2Ghidra	1.344 ×	**0.012**	1.294 ×	0.421	1.451 ×	**0.008**	42.522 ×	**0.011**
RetDec	1.341 ×	**0.008**	0.798 ×	0.222	1.704 ×	**0.008**	26.201 ×	**0.007**
Rev.Ng	1.073 ×	**0.008**	16.836 ×	**0.008**	1.395 ×	**0.008**	297.763 ×	**0.011**
GeoMean:	**1.16** ×		**3.56** ×		**1.32** ×		**59.61** ×	

Table 4: BIN2WRONG's mean coverage (COV) and ratio of coverage-increasing binaries (BIN) relative to competitors.

Results: BIN2WRONG achieves mean **1.16**× and **1.32**× code coverage relative to Cornucopia and DecFuzzer, respectively, with all improvements showing statistically-significant p-values (< 0.05). Although BIN2WRONG shows statistically-significant lower coverage on Reko, it demonstrates substantially higher binary *quality*—the ratio of coverage-*increasing* binaries to total binaries generated—with a mean **3.56**× and **59.61**× relative to Cornucopia's and DecFuzzer's, respectively. This indicates that **BIN2WRONG's binaries exercise considerably more decompiler internals** compared to those generated by prior, *un*-systematic decompiler fuzzers.

Decompiler	Coverage-increasing Binaries by Mutation Configuration						
	ALL THREE	SRC +COM	SRC +OPT	COM +OPT	ONLY SRC	ONLY COM	ONLY OPT
Bin.Ninja	73.93%	0.04%	22.31%	3.26%	0.00%	0.00%	0.45%
Reko	82.47%	0.00%	16.64%	0.88%	0.00%	0.00%	0.00%
R2Ghidra	79.32%	0.02%	20.59%	0.07%	0.00%	0.00%	0.00%
RetDec	77.44%	0.00%	22.52%	0.03%	0.00%	0.00%	0.01%
Rev.Ng	68.56%	0.08%	30.96%	0.39%	0.00%	0.00%	0.00%
Mean:	**76.34%**	**0.03%**	**22.61%**	**0.93%**	**0.00%**	**0.00%**	**0.09%**

Table 5: BIN2WRONG's mean distributions of coverage-increasing binaries across all mutation dimension(s).

Furthermore, in an ablation study measuring the coverage impacts of different mutation configurations (Table 5), we observe that mutating source, compiler, and optimization *altogether* consistently produces the most coverage-enhancing binaries. Consequently, **by maximizing binary diversity, BIN2WRONG achieves the highest-overall coverage** across both open- and closed-source decompilers.

5.3 Q2: Decompiler Bug Discovery

We assess BIN2WRONG's effectiveness in identifying *semantic decompilation errors* across seven free and commercial decompilers: Angr, Binary Ninja, R2Ghidra, Reko, Relyze, RetDec, and Rev.Ng. We perform a subsequent manual root cause analysis to deduplicate all fuzzers' uncovered runtime divergences into their *unique* semantics-affecting decompiler defects. We report all identified decompiler correctness errors (Table 6) to their respective developers and maintainers.

Results: Overall, BIN2WRONG uncovers a total of **48** semantic decompilation bugs, **42** of which are uniquely found by BIN2WRONG alone, with **30** confirmed by their respective developers. Comparatively, mutation-restricted approaches Cornucopia and DecFuzzer reveal far fewer bugs—ten and zero, respectively—demonstrating the advantages of **BIN2WRONG's high-diversity, high-coverage systematic binary mutation** in thorough decompiler defect discovery. While § 5.2 suggests many of BIN2WRONG's binaries cover much of the same underlying decompiler logic—a common

Decompiler	Total Bugs Found, Confirmed, and Unique per Approach								
	BIN2WRONG			Cornucopia			DecFuzzer		
	BUGS	CONF	UNIQ	BUGS	CONF	UNIQ	BUGS	CONF	UNIQ
Angr	9	9	8	2	2	1	0	0	0
BinaryNinja	11	11	11	0	0	0	0	0	0
Reko	6	6	4	3	2	1	0	0	0
R2Ghidra	2	2	1	1	1	0	0	0	0
Relyze	7	N/A	6	2	N/A	1	0	0	0
RetDec	11	N/A	10	2	N/A	1	0	0	0
Rev.Ng	2	2	2	0	0	0	0	0	0
Total:	**48**	**30**	**42**	**10**	**5**	**4**	**0**	**0**	**0**

Table 6: Total decompiler bugs found (BUGS), total confirmed (CONF), and total unique (UNIQ) per each testing approach. N/A = instances where our correspondence went unanswered.

phenomenon in fuzzing as a whole—BIN2WRONG's broader strength is in its surfacing of subtle semantic bugs triggered by diverse source, compiler, optimization, and format *combinations*. Thus, we reason that BIN2WRONG's higher bug discovery is more directly related to its higher binary diversity (Table 3), rather than its higher code coverage (Table 4).

5.4 Q3: Analysis of Uncovered Bugs

While prior works categorize decompiler bugs into broad high-level classes (e.g., "Type Recovery" [54]), none delve into the specific *low-level causes*—or their prevalence—behind these critical failures. In the following, we conduct a comprehensive analysis of the semantics-affecting decompiler defects uncovered by BIN2WRONG, identifying their distinct source-related and compilation-related origins, supported by case studies on unique bugs discovered by BIN2WRONG.

5.4.1 Causes of Source-related Defects

Among the 48 semantics-affecting bugs discovered by BIN2WRONG, we identify **data recovery** as the most frequent error source (**68.75%**), followed by **expression** recovery (**20.83%**) and **control** structure recovery (**10.42%**).

Data Object, Type, and Value Recovery: Of the 33 data recovery bugs, we identify two involving incorrectly-recovered data *objects*: the first stems from Binary Ninja erroneously interpreting the `truncf()` function as taking multiple arguments instead of one [58]; while the second arises from Reko misordering function arguments, despite recovering them correctly otherwise. Table 7 presents a breakdown of the remaining 31 data-related bugs, categorized by *type* and *value* errors across integer, float, and string variables and literals. Although decompiler performance varies across data constructs, BIN2WRONG reveals several type recovery issues specifically related to **floating-point** data, which we discuss below.

Case Study: Floating-point Recovery. Figure 3 shows a BIN2WRONG-found error in floating-point recovery affecting Binary Ninja, Angr, and Reko. In binaries, floating-point values follow the IEEE-754 format, with instructions like `movss` for single-precision and `movsd` for double-precision, distinct from those used for integers. Although decompilers

	Integers				Floats				Strings			
	Value		Type		Value		Type		Value		Type	
	VAR	LIT	VAR	LIT	VAR	LIT	VAR	LIT	VAR	LIT	VAR	LIT
Angr	1	3	2	1	0	0	0	1	0	0	0	0
Bin.Ninja	0	0	1	2	0	0	0	1	0	1	0	0
Reko	0	0	0	0	0	1	0	0	0	2	0	0
R2Ghidra	0	1	0	0	0	0	0	0	0	0	0	0
Relyze	1	1	1	1	0	0	0	0	0	2	0	0
RetDec	1	4	2	0	0	0	0	1	0	0	0	0
Rev.Ng	0	0	0	0	0	0	0	0	0	0	0	0

Table 7: Summary of BIN2WRONG-uncovered data *type* and *value* bugs spanning variables (VAR) and literals (LIT).

accurately recognize floating-point *instructions* in disassembly (e.g., line#2 in Figure 3), they often fail to convert the hexadecimal representation of floating-point values *back* to their original form; instead, they output the *hex* form directly, incorrectly interpreting these values as integers rather than recovering them as floating-point numbers. These issues highlight the critical need to consider **diverse data constructs**, underscoring limitations of approaches like DecFuzzer, DSmith, and D-Helix which fail to support floating-point data (§ 3.1).

```
1  movsd xmm0,[rbp];
2  double v0 = 1.0;
```
(a) Original

```
1  int64_t v1 =
2  0x3FF0000000000000;
```
(b) Decompiled

Figure 3: Erroneous type recovery of float-point data as integers, affecting Angr, Binary Ninja, and Reko.

Expression and Control Recovery: We analyze the eight expression-related semantic errors identified by BIN2WRONG and categorize them into two groups. Four involve misrecoveries of **individual arithmetic operators**: incorrect handling of the modulo operator and interpreting signed comparisons as unsigned (Binary Ninja); unnecessary multiplication symbols (Reko); and reversing arithmetic operators to their opposites (RetDec). Our remaining four errors arise from mishandled parsing and refinement of **data-affecting instruction sequences**, including: inaccurate propagation of modulo results (Angr); erroneous omission of instructions preceding inlined function code (Binary Ninja); extraneous operations (Relyze); missed constant-affecting operations (RetDec); and functions that fail to parse correctly (both Reko and RetDec).

Of the five bugs arising from control structure recovery, only one involves loops (R2Ghidra), while the remaining four are related to `switch-case` structures (2×Binary Ninja, 2×Rev.Ng). Figure 4 presents a BIN2WRONG-uncovered bug exposing `switch-case` recovery challenges uniquely faced by commercial decompiler Binary Ninja.

Case Study: Switch-Case Logic Recovery. In programs with `switch-case`s, Binary Ninja's medium-level intermediate representation (IR) accurately translates `case` statements into semantically-equivalent nested `if-else` statements, preserving the intended logic. However, during translation to higher-level IR, Binary Ninja incorrectly *breaks* these into separate `if` statements. Although the conditions—the original `case` values—remain correct, this restructuring makes

```
1  int var = 0;
2  switch(var){
3    case 0:    Reached?
4      var = 5;  ✔
5      break;
6    case 1:
7      ...
8    case 2:
9      ...
10   default:
11     idx = 0;  ✗
12     break;
13 }
```
(a) Original

```
1  int var = 0;
2  if (var == 2)
3    ...
4  }
5  else{
6    if (var == 0) Reached?
7      var = 5;      ✔
8    if (var == 1)
9      ...
10   if (var > 2){
11     idx = 0;      ✔
12   }
13 }
```
(b) Decompiled

Figure 4: Incorrect switch-case recovery, making the original "case 0" and "default" blocks erroneously reachable.

multiple code blocks *reachable*, leading to divergent execution and incorrect program outputs. Binary Ninja's developers classified this bug as **High Severity**, indicating the need for a substantial overhaul of their core control-flow restructuring—underscoring the critical role of **diverse control constructs** in BIN2WRONG's systematic decompiler correctness testing.

5.4.2 Causes of Compilation-related Defects

We further examine the influence of different compilers, executable formats, and optimization flags on decompilation defects. Table 8 details the compilation parameters for all of BIN2WRONG's 48 uncovered decompiler bugs.

	Binary Formats & Associated Compiler(s)						# Opts		
	ELF				PE	Mach-O			
	CLANG	GCC	TCC	ICX	MSVC	APPLE	0	1	2+
Angr	9	7	7	9	4	9	9	0	0
Bin.Ninja	8	7	4	8	4	8	8	1	2
Reko	2	2	2	2	2	2	2	0	0
R2Ghidra	4	4	5	4	3	4	6	0	0
Relyze	7	5	5	6	3	0	7	0	0
RetDec	9	7	6	7	5	8	10	0	1
Rev.Ng	2	2	2	2	2	0	2	0	0

Table 8: High-level summary of the compilation parameters for all BIN2WRONG-discovered semantic bugs.

Optimization-dependent Bugs. Although most bugs are not directly caused by compiler optimizations, we observe that various optimizations are key to exposing intricate edge cases. We find three such bugs (2×Binary Ninja, 1×RetDec) from combinations of **two or more** optimizations, where decompilers produced only partial code and missed significant logic. Additionally, one bug, triggered only by the `-O1` optimization level, led to errors in loop logic recovery (in Binary Ninja). We detail the latter example below, shown in Figure 5.

```
1  while (v0!=11) { v0++ };
```
(a) Unoptimized code.

```
1  while (v0!=0xFFF5) { v0-- };
```
(b) Optimized & decompiled.

Figure 5: Binary Ninja bug stemming from the `-O1` optimization negating the loop constraint, causing divergent `v0` values in the original and decompiled programs post-loop-execution.

Case Study: Optimized Loop Recovery. Shown in Figure 5 is a `while` loop with counter incremented on each iteration, breaking only on count `11`. When compiled with `-O0`, the binary closely resembles this original structure; however, with `-O1`, the compiler replaces `11` with `-11` and the increment operation with a *decrement*. As negative numbers are stored in two's complement, Binary Ninja incorrectly recovers only this form, deviating from the original loop's semantics. BIN2WRONG caught this via the different `v0` values seen in the original and decompiled programs *post*-loop-execution (i.e., `11` with `-11`, respectively). This issue reiterates the importance of considering **diverse optimizations**—testing decompilers' ability to handle *each* effectively.

Compiler- and Format-dependent Bugs. We identify **eight** bugs uniquely stemming from a *single* executable format (ELF). However, these bugs are triggered by different compilers: four occur only in GCC-compiled binaries, one appears solely in TCC-compiled binaries, and the remaining span two or more compilers. Interestingly, we see the vast majority of bugs span *multiple*—but not all—executable formats and compilers: 14 appear in compilers *other* than GCC, 7 in ones other than Clang, 17 in ones other than TCC, 24 in ones other than MSVC, 10 in ones other than ICX, and 10 in ones other than AppleClang. Figure 6 shows an example bug that eludes PE binaries yet appears in both ELF *and* Mach-O formats.

```
1  mov r8,  z
2  mov rdx, y
3  mov rcx, x
4  func(x, y, z);
```
(a) PE convention & code.

```
1  mov rdi, x
2  mov rsi, y
3  mov rdx, z
4  func(z, y, x);
```
(b) Mach-O & ELF decompiled.

Figure 6: Call convention bug in Reko on Mach-O & ELF binaries (right) versus the analogous PE form (left).

Case Study: Calling Convention Recovery. In PE binaries, Microsoft's x64 conventions enforces that the first, second, and third integer arguments of a function are passed via registers `rcx`, `rdx`, and `r8`, respectively. Yet for ELF and Mach-O, AMD64's System V ABI instead passes these via `rdi`, `rsi`, and `rdx`, respectively. Shown in Figure 6, Reko's call recovery succeeds on PE binaries, yet *fails* on *non-PE* binaries' conventions, resulting in incorrect argument ordering. Ensuring decompilers' accuracy with these subtle differences reinforces the need to explore **both *ELF* and *non-ELF*** formats.

```
1  0x100 → 0x200
2  0x200 → 0x4142
3  mov rsi, [0x100]
4  char *s ="AB";
```
(a) Clang convention (one-level indirection) and original code.

```
1  0x100 → 0x150
2  0x150 → 0x200
3  0x200 → 0x4142
4  mov rax, [0x100]
5  mov rsi, [rax]
6  int64_t s =0x200;
```
(b) TCC convention (two-level indirection) and decompiled code.

Figure 7: TCC-specific string literal recovery bug in Reko.

Case Study: Compiler-specific String Recovery. Figure 7 depicts a compiler- *and* source-related bug in Reko. While most compilers store the string's direct memory address, allowing retrieval with a single dereference, TCC introduces an

extra indirection—a pointer to an address holding the actual location of the string, requiring *two* dereferences to access it. This approach presents challenges for decompilers, as seen in Reko, which retrieves only the *first* level of the memory address, leaving the second dereference unresolved. As a result, Reko fails to recognize the additional indirection, obscuring the true content of the string literal—and the program's intended semantics. Such defects demonstrate how binary semantics are uniquely influenced by **variations in both source *and* compilation**, highlighting BIN2WRONG's strengths over prior approaches that largely overlook such diversity.

6 Discussion

Below we discuss potential limitations of this work and our prototype decompiler fuzzer, BIN2WRONG.

6.1 Responses from Decompiler Developers

Developers of Angr, Binary Ninja, R2Ghidra, Reko, and Rev.Ng promptly **confirmed all of BIN2WRONG's uncovered issues as valid** decompilation bugs. Unfortunately, our attempts to contact developers of RetDec and Relyze went unanswered. Fortunately, most developers responded and quickly resolved many of our reported bugs, underscoring the importance of these fixes on the reliability of their tools.

We see that Binary Ninja is the only decompiler whose developers disclose severity scores of publicly-reported bugs, classifying eight of BIN2WRONG's found defects as follows: three *low* severity, four as *medium*, and one as *high* severity. These errors spanned *all three* of Binary Ninja's internal stages: one in its medium-level intermediate language (IL) generation, two in high-level IL, and five in final C code generation. BIN2WRONG's discovered high-severity bug (Figure 4) drew significant attention in particular, **prompting a total overhaul** of Binary Ninja's core control-flow structuring techniques. This bug was later featured in a blog post detailing the efforts taken to improve the soundness of Binary Ninja's decompilation—**further highlighting BIN2WRONG's real-world impacts** in uncovering critical semantic errors in even mainstream, commercially-sold decompilers.[1]

6.2 Reliance on Recompilation

BIN2WRONG's differential testing (§ 4.2.2) relies on *recompiling* decompiled code, with minor patching applied to fix common decompiler syntax errors. As BIN2WRONG's post-patch recompilation rate averages 47.8%—in-line with D-Helix's own 45% post-patching recompilation rate for Angr [142]—many syntax errors remain unresolvable by our current patches. We observe **84.03%** result from undeclared identifiers, **7.51%** from incompatible type conversions (e.g., pointers → integers), **6.24%** from extraneous arguments, **1.49%** from extraneous symbols (e.g., code comments), and **0.52%** from patch conflicts (e.g., BIN2WRONG adding `extern` within expressions). While more advanced program repair will improve recompilation, we posit its overhead will greatly reduces BIN2WRONG's total throughput. Thus, as research suggests *faster* fuzzing is better [63], we prioritize speed over 100% recompilation. Interestingly, the remaining **0.21%** of issues stem from *compiler* errors, typically with Clang or ICX crashing under certain optimization sets. Given their infrequency, we consider identifying and culling these specific conflicts outside our current scope.

6.3 Other Potential Binary Dimensions

Binary obfuscation—such as control-flow flattening, opaque predicates, and virtual-machine-based obfuscation—are often employed to hinder decompilers, typically to protect intellectual property or complicate reverse engineering [23, 107]. BIN2WRONG currently deliberately excludes such techniques, as contemporary decompilers tend to delegate deobfuscation tasks to specialized tools and plugins [35, 109]. However, BIN2WRONG easily supports incorporating obfuscation with minor adjustments, though we leave the exploration and requisite reevaluation of this to future work.

While we also considered linkers in BIN2WRONG's mutation space, our investigation found no prior decompiler bugs where the linker was the true root cause of semantic recovery errors. All linker-related issues we found were limited to syntactic or format-specific parsing failures—e.g., malformed headers or symbol resolution issues that simply prevented the binary from being fully analyzed at all (e.g., [64, 67, 83]). We believe this is unsurprising, as linkers must conform to OS-level specifications (e.g., System V ABI), and thus typically do not touch control/data semantics. As such, we scoped linking out of BIN2WRONG's design, though we anticipate this could be explored in future work focusing on malformed binaries or toolchain robustness.

6.4 Search Space and Bug Discovery Cost

While BIN2WRONG's combined source/compiler/format/optimization search space currently spans 6 compilers, 3 formats, 5,183 optimizations, and dozens of source constructs—easily amounting to trillions of unique binary generation configurations—BIN2WRONG's goal is not to exhaustively search this space, but to sample it broadly and systematically. Inspired by mutation-based fuzzers like AFL++ [29], we find that even simple, random mutations across multiple dimensions are highly effective at uncovering semantic bugs missed by prior tools limited to one or two axes (Table 2). BIN2WRONG's novelty thus lies in unifying diverse mutation dimensions into a single framework—showing that breadth alone, when systematically applied, surfaces impactful bugs that dimension-isolated fuzzers cannot reach. While we envision opportunities to more directly target distinct elements

[1] https://binary.ninja/2024/06/19/restructuring-the-decompiler.html

of this search space that hinder decompilation (e.g., specific source and optimization combinations guaranteed to produce interesting machine code), we leave this to future work.

7 Related Work

Below we discuss recent related research in the areas of binary decompilation, as well as the use of program generation and mutation in other applications of automated testing.

7.1 Recent Advancements in Decompilation

Several works are leveraging advances in machine learning to enhance the recovery of variable *type* information: Coda [30] employs neural networks, while Slade [5] and ReSym [135] utilize large language models. Machine learning is also seeing use in improving recovery of variable *names*, further boosting the human readability of decompiled code. DIRE [51], DIRTY [21], and TYGR [141] all utilize neural networks, whereas VarBERT [79] incorporates additional pre-training on human-written source code to produce more intuitive variable names. Many efforts are also enhancing restructuring and optimization of decompiler-generated *code*—particularly for `gotos`: Phoenix [18], Rev.Ng [24,33], and DREAM [137] aggressively cull `gotos`, while SAILR [15] instead introduces compiler-specific heuristics for eliminating only *spurious* ones. Though these recent advancements largely exist as separate from mainstream decompiler platforms, we expect that future efforts will bridge the gap between academically-sourced decompilation enhancements and industrial decompilers. We foresee BIN2WRONG playing a key role in enabling automated semantic testing of decompilers—and accelerating discovery of their bugs—as these platforms continue to evolve with increasing sophistication each year.

7.2 Program Generation and Mutation

A cornucopia of program generation and mutation techniques have emerged from the field of compiler fuzzing: Csmith [139], Rustsmith [88], and YARPGen [55] randomly generate grammar-conforming programs, while GrayC [27] mutates existing ones via Clang's AST API. More recently, Whitefox [138], GoFuzz [32], and Fuzz4All [134] instead use LLMs to explore more diverse source code constructs. As BIN2WRONG builds upon the program generation and mutation popularized by compiler fuzzing, we expect future synergistic approaches to extend BIN2WRONG's to other compiled languages. For instance, incorporating other language-specific AST APIs will help BIN2WRONG test emergent decompilers targeting non-C code, such as Go [41] and Rust [59].

7.3 Grey-box Fuzzers

BIN2WRONG's overall design inherits many design advantages from its core grey-box fuzzer, AFL++ [29]—which itself is one of many mainstream general-purpose fuzzers seeing large-scale adoption today [31]. Yet, standard grey-box fuzzers are not built to find semantic bugs such as decompilation recovery errors, and instead only look for crashing signals (e.g, `SIGSEGV`). While semantic bug-finding is not a new concept [55, 139], to our knowledge, BIN2WRONG is the first system to extend AFL++ with this capability in fuzzing decompilers. We anticipate these components of BIN2WRONG could likely be repurposed for similar semantics-related bug domains (e.g., compiler fuzzing, transpiler fuzzing).

8 Conclusion

Current decompiler testing approaches broadly overlook how source code, compilers, optimizations, executable formats, *and combinations thereof* collectively shape binary executable code. In this work, we show that giving equal consideration to these factors yields significantly more diverse binaries, enabling far more effective testing of decompilers' correctness. Beyond attaining the highest-overall code coverage across seven free and industrial decompilers, our prototype BIN2WRONG reveals the most semantic errors of all decompiler testing approaches—the vast majority of which remain undetectable by prior state-of-the-art methods. At the time of writing, **30 out of 48 decompiler errors found by BIN2WRONG are confirmed by developers**, with several prompting major changes in real-world decompilers.

By embracing principles that make conventional mutation-based application fuzzing successful—namely, maximizing testcase diversity through pursuing as broad of mutation as possible—we show that complex decompiler semantic errors are easily discovered in as little as 24 hours. We thus envision an era where holistic, automated techniques help improve the accuracy of today's many critical binary analysis tools—facilitating faster and more reliable completion of the downstream tasks that rely on these tools' precision.

Acknowledgments

We thank our reviewers for helping us improve the paper. This material is based upon work supported by the National Science Foundation under Grant No. 2419798.

References

[1] Sergi Alvarez. Radare2: Libre Reversing Framework for Unix Geeks. `https://github.com/radareorg/radare2`, 2024.

[2] Bob Amstadt, Eric Youngdale, and Alexandre Julliard. Wine: Run Windows applications on Linux, BSD, Solaris and macOS. `https://www.winehq.org/`, 2024.

[3] Quynh Nguyen Anh. Capstone: Next Generation Disassembly Framework. https://www.capstone-engine.org/, 2024.

[4] Apple. Apple xcode: C++ language support. https://developer.apple.com/xcode/cpp/, 2024.

[5] Jordi Armengol-Estapé, Jackson Woodruff, Chris Cummins, and Michael F.P. O'Boyle. SLaDe: A Portable Small Language Model Decompiler for Optimized Assembly. In *IEEE/ACM International Symposium on Code Generation and Optimization*, CGO, 2024.

[6] Avast. Retdec: A retargetable machine-code decompiler based on LLVM. https://github.com/avast/retdec, 2024.

[7] Avast Software. RetDec issue #293: Code with fdivr decompiled wrong. https://github.com/avast/retdec/issues/293, 2018.

[8] Avast Software. RetDec issue #375: Incorrect decompile. https://github.com/avast/retdec/issues/375, 2018.

[9] Avast Software. RetDec issue #83: Avoid creating an empty if block. https://github.com/avast/retdec/issues/83, 2018.

[10] Avast Software. RetDec issue #669: Switch-cases wrongly deleted, indirect jump not reconstructed correctly. https://github.com/avast/retdec/issues/669, 2019.

[11] Avast Software. RetDec issue #673: label lab_generated_0 not declared and goto referencing it in decompile inner worlds. https://github.com/avast/retdec/issues/673, 2019.

[12] Avast Software. RetDec issue #717: Constants error on simpilify load instruction. https://github.com/avast/retdec/issues/717, 2020.

[13] Avast Software. RetDec issue #724: Incorrect translation:assembly to c. https://github.com/avast/retdec/issues/724, 2020.

[14] Avast Software. RetDec: Vtable-handling heuristics. https://github.com/avast/retdec/blob/master/src/rtti-finder/vtable/vtable_finder.cpp, 2024.

[15] Zion Leonahenahe Basque, Ati Priya Bajaj, Wil Gibbs, Jude O'Kain, Derron Miao, Tiffany Bao, Adam Doupé, Yan Shoshitaishvili, and Ruoyu Wang. Ahoy SAILR! There is No Need to DREAM of C: A Compiler-aware Structuring Algorithm for Binary Decompilation. In *USENIX Security Symposium*, SEC, 2024.

[16] Fabrice Bellard. QEMU, a Fast and Portable Dynamic Translator. In *USENIX Annual Technical Conference*, ATC, 2005.

[17] Fabrice Bellard. TCC: the Tiny C Compiler. https://bellard.org/tcc/, 2024.

[18] David Brumley, JongHyup Lee, Edward J Schwartz, and Maverick Woo. Native x86 Decompilation using {Semantics-Preserving} Structural Analysis and Iterative {Control-Flow} Structuring. In *USENIX Security Symposium*, SEC, 2013.

[19] Ying Cao, Runze Zhang, Ruigang Liang, and Kai Chen. Dsmith artifact. 2024.

[20] Ying Cao, Runze Zhang, Ruigang Liang, and Kai Chen. Evaluating the Effectiveness of Decompilers. In *ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA, 2024.

[21] Qibin Chen, Jeremy Lacomis, Edward J Schwartz, Claire Le Goues, Graham Neubig, and Bogdan Vasilescu. Augmenting Decompiler Output with Learned Variable Names and Types. In *USENIX Security Symposium*, SEC, 2022.

[22] Cristina Cifuentes. *Reverse Compilation Techniques*. PhD thesis, Queensland University of Technology, 1994.

[23] Christian Collberg. The Tigress C Obfuscator. https://tigress.wtf/introduction.html, 2024.

[24] Alessandro Di Federico, Mathias Payer, and Giovanni Agosta. Rev.Ng: a Unified Binary Analysis Framework to Recover CFGs and Function Boundaries. In *International Conference on Compiler Construction*, CC, 2017.

[25] Alessandro Di Federico, Mathias Payer, and Giovanni Agosta. Rev.Ng: Mach-O module. https://github.com/revng/revng/blob/develop/lib/Model/Importer/Binary/MachOImporter.cpp, 2024.

[26] Luboš Doležel. Darling: a translation layer that lets you run macOS software on Linux. https://www.darlinghq.org/, 2024.

[27] Karine Even-Mendoza, Arindam Sharma, Alastair F. Donaldson, and Cristian Cadar. GrayC: Greybox Fuzzing of Compilers and Analysers for C. In *ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA, 2023.

[28] Andrea Fioraldi. AFL-QEMU-Cov: Measure basic blocks coverage of all testcases in the afl queue

using a patched QEMU. https://github.com/andreafioraldi/afl-qemu-cov, 2024.

[29] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. AFL++: Combining Incremental Steps of Fuzzing Research. In *USENIX Workshop on Offensive Technologies*, WOOT, 2020.

[30] Cheng Fu, Huili Chen, Haolan Liu, Xinyun Chen, Yuandong Tian, Farinaz Koushanfar, and Jishen Zhao. Coda: An End-to-End Neural Program Decompiler. In *Advances in Neural Information Processing Systems*, NeurIPS, 2019.

[31] Google. ClusterFuzz, 2018.

[32] Qiuhan Gu. LLM-Based Code Generation Method for Golang Compiler Testing. In *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE, 2023.

[33] Andrea Gussoni, Alessandro Di Federico, Pietro Fezzardi, and Giovanni Agosta. A Comb for Decompiled C Code. In *ACM Asia Conference on Computer and Communications Security*, ASIACCS, 2020.

[34] HyungSeok Han, JeongOh Kyea, Yonghwi Jin, Jinoh Kang, Brian Pak, and Insu Yun. QueryX: Symbolic Query on Decompiled Code for Finding Bugs in COTS Binaries. In *IEEE Symposium on Security and Privacy*, Oakland, 2023.

[35] Hex-Rays. gooMBA: a Hex-Rays Decompiler plugin that simplifies Mixed Boolean-Arithmetic (MBA) expressions. https://github.com/HexRaysSA/goomba, 2024.

[36] Hex-Rays. IDA Pro: A Powerful Disassembler, Decompiler and a Versatile Debugger. https://hex-rays.com/ida-pro, 2024.

[37] Harshada Hole. Microsoft: Discover Hidden Insights with Profiler's External Code Decompilation. https://devblogs.microsoft.com/visualstudio/discover-hidden-insights-with-profilers-external-code-decompilation/, 2024.

[38] Intel. oneAPI DPC++/C++ Compiler: a Standards-based, Cross-architecture Compiler. https://www.intel.com/content/www/us/en/developer/tools/oneapi/dpc-compiler.html, 2024.

[39] Intel Corporation. The X86 Encoder Decoder. https://intelxed.github.io, 2024.

[40] Internet Archive. Windows XP Unofficial Service Pack 4. https://archive.org/details/xp-unofficial-sp4-jan2022_20220113, 2016.

[41] Joakim Kennedy. Go Reverse Engineering Tool Kit. https://github.com/goretk, 2024.

[42] Soomin Kim, Markus Faerevaag, Minkyu Jung, Seungll Jung, DongYeop Oh, JongHyup Lee, and Sang Kil Cha. Testing Intermediate Representations for Binary Analysis. In *IEEE/ACM International Conference on Automated Software Engineering*, ASE, 2017.

[43] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating Fuzz Testing. In *ACM SIGSAC Conference on Computer and Communications Security*, CCS, 2018.

[44] John Källén. Reko issue #529: Short circuit expressions not detected. https://github.com/uxmal/reko/issues/529, 2018.

[45] John Källén. Reko issue #543: Should encode chars >0x7f correctly in string literals. https://github.com/uxmal/reko/issues/543, 2018.

[46] John Källén. Reko issue #1114: Multiple issues in andn. https://github.com/uxmal/reko/issues/1114, 2021.

[47] John Källén. Reko issue #1149: Wrong decompilation of loop start. https://github.com/uxmal/reko/issues/1149, 2022.

[48] John Källén. Reko issue #1152: Wrong decompilation of movi/mvni. https://github.com/uxmal/reko/issues/1152, 2022.

[49] John Källén. Reko: a general purpose decompiler. https://github.com/uxmal/reko, 2024.

[50] John Källén. Reko: Control structure heuristics. https://github.com/uxmal/reko/tree/master/src/Decompiler/Structure, 2024.

[51] Jeremy Lacomis, Pengcheng Yin, Edward Schwartz, Miltiadis Allamanis, Claire Le Goues, Graham Neubig, and Bogdan Vasilescu. Dire: A Neural Approach to Decompiled Identifier Naming. In *IEEE/ACM International Conference on Automated Software Engineering*, ASE, 2019.

[52] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *IEEE/ACM International Symposium on Code Generation and Optimization*, CGO, 2004.

[53] Vladimir I Levenshtein. Binary Codes Capable of Correcting Deletions, Insertions, and Reversals. *Soviet Physics Doklady*, 10(8), 1966.

[54] Zhibo Liu and Shuai Wang. How Far We Have Come: Testing Decompilation Correctness of C Decompilers. In *ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA, 2020.

[55] Vsevolod Livinskii, Dmitry Babokin, and John Regehr. Random Testing for C and C++ Compilers with YARPGen. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA, 2020.

[56] LLVM. libClang. https://clang.llvm.org/docs/LibClang.html, 2024.

[57] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. {MOPT}: Optimized Mutation Scheduling for Fuzzers. In *USENIX Security Symposium*, SEC, 2019.

[58] POSIX Programmer's Manual. Truncf manpage. https://man7.org/linux/man-pages/man3/truncf.3p.html, 2024.

[59] Dhruv Maroo. GhidRust: Rust decompiler plugin for Ghidra. https://github.com/DMaroo/GhidRust, 2024.

[60] Microsoft. MSVC: Inlining Decisions in Visual Studio. https://devblogs.microsoft.com/cppblog/inlining-decisions-in-visual-studio/, 2024.

[61] Microsoft. Visual Studio Code: Develop C and C++ applications. https://visualstudio.microsoft.com/vs/features/cplusplus/, 2024.

[62] Eugene W. Myers. AnO(ND) Difference Algorithm and its Variations. *Algorithmica*, 1(1), 1986.

[63] Stefan Nagy and Matthew Hicks. Full-speed Fuzzing: Reducing Fuzzing Overhead through Coverage-guided Tracing. In *IEEE Symposium on Security and Privacy*, Oakland, 2019.

[64] National Security Agency. Ghidra issue #1054: Does not detect dos executables. https://github.com/NationalSecurityAgency/ghidra/issues/1054, 2019.

[65] National Security Agency. Ghidra issue #288: Invalid decompilation of cmpxchg instruction on x86_64. https://github.com/NationalSecurityAgency/ghidra/issues/288, 2019.

[66] National Security Agency. Ghidra issue #778: Decompiled f64 literals lack precision (e.g. for 1/4294967296.0). https://github.com/NationalSecurityAgency/ghidra/issues/778, 2019.

[67] National Security Agency. Ghidra issue #1054: 9.2: createinitializedblock() no longer supports memory region names with spaces. https://github.com/NationalSecurityAgency/ghidra/issues/1054, 2020.

[68] National Security Agency. Ghidra issue #3146: Mach-o library calls not handled properly. https://github.com/NationalSecurityAgency/ghidra/issues/3146, 2021.

[69] National Security Agency. Ghidra issue #3191: Incorrect packing in structures with unions. https://github.com/NationalSecurityAgency/ghidra/issues/3191, 2021.

[70] National Security Agency. Ghidra issue #4708: Ghidra doesn't handle floats in the msvc x86-64 calling convention properly. https://github.com/NationalSecurityAgency/ghidra/issues/4708, 2022.

[71] National Security Agency. Ghidra issue #4983: Improving work with function arguments. https://github.com/NationalSecurityAgency/ghidra/issues/4983, 2023.

[72] National Security Agency. Ghidra issue #5900: Incorrect sizing of data type in assignment leading to bad c output. https://github.com/NationalSecurityAgency/ghidra/issues/5900, 2023.

[73] National Security Agency. Ghidra issue #6119: Decompiler: double casting of variables. https://github.com/NationalSecurityAgency/ghidra/issues/6119, 2024.

[74] National Security Agency. Ghidra issue #6488: Missing and incorrrect variable assignments resulting from function calls. https://github.com/NationalSecurityAgency/ghidra/issues/6488, 2024.

[75] National Security Agency. Ghidra issue #6528: Decompiler erroneously swapping operators without changing the operand in floating-point comparisons where data is marked constant (x86). https://github.com/NationalSecurityAgency/ghidra/issues/6528, 2024.

[76] National Security Agency. Ghidra issue #6648: Sequence of rule applied affects the size ouput. https://github.com/NationalSecurityAgency/ghidra/issues/6648, 2024.

[77] National Security Agency. Ghidra issue #6708: [floats] truncated precision in decompiler view. https://github.com/NationalSecurityAgency/ghidra/issues/6708, 2024.

[78] National Security Agency. Ghidra Software Reverse Engineering Framework. https://ghidra-sre.org/, 2024.

[79] Kuntal Kumar Pal, Ati Priya Bajaj, Pratyay Banerjee, Audrey Dutcher, Mutsumi Nakamura, Zion Leonahenahe Basque, Himanshu Gupta, Saurabh Arjun Sawant, Ujjwala Anantheswaran, Yan Shoshitaishvili, et al. len or index or count, anything but v1": Predicting Variable Names in Decompilation Output with Transfer Learning. In *IEEE Symposium on Security and Privacy*, Oakland, 2024.

[80] Chengbin Pang, Ruotong Yu, Yaohui Chen, Eric Koskinen, Georgios Portokalidis, Bing Mao, and Jun Xu. SoK: All You Ever Wanted to Know About x86/x64 Binary Disassembly But Were Afraid to Ask. In *IEEE Symposium on Security and Privacy*, Oakland, 2021.

[81] PNF Software. JEB3: Reverse Engineering for Professionals. https://www.pnfsoftware.com/, 2024.

[82] radareorg. Radare2 issue #4422: Incorrect value of rax after xor eax,eax. https://github.com/radareorg/radare2/issues/4422, 2016.

[83] radareorg. Radare2 issue #9420: Some combo of a linkerscript, minimal sections copied, and a .eh_frame section cause r2 to see file as only 0xff bytes. https://github.com/radareorg/radare2/issues/9420, 2018.

[84] radareorg. Radare2 issue #17036: Incorrect detection of jump table's ending causing incorrect disassemblies in middle of opcodes (x86/64). https://github.com/radareorg/radare2/issues/17036, 2020.

[85] Relyze Software Limited. Relyze: Interactive Software Reverse Engineering. https://www.relyze.com/, 2024.

[86] Michael Rodler, David Paaßen, Wenting Li, Lukas Bernhard, Thorsten Holz, Ghassan Karame, and Lucas Davi. EF/CF: High Performance Smart Contract Fuzzing for Exploit Generation. In *IEEE European Symposium on Security and Privacy*, EuroS&P, 2023.

[87] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. AddressSanitizer: A Fast Address Sanity Checker. In *USENIX Annual Technical Conference*, ATC, 2012.

[88] Mayank Sharma, Pingshi Yu, and Alastair F Donaldson. Rustsmith: Random Differential Compiler Testing for Rust. In *ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA, 2023.

[89] Vidush Singhal, Akul Abhilash Pillai, Charitha Saumya, Milind Kulkarni, and Aravind Machiry. Cornucopia: A Framework for Feedback Guided Generation of Binaries. In *IEEE/ACM International Conference on Automated Software Engineering*, ASE, 2022.

[90] Robert Tarjan. Testing Flow Graph Reducibility. In *ACM Symposium on Theory of Computing*, STOC, 1973.

[91] The angr Project. Angr issue #2454: Inconsistent call graphs when resolving indirect calls. https://github.com/angr/angr/issues/2454, 2021.

[92] The angr Project. Angr issue #2911: Decompilation is not valid when there is double quote in string. https://github.com/angr/angr/issues/2911, 2021.

[93] The angr Project. Angr issue #2914: x86: Calling convention recovery is broken when multiple sizes of the same register are used. https://github.com/angr/angr/issues/2914, 2021.

[94] The angr Project. Angr issue #2915: x86: decompilation is missing exit condition of rep. https://github.com/angr/angr/issues/2915, 2021.

[95] The angr Project. Angr issue #3271: Ail optimization may incorrectly collapse converts. https://github.com/angr/angr/issues/3271, 2022.

[96] The angr Project. Angr issue #3377: test_decompilation_excessive_condition_removal is failing. https://github.com/angr/angr/issues/3377, 2022.

[97] The angr Project. Angr issue #3432: Return value type becomes int rather than long. https://github.com/angr/angr/issues/3432, 2022.

[98] The angr Project. Angr issue #3512: Failed switch-case identification. https://github.com/angr/angr/issues/3512, 2022.

[99] The angr Project. Angr issue #3702: Cfg incorrectly recovered? https://github.com/angr/angr/issues/3702, 2022.

[100] The angr Project. Angr issue #3703: Calling convention recovery. https://github.com/angr/angr/issues/3703, 2022.

[101] The angr Project. Angr issue #3737: Decompiler bug: incorrect arguments for xstrtol function call. https://github.com/angr/angr/issues/3737, 2023.

[102] The angr Project. Angr issue #3992: Calling convention gives wrong registers for arguments. https://github.com/angr/angr/issues/3992, 2023.

[103] The angr Project. Angr issue #4082: Incorrect loop ending under phonenix. https://github.com/angr/angr/issues/4082, 2023.

[104] The angr Project. Angr issue #4358: Incorrect goto edges reported from region simplifier. https://github.com/angr/angr/issues/4358, 2023.

[105] The angr Project. Angr issue #4420: Decompiler test test_decompiling_incorrect_duplication_chcon_main should have less gotos. https://github.com/angr/angr/issues/4420, 2024.

[106] The angr Project. Angr issue #4573: Expression incorrectly moved to lhs and not eliminated. https://github.com/angr/angr/issues/4573, 2024.

[107] Romain Thomas. Open-Obfuscator: O-MVLL & dProtect. https://obfuscator.re/, 2024.

[108] Vector 35. Binary Ninja: An interactive decompiler, disassembler, debugger, and binary analysis platform built by reverse engineers, for reverse engineers. https://binary.ninja/, 2024.

[109] Vector 35. Opaque Predicate Patcher. https://github.com/Vector35/OpaquePredicatePatcher, 2024.

[110] Vector35. Binary Ninja issue #1074: Analysis seems to ignore tailcalls in doing possible value analysis. https://github.com/Vector35/binaryninja-api/issues/1074, 2018.

[111] Vector35. Binary Ninja issue #1434: Incorrect jump table values. https://github.com/Vector35/binaryninja-api/issues/1434, 2019.

[112] Vector35. Binary Ninja issue #1581: Hlil incorrect control flow. https://github.com/Vector35/binaryninja-api/issues/1581, 2020.

[113] Vector35. Binary Ninja issue #1627: Sort hlil cases in switch statements. https://github.com/Vector35/binaryninja-api/issues/1627, 2020.

[114] Vector35. Binary Ninja issue #1789: Hlil confusing/incorrect if statement conditions. https://github.com/Vector35/binaryninja-api/issues/1789, 2020.

[115] Vector35. Binary Ninja issue #1791: Lookup table calculation fails. https://github.com/Vector35/binaryninja-api/issues/1791, 2020.

[116] Vector35. Binary Ninja issue #2824: continue statement wrongly optimized out during optimizations/lifting within a jump table. https://github.com/Vector35/binaryninja-api/issues/2824, 2022.

[117] Vector35. Binary Ninja issue #3186: Lifting llil->mlil shift value lifted incorrectly. https://github.com/Vector35/binaryninja-api/issues/3186, 2022.

[118] Vector35. Binary Ninja issue #3191: Shortcomings of bn's handling of float/vector values in the windows x64 calling convention. https://github.com/Vector35/binaryninja-api/issues/3191, 2022.

[119] Vector35. Binary Ninja issue #3719: Binja does not define second string when comparing. https://github.com/Vector35/binaryninja-api/issues/3719, 2022.

[120] Vector35. Binary Ninja issue #4110: Problems with decompiler splitting a 64-bit store into stores of two 32-bit floats. https://github.com/Vector35/binaryninja-api/issues/4110, 2023.

[121] Vector35. Binary Ninja issue #4252: Hlil generating constant if-condition. https://github.com/Vector35/binaryninja-api/issues/4252, 2023.

[122] Vector35. Binary Ninja issue #4368: Function ... hlil has invalid goto targets. https://github.com/Vector35/binaryninja-api/issues/4368, 2023.

[123] Vector35. Binary Ninja issue #4692: Don't display integer arguments as strings. https://github.com/Vector35/binaryninja-api/issues/4692, 2023.

[124] Vector35. Binary Ninja issue #4800: Wrong cast. https://github.com/Vector35/binaryninja-api/issues/4800, 2023.

[125] Vector35. Binary Ninja issue #3081: Placeholder title. https://github.com/Vector35/binaryninja-api/issues/3081, 2024.

[126] Vector35. Binary Ninja issue #4223: Placeholder title. https://github.com/Vector35/binaryninja-api/issues/4223, 2024.

[127] Vector35. Binary Ninja issue #4971: Function is called without a parameter even though its type requires one. https://github.com/Vector35/binaryninja-api/issues/4971, 2024.

[128] Vector35. Binary Ninja issue #4977: Incorrect main function detection in pe, pdb causes duplicate. https://github.com/Vector35/binaryninja-api/issues/4977, 2024.

[129] Vector35. Binary Ninja issue #5009: Cannot figure out the parameter of the call if the parameters are moved into the stack rather than pushed. https://github.com/Vector35/binaryninja-api/issues/5009, 2024.

[130] Vector35. Binary Ninja issue #5030: Inconsistent display of bool values in conditional statements. https://github.com/Vector35/binaryninja-api/issues/5030, 2024.

[131] Vector35. Binary Ninja issue #5650: No-return not being treated as such if the call instruciton has unresolved stack usage. https://github.com/Vector35/binaryninja-api/issues/5650, 2024.

[132] Fish Wang and Yan Shoshitaishvili. Angr - The Next Generation of Binary Analysis. In *IEEE Cybersecurity Development*, SecDev, 2017.

[133] Mingyuan Wu, Ling Jiang, Jiahong Xiang, Yanwei Huang, Heming Cui, Lingming Zhang, and Yuqun Zhang. One Fuzzing Strategy to Rule Them All. In *International Conference on Software Engineering*, ICSE, 2022.

[134] Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. Fuzz4All: Universal Fuzzing with Large Language Models. In *IEEE/ACM International Conference on Software Engineering*, ICSE, 2024.

[135] Danning Xie, Zhuo Zhang, Nan Jiang, Xiangzhe Xu, Lin Tan, and Xiangyu Zhang. ReSym: Harnessing LLMs to Recover Variable and Data Structure Symbols from Stripped Binaries. In *ACM SIGSAC Conference on Computer and Communications Security*, CCS, 2024.

[136] Khaled Yakdan, Sergej Dechand, Elmar Gerhards-Padilla, and Matthew Smith. Helping Johnny to Analyze Malware: A Usability-Optimized Decompiler and Malware Analysis User Study. In *IEEE Symposium on Security and Privacy*, Oakland, 2016.

[137] Khaled Yakdan, Sebastian Eschweiler, Elmar Gerhards-Padilla, and Matthew Smith. No More Gotos: Decompilation Using Pattern-Independent Control-Flow Structuring and Semantic-Preserving Transformations. In *Network and Distributed System Security Symposium*, NDSS, 2015.

[138] Chenyuan Yang, Yinlin Deng, Runyu Lu, Jiayi Yao, Jiawei Liu, Reyhaneh Jabbarvand, and Lingming Zhang. WhiteFox: White-Box Compiler Fuzzing Empowered by Large Language Models. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA, 2024.

[139] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and Understanding Bugs in C Compilers. In *ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI, 2011.

[140] Yupeng Yang, Yongheng Chen, Rui Zhong, Jizhou Chen, and Wenke Lee. Towards Generic Database Management System Fuzzing. In *USENIX Security Symposium*, SEC, 2024.

[141] Chang Zhu, Ziyang Li, Anton Xue, Ati Priya Bajaj, Wil Gibbs, Yibo Liu, Rajeev Alur, Tiffany Bao, Hanjun Dai, Adam Doupé, et al. {TYGR}: Type Inference on Stripped Binaries using Graph Neural Networks. In *USENIX Security Symposium*, SEC, 2024.

[142] Muqi Zou, Arslan Khan, Ruoyu Wu, Han Gao, Antonio Bianchi, and Dave Jing Tian. {D-Helix}: A Generic Decompiler Testing Framework Using Symbolic Differentiation. In *USENIX Security Symposium*, SEC, 2024.

[143] Zynamics. Bindiff: a comparison tool for binary files. https://www.zynamics.com/bindiff.html, 2024.

HEC: Equivalence Verification Checking for Code Transformation via Equality Saturation

Jiaqi Yin* Zhan Song* Nicolas Bohm Agostini† Antonino Tumeo† Cunxi Yu*

*University of Maryland, College Park †Pacific Northwest National Laboratory
{jyin629,cunxiyu}@umd.edu

Abstract

In modern computing systems, compilation employs numerous optimization techniques to enhance code performance. Source-to-source code transformations, which include control flow and datapath transformations, have been widely used in High-Level Synthesis (HLS) and compiler optimization.

While researchers actively investigate methods to improve performance with source-to-source code transformations, they often overlook the significance of verifying their correctness. Current tools cannot provide a holistic verification of these transformations. This paper introduces HEC, a framework for equivalence checking that leverages the e-graph data structure to comprehensively verify functional equivalence between programs. HEC utilizes the MLIR as its frontend and integrates MLIR into the e-graph framework. Through the combination of dynamic and static e-graph rewriting, HEC facilitates the validation of comprehensive code transformations.

We demonstrate effectiveness of HEC on PolyBenchC benchmarks, successfully verifying loop unrolling, tiling, and fusion transformations. HEC processes over 100,000 lines of MLIR code in 40 minutes with predictable runtime scaling. Importantly, HEC identified two critical compilation errors in mlir-opt: loop boundary check errors causing unintended executions during unrolling, and memory read-after-write violations in loop fusion that alter program semantics. These findings demonstrate HEC practical value in detecting real-world compiler bugs and highlight the importance of formal verification in optimization pipelines.

1 Introduction

In the post-Moore era, the pressing need to enhance the runtime performance of computing systems has driven extensive research into optimization strategies. Source-to-source transformations, such as control flow and datapath transformations, have been widely used to advance computing efficiency. Control flow transformations optimize parameters like unrolling and tiling factors to enhance hardware parallelism and performance [3, 45, 55]. Furthermore, there is significant interest in datapath and gate-level transformations [11, 16, 24, 56]. These combined efforts highlight a multi-faceted approach to optimize computing efficiency in various areas, including HLS [14, 15, 27] and compiler optimization [26]. However, the stability and correctness of these transformations are often not guaranteed. It is essential to ensure the equivalence between the programs before and after transformations.

Existing verification approaches typically focus on a single aspect of verification, thus failing to provide a comprehensive framework for verifying multi-domain transformations. For example, Polycheck [7] and ISA [2] attempt to verify equivalence for affine problems, devoted to control flow transformations. Furthermore, Samuel et al. [18] have developed verification methods specifically for datapath transformations in RTL code. These verification tools are limited in scope, and unable to offer comprehensive equivalence checking for both control flow and datapath transformations simultaneously.

As an emerging technique in compiler optimization [12,46], hardware design automation [9, 10, 16, 48, 49, 54, 58], and theorem proving [4, 5, 21, 23], e-graph can represent multiple equivalent expressions within a single graph data structure. This capability enables possibilities of validation between optimized code against its original version. In this work, we propose HEC, a formal equivalence checking framework based on e-graph rewriting. HEC is designed to provide a holistic verification of codes through equality saturation. HEC combines static rewriting rules for datapath transformations with dynamic rewriting capabilities for control flow transformations. The main contributions of HEC are summarized as follows:

- Introducing HEC, an e-graph based verification framework taking MLIR codes as inputs and performing equivalence checking. It simultaneously verifies control flow and datapath transformations, ensuring holistic equivalence checking of multi-domain transformations with equality saturation.
- Designing a hybrid ruleset to combine static rewriting rules with dynamic rewriting capabilities within the e-graph infrastructure, enabling the framework to adapt to runtime-dependent transformation parameters and accommodate diverse transformation patterns.

```
1 %av, %bv: memref<101xi1>
2
3 %true = arith.constant true
4 affine.for %arg1=0 to 101 {
5   %1=affine.load %av[%arg1]:memref<101xi1>
6   %2=affine.load %bv[%arg1]:memref<101xi1>
7   %3=arith.andi %1, %2:i1
8   %4=arith.xori %3, %true:i1
9 }
```

Listing 1: Baseline Code

```
1 %av, %bv: memref<101xi1>
2
3 affine.for %arg1=0 to 101 {
4   %true = arith.constant true
5   %1=affine.load %av[%arg1]:memref<101xi1>
6   %2=affine.load %bv[%arg1]:memref<101xi1>
7   %3=arith.andi %1, %2:i1
8   %4=arith.xori %3, %true:i1
9 }
```

Listing 2: Variant B

```
1 %av, %bv: memref<101xi1>
2
3 %true = arith.constant true
4 affine.for %arg1=0 to 101 {
5   %1=affine.load %av[%arg1]:memref<101xi1>
6   %2=affine.load %bv[%arg1]:memref<101xi1>
7   %3=arith.xori %1, %true:i1
8   %4=arith.xori %2, %true:i1
9   %5=arith.ori %3, %4:i1
10 }
```

Listing 3: Variant C

```
1 %av, %bv: memref<101xi1>
2
3 %true = arith.constant true
4 affine.for %arg1=0 to 101 step 3 {
5  affine.for %arg2= %arg1 to min (%arg1+3, 101) {
6   %1=affine.load %av[%arg2]:memref<101xi1>
7   %2=affine.load %bv[%arg2]:memref<101xi1>
8   %3=arith.andi %1, %2:i1
9   %4=arith.xori %3, %true:i1
10  }
11 }
```

Listing 4: Variant D

Figure 1: Variants B, C, and D achieve equivalence with the baseline (Listing 1) through loop hoisting, De Morgan's laws, and loop tiling, respectively. This multifaceted transformation, encompassing both datapath and control flow aspects, highlights the challenges in code verification. HEC offers a hybrid, combined solution to address this challenge.

- Introducing a robust graph representation to model MLIR input code, encapsulating both control flow and datapath structures. This representation unifies variable renaming, loop decomposition, and operation tracking, serving as an interface for integration with the e-graph framework.
- Demonstrating the scalability and efficiency of HEC through comprehensive evaluations on PolyBenchC benchmarks. Specifically, HEC can process over 100,000 lines of MLIR code in 40 minutes.
- Identifying two types of compilation errors introduced by the MLIR compiler with `mlir-opt`, in PolyBenchC [40,50] benchmark suite.

2 Motivating Example

Unified datapath/control flow verification: Code transformation is a critical aspect of code optimization, encompassing both control flow transformations and data path transformations. Typical control flow transformations include loop unrolling, tiling, and fusion, etc [19, 42, 47, 59]. Additionally, numerous studies focus on optimizing the computation graph at the operator level for datapath optimization [16, 17, 24, 56]. For instance, Jia et al. [24] optimize deep neural network (DNN) computations through graph substitution. These code optimizations underscore the necessity for rigorous verification of code transformations. In this section, we illustrate the challenges associated with code transformation verification using the example presented in Figure 1.

Listing 1 presents the baseline code that iterates over two vectors, each of length 101. For each pair of elements `a` and `b`, the loop performs the `NAND(a, b)` operation. In Listing 3, a datapath transformation is demonstrated using De Morgan's law, resulting in the `OR(a', b')` operation. Listing 2 and Listing 1 are functionally equivalent through loop hoisting, where the variable `%true` is moved inside the loop body. Similarly, Listing 4 and Listing 1 are equivalent via loop tiling, where the loop on Line 4 in Listing 1 is split into two loops.

However, conventional verification frameworks, including PolyCheck [7] and ISA [2], focus on only one aspect of verification. A holistic verification approach is essential to ensure the correctness and reliability of code transformations across all optimization levels, including control flow and datapath transformations. Individually verifying each transformation layer may overlook complex dependencies and interactions that can lead to subtle bugs or functional discrepancies.

Fortunately, e-graphs [53] enable the simultaneous representation of multiple equivalent program expressions, effectively managing dependencies and interactions between control flow and datapath transformations. They support bidirectional rewriting rules, allowing transformations to be applied flexibly while maintaining equivalence across optimization levels. Additionally, e-graphs offer a compact representation of equivalence classes, reducing computational overhead and enhancing scalability for large codebases. These advantages collectively motivate the development of a novel e-graph based verification approach. In this paper, we introduce how the four variants in Figure 1 are integrated into the e-graph.

Challenges: Although e-graphs provide prominent merits for efficient verification, conventional term rewriting still poses challenges when integrating control flow transformations. For example, loop tiling in Listing 4 creates a new loop variable `%arg2`, and the loop body (lines 6 to 9) replaces all instances that originally consume `%arg1` with the newly created variable `%arg2`. Additionally, the tiling factor (e.g.,

3 in Listing 4) is unknown until the compilation of the e-graph. Consequently, it is challenging to develop a static term rewriting pattern that can universally represent all tiling scenarios across diverse input codes. To effectively incorporate code equivalence checking within the e-graph framework, the rewriting rules must also include information that is only determinable at runtime, such as the specific tiling factor, variable names generated during transformation, and loop bounds that depend on input parameters. In these complex scenarios, *dynamic rewriting* rules must be tailored to the specific characteristics and requirements of the input code. Static rules alone are insufficient to handle runtime-dependent factors, which can vary significantly across different codebases and transformation instances. By customizing dynamic rules, HEC can accurately capture and verify intricate control flow transformation patterns, ensuring robust and reliable code equivalence checking even in highly sophisticated code structures. A more detailed discussion of the presented challenges and dynamic rewriting is provided in Section 4.2.

To this end, we propose HEC, a formal equivalence verification framework utilizing e-graphs with a hybrid approach of static and dynamic rewriting. The integration of both static and dynamic rewriting rules within e-graph ensures robust verification of multi-domain source-to-source transformations. In the following sections, we demonstrate how the different variants from Listing 2 to Listing 4 are verified to be equivalent to Listing 1 using various approaches: HEC graph representation, static rewriting, and dynamic rewriting.

3 Background

3.1 Source-to-Source Equivalence Checking

Code equivalence checking is a critical challenge in software engineering, which involves confirming that two code segments, despite having different syntactic or structural forms, exhibit identical behaviors under all operational conditions. This verification ensures that transformations made for optimization, refactoring, or adaptation do not inadvertently alter the functional output or introduce undefined behavior.

Consider two programs, P_A and P_B, that operate on a common input set I. P_A produces a set of possible outputs O_A, while P_B yields an output set O_B, both corresponding to the input set I. P_A and P_B are deemed functionally equivalent if and only if their output sets are identical for all possible inputs in I. This relationship can be formally expressed as:

$$\forall I, \quad O_A(I) = O_B(I).$$

This equation asserts that for every input I, the output $O_A(I)$ of program P_A must exactly match the output $O_B(I)$ of program P_B, thereby affirming their functional equivalence.

Some studies attempt to establish program equivalence checking across various domains [2, 7, 25, 41, 43, 52]. Weilei et al. [7] introduced PolyCheck, a system designed to verify the equivalence of affine programs and their transformations through dynamic analysis. However, the applicability of PolyCheck is fundamentally limited to affine transformations, thereby restricting its use in broader code transformations. Similarly, other frameworks, such as ISA [2], suffer from restricted transformation coverage. Generally, these frameworks are sensitive to the format of the input code, and none can comprehensively provide equivalence checking that spans data path and control flow transformations simultaneously.

3.2 Multi-Level Intermediate Representation

MLIR [29] (Multi-Level Intermediate Representation) is a compiler infrastructure within the LLVM [28] project. Developed to address the complexity of generating efficient code for diverse hardware targets, MLIR facilitates the co-design of high-level optimizations and hardware-specific transformations. By providing a unified interface for different levels of abstraction, from high-level algorithmic constructs down to low-level hardware instructions, MLIR supports various domain-specific optimizations. Its modular design allows developers to define custom dialects and transformations, making it highly adaptable to various computing paradigms.

Some studies focus on optimizing and transforming the MLIR infrastructure to enhance HLS and code generation. For instance, CIRCT [13] converts MLIR to RTL code, serving as an open-source HLS tool. Cheng et al. [11] propose a MLIR based code optimization explorer based on the e-graph to address the phase-ordering problem [32]. Tools such as HeteroCL [26] and ScaleHLS [57] transform MLIR into HLS C, while other projects like SODA [3] and POLSCA [61] convert it into LLVM IR. Despite these developments, there is a lack of frameworks capable of supporting transformation verification for MLIR.

3.3 Equality Saturation

An e-graph (equivalence graph) is a data structure that compactly represents a congruence relation over expressions [37, 38, 46]. Widely utilized in compiler optimizations, program analysis, and formal verification, e-graph efficiently manages large sets of equivalent expressions by sharing common sub-expressions, allowing it to store an exponential number of expressions in linear space.

An **e-graph** $\mathcal{G}$ is formally defined as a tuple $\mathcal{G} = (N, E, \phi)$:

- N is a finite set of nodes (also called **e-nodes**). E-nodes N represent individual expressions or sub-expressions.
- $E \subseteq N \times N$ is an equivalence relation on N, partitioning N into equivalence classes known as **e-classes**. E-classes are the equivalence classes induced by E. Nodes within the same e-class are considered equivalent under the equivalence relation E. Here E has the property of closure under

congruence [36, 37] which ensures that the equivalence relation is preserved under the application of operators.

- $\phi : N \to \Sigma_k \times N^k$ is a labeling function that maps each node to an operator and a list of child nodes, where:
  - Σ_k is the set of operators of operands number k.
  - N^k denotes an tuple of k child nodes from N.

E-graphs have been used to power the optimization technique called **equality saturation**, which is an optimization process that exhaustively applies a set of rewriting rules to an initial expression within an e-graph until no new equivalence can be added. Formally, the following are defined:

- $\mathcal{G} = (N, E, \phi)$ denotes an e-graph,
- $R = \{(l_1, r_1), \ldots, (l_m, r_m)\}$ is the rewriting ruleset,
- g_0 is the initial expression encoded in $\mathcal{G}_0$.

The process involves three main steps. First, construct $\mathcal{G}_0$ from the initial expression g_0. Second, for each pair $(l_i, r_i) \in R$, identify matches within $\mathcal{G}_t$ and add the equivalences $[l_i]_E \sim [r_i]_E$ to form $\mathcal{G}_{t+1}$ via term rewriting [22]. Finally, iterate until $\mathcal{G}_{t+1} = \mathcal{G}_t$, indicating that no further changes occur. A detailed example of e-graph exploration is shown in Figure 2.

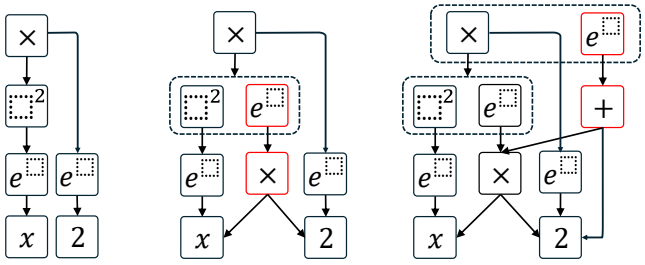


Figure 2: The e-graph rewriting for the expression $(e^x)^2 \times e^2$. **Left**: Initial e-graph. **Middle**: E-graph after applying the rewriting rule $(e^x)^2 \times e^2 \Rightarrow e^{2x} \times e^2$. **Right**: E-graph following the rewriting rule $e^{2x} \times e^2 \Rightarrow e^{2x+2}$. Dotted boxes represent e-classes; arrows show connections between e-nodes and their e-classes. Newly added e-nodes are highlighted in red.

Equality saturation has sparked research interests including compiler optimization [46], hardware design automation [16, 48, 54], theorem proving [4, 5, 21, 23], and more [8, 39, 44, 51, 56]. E-graph is supported by SMT solvers, such as Z3 [20, 21], and Willsey et al. developed an advanced e-graph framework **egg** for equality saturation [53, 60]. In this work, HEC incorporates the egg for e-graph implementation.

Formally, the functional equivalence of two programs P_A and P_B can be defined by representing programs within a shared e-graph $\mathcal{G} = (N, E, \phi)$ and verifying that their corresponding e-nodes $n_A, n_B \in N$ reside within the same equivalence class. Specifically, the e-graph exhaustively applies a comprehensive set of rewriting rules R to capture all possible code transformations. If, after saturation, n_A and n_B can be united in the same e-class, then programs P_A and P_B are deemed functionally equivalent. This approach transforms program equivalence checking into determining e-class membership within the e-graph, thereby ensuring that P_A and P_B exhibit identical behaviors across all inputs I.

4 Proposed Approach

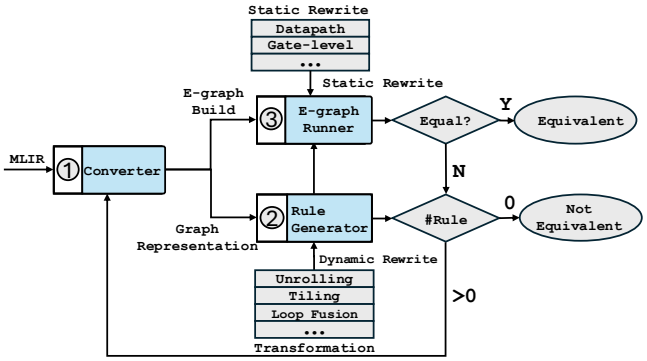


Figure 3: Overview of the HEC Framework: Initially, the input MLIR code is converted into a graph-based representation through dataflow analysis (step ①). This graph representation serves as a foundation for constructing the e-graph. The dynamic rules (step ②) are specifically designed to handle complex loop transformations that are not suitable for static rewriting. Finally, the e-graph runner (step ③) employs a hybrid ruleset that includes both static datapath rewriting and dynamic rewriting.

Here we provide an overview of HEC. To enhance versatility and avoid confinement to any specific programming language, HEC utilizes MLIR as its front-end input language. Figure 3 offers a comprehensive overview of the HEC approach. The core functionality of HEC lies in the verification of input code after datapath and control flow transformations, leveraging the e-graph with a hybrid static/dynamic rewriting ruleset.

HEC accepts MLIR as input language and any framework capable of generating MLIR code can serve as the front end, such as Polygeist [35] and IREE [1]. HEC incorporates a converter that processes the input MLIR code into a graph representation, acting as an interface for MLIR and the e-graph runner. Details of this graph representation are elaborated in Section 4.1, corresponding to step ① in Figure 3.

A major challenge within the HEC framework is to manage control flow transformations, such as loop unrolling and tiling. Standard e-graph rewriting techniques typically struggle to accurately represent these transformations. To address this issue, we have developed a dynamic rule generator (step ②) that creates rewriting rules at runtime to handle control flow transformations. This enables the creation of custom rewriting

rules tailored to each unique code input. The dynamic rule generation process is elaborated in Section 4.2.

Moreover, HEC incorporates static datapath rewriting rules. In Section 4.3, we present the e-graph based verification flow and the collaboration between static and dynamic rules, which are associated with step ③ in Figure 3. Additionally, we provide a detailed example to illustrate the process.

4.1 MLIR graph representation

Integrating the input MLIR code into the e-graph data structure requires translating the code into a graph representation. This translation process (step ①) involves two primary tasks to ensure the correctness of the mapping between the MLIR code and the graph representation. First, it needs to collect all necessary information to construct the e-graph, including dialects, operations, data type information, and other relevant metadata. Second, this involves explicitly representing loop transformation structures to ensure the graph accurately captures essential loop parameters like start, end, step, and the loop body. The generated graph representation functions as a bridge between MLIR, the e-graph runner, and the rule generator. Both the e-graph runner and the dynamic rule generator use the graph representation as input for further process.

Task 1 – The essential information encapsulated in graph representation closely mirrors the structure and characteristics of MLIR operations. We designed this representation akin to an Abstract Syntax Tree (AST), where the graph nodes correspond to individual MLIR operations. Each node comprehensively encapsulates critical information, such as the operator's name, input terms, data types, and dimensions. The interconnections between nodes are depicted as graph edges.

To construct the graph representation, we need to establish a one-to-one mapping between variable names and computation nodes. According to MLIR syntax, multiple variables may share the same variable name within their respective scopes. To address this issue, we rename all variables based on their global order of appearance in the input MLIR code. For example, if a variable %1 exists in multiple loop scopes, the variable in the subsequent loops will be renamed according to its appearance order. Similarly, all computation nodes are indexed based on their order of appearance. For example, the load operation in line 6 of Listing 1 is shown in Listing 5.

```
1 Vertex Name: Affine_Load_1
2 Dtype: i1
3 Dimension: 1
4 Input Edges: %bv, %arg1
5 Output Edges: %2
6
7 Edge Name: %2
8 Source Vertex: Affine_Load_1
9 Target Vertices: Arith_Andi_0
```

Listing 5: The graph representation attributes for the load operation and the term %2 in Listing 1 (line 6). In this example, the load operation is indexed based on its order of appearance, and is assigned an index of 1

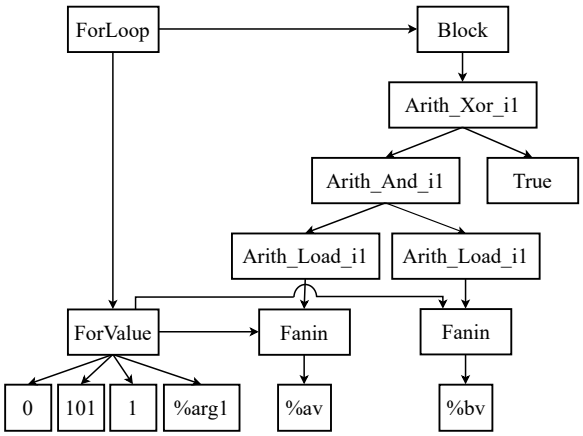


Figure 4: The dataflow graph for Listing 1

Task 2 – The second challenge involves accurately representing *for* loop constructs within our graph representation. To address this, we decompose each *for* loop into two primary components: a *loopvalue* component and a *block* operation. A visualization of Listing 1 is provided in Figure 4. Specifically, the structure of a *for* loop is represented as follows:

- The *loopvalue* component encapsulates the loop parameters, including the loop start value, end value, and increment step. All operations that use the loop variable as input will consume the *loopvalue* component instead.
- To differentiate loop variables that share the same start, end, and step values, the *loopvalue* component includes the variable name as an additional input. This differentiation is not necessary for common datapath nodes such as *add*.
- The *block* operation represents the loop body. Specifically, the inputs to the *block* operation consist of the output terms from isolated operations performed within the loop body. Isolated output terms are those not consumed by any other operations within the block scope. For example, only %4 is isolated in Listing 1. The other output terms, from %1 to %3, are consumed by subsequent operations, thereby making %4 the only output for the *block* operation.
- For operations that do not produce output data, such as affine.store, HEC introduces a pseudo output term to maintain operation tracking. All pseudo output terms are considered isolated.
- The sequence of operations within the *block* operation is order-sensitive, ensuring that the original operation order is preserved within the loop block.
- All hierarchical structures, including *for*, *func*, and *if* constructs, maintain similar block operations.

Another benefit of such representation is its ability to automatically unify code transformations resulting from variable

Table 1: Selection of static rewriting rules integrating into HEC to manage datapath transformations. All datapath rewriting rules are signage and bitwidth dependent.

Class	Left-hand Side	Right-hand Side
Datapath	$a \ll b$	$a \times 2^b$
	$(a \times b) \ll c$	$(a \ll c) \times b$
	$a \times b \times c$	$a \times (b \times c)$
	$(a \ll b) \ll c$	$a \ll (b + c)$
Gate-level	$\neg(a \& b)$	$\neg a \parallel \neg b$
	$a \oplus b$	$(a \wedge \neg b) \vee (\neg a \wedge b)$
	$a \oplus 0$	a
	$\neg a$	$a \oplus True$

movements without affecting data dependencies. For example, Listing 2 is transformed from Listing 1 through loop hoisting. In this transformation, the variable `%true` is moved inside the loop body. Since `%true` is consumed by the operation `arith.xori`, it is not isolated, and the block body does not track the term order. Both Listing 1 and Listing 2 correspond to the same dataflow depicted in Figure 4. The HEC graph representation automatically unifies the loop hoisting transformation before applying e-graph rewriting.

4.2 Hybrid E-graph Rewriting

In this section, we introduce the HEC hybrid rewriting rule sets, which encompass both static datapath rewriting rules and dynamic rewriting rules. We specifically focus on the dynamic rewriting rules, which are designed to handle control flow transformations.

Static Rewriting: HEC incorporates a comprehensive suite of static rewriting rules, including 62 bitwidth-dependent datapath rewriting rules. A selection of these static rewriting rules is presented in Table 1. These static rules are specifically designed to rewrite the input program at the operator level using Boolean algebra and standard arithmetic algebra, enabling efficient datapath transformations.

For example, as illustrated in Listing 3, the framework performs `OR(a', b')` operations for the input operands. Here, the NOT operation `NOT(a)` is expressed as `XOR(a, True)` through the rewriting rule $\neg a \Leftrightarrow a \oplus \texttt{True}$. Similarly, the operations in Listing 1 form `NAND(a, b)`. These two code segments collectively constitute the left-hand side and right-hand side of the rewriting rule $\overline{a \& b} \Leftrightarrow \overline{a} \| \overline{b}$, after which the two code segments are unified into the same e-class.

Why we need dynamic rewriting: Pre-defined static rewriting rules provide robust support for datapath transformations, but they face significant challenges when applied to control flow transformations. To illustrate these challenges and to highlight the need for dynamic rewriting rules, we present a simple example.

```
%av, %bv: memref<101xi1>

%true = arith.constant true
affine.for %arg1 = 0 to 100 step 2 {
%1 = affine.load %av[%arg1] : memref<101xi1>
%2 = affine.load %bv[%arg1] : memref<101xi1>
%3 = arith.andi %1, %2 : i1
%4 = arith.xori %3, %true : i1
%5 = affine.apply affine_map<(d0) -> (d0 + 1)>(%arg1)
%6 = affine.load %av[%5] : memref<101xi1>
%7 = affine.load %bv[%5] : memref<101xi1>
%8 = arith.andi %6, %7 : i1
%9 = arith.xori %8, %true : i1
}
affine.for %arg2 = 100 to 101 {
%10 = affine.load %av[%arg2] : memref<101xi1>
%11 = affine.load %bv[%arg2] : memref<101xi1>
%12 = arith.andi %10, %11 : i1
%13 = arith.xori %12, %true : i1
}
```

Listing 6: Unrolling variant with Listing 1

```
(block
(forcontrol
    (forvalue 0 101 1 %arg1)
    (block
        (arith_xori_i1 (arith_andi_i1 (load_i1 (fanin %av
            (forvalue 0 101 1 %arg1))) (load_i1 (fanin
        %bv (forvalue 0 101 1 %arg1)))) (
        arith_constant_i1 1))
    )))
```

Listing 7: Rewriting rule lhs corresponding to Listing 1

```
#map = affine_map<(d0) -> (d0 + 1)>
(combine
(forcontrol
    (forvalue 0 100 2 %arg1)
    (block
        (arith_xori_i1 (arith_andi_i1 (load_i1 (fanin %av
            (forvalue 0 100 2 %arg1))) (load_i1 (fanin
        %bv (forvalue 0 100 2 %arg1)))) (
        arith_constant_i1 1))
        (arith_xori_i1 (arith_andi_i1 (load_i1 (fanin %av
            (apply (map0 (forvalue 0 100 2 %arg1))))) (
        load_i1 (fanin %bv (apply (map0 (forvalue 0
        100 2 %arg1)))))) (arith_constant_i1 1))
    ))
(forcontrol
    (forvalue 100 101 1 %arg2)
    (block
        (arith_xori_i1 (arith_andi_i1 (load_i1 (fanin %av
            (forvalue 100 101 1 %arg2))) (load_i1 (
        fanin %bv (forvalue 100 101 1 %arg2)))) (
        arith_constant_i1 1))
    )))
```

Listing 8: Rewriting rule rhs corresponding to Listing 6

Consider the MLIR code shown in Listing 6, which is an unrolled version of Listing 1 with an unrolling factor of 2. To establish the equivalence between these two codes, the e-graph runner requires a rewriting rule of the form $lhs \Leftrightarrow rhs$, where the left-hand side (lhs) and right-hand side (rhs) are provided in Listings 7 and 8, respectively. However, constructing static rewriting rules to capture such transformations presents several challenges: (1) The number of operations within the *block* operation (line 5 in Listing 8) varies according to the unrolling factor. Static e-graph rewriting rules necessitate a predetermined number of input parameters prior to e-graph runner compilation, making it difficult to accommodate varying operation counts; (2) The isolated dataflow structures need to be isomorphic (line 5 in Listing 7 compared to lines 6, 7, and 12 in Listing 8). Assessing the isomorphism of these

Table 2: Selection of dynamic rewriting integrated into HEC to address control flow transformations. The pattern and transformation are interchangable if they meet the conditions.

	Left-hand Side	Right-hand Side	Condition

Unrolling	for **m1** to **n2** step **k2**:
	  Loop-body-2
		for **m1** to **n1** step **k1**:
		  Loop-body-1
		for **m2** to **n2** step **k2**:
		  Loop-body-2
			1. $\lceil(\mathbf{n2}-\mathbf{m1})/\mathbf{k2}\rceil == \lceil(\mathbf{n2}-\mathbf{m2})/\mathbf{k2}\rceil + \lceil(\mathbf{n1}-\mathbf{m1})/\mathbf{k1}\rceil \times (\mathbf{k1}/\mathbf{k2})$
			2. Loop-body-1 is **k1/k2** times replication of Loop-body-2

Tiling	for **%1** = **m1** to **n1** step **k2**:
	  Loop-body
		for **%1** = **m1** to **n1** step **k1**:
		  for **%2** = **%1** to **n2** step **k2**:
		  Loop-body
			1. $\mathbf{k1} == f * \mathbf{k2}$ (f is the tiling factor)
			2. $\mathbf{n2} = \min(\mathbf{\%1}+\mathbf{k1}, \mathbf{\%2})$

Fusion	for **m** to **n** step **k1**:
	  Loop-body-1
	for **m** to **n** step **k2**:
	  Loop-body-2
		for **m** to **n** step **k1** × **k2**:
		  Loop-body-3
			1. Loop-body-3 is **k2** times replication of loop-body-1 plus **k1** times replication of loop-body-2
			2. No memory RAW violation across Loop-body-1 and Loop-body-2

Coalesing	for **%1** = **m1** to **n1** step **k1**:
	  for **%2** = **%1** to **n2** step **k2**:
	  Loop-body
		for **%3** = 0 to **n1** × **n2**:
		  %1=(floordiv **%3 n2**)
		  %2=(mod **%3 n2**)
		  Loop-body
			1. The reference of **%1**, **%2** is replaced by (floordiv **%3 n2**) and (mod **%3 n2**), respectively.

dataflow structures is challenging simply using static rewriting rules; (3) Loop unrolling introduces new variables, and their metadata (such as variable names and data types) is collected at runtime. For example, the loop variable %arg1 is created during the unrolling process in Listing 6. Since this runtime-collected information must be determined prior to compilation, creating static "unrolling" e-graph rewriting rules that accommodate all unrolling scenarios becomes impractical. In complex scenarios like loop unrolling, where the unrolling factor and variable metadata can vary dynamically based on input code, static rewriting rules prove insufficient. Therefore, dynamic rewriting rules must be customized to handle such variability. These challenges collectively prompt us to develop dynamic rewriting rules, generated at runtime, to effectively handle complex control flow transformations.

Dynamic Rewriting: To overcome the limitations of static rewriting, HEC generates dynamic rewriting rules at runtime, tailoring them to each unique code input to achieve high adaptability across diverse scenarios. In Step ②, the dynamic rule generator leverages the pre-defined transformation patterns to assess computation nodes for applicable transformations. Once suitable nodes are identified, HEC creates new dynamic rewriting rules based on the graph representation.

Using loop unrolling transformations as an example to demonstrate our verification approach, we establish the transformation criteria shown in Figure 5, which are used to determine whether a code segment qualifies as an unrolled variant. These criteria define the requirements for *for* loops to be recognized as equivalent to loop unrolling transformations. For instance, the loops in lines 4 and 15 of Listing 6 satisfy the conditions outlined in Figure 5, indicating their eligibility for the loop unrolling transformation pattern and confirming their equivalence to those in Listing 1. Additionally, we introduce

```
1 affine.for %arg1 = m1 to n2 step k2:
2   // Loop-body-2
```

Transformation Pattern Condition:

$\lceil(n2-m1)/k2\rceil == \lceil(n2-m2)/k2\rceil + \lceil(n1-m1)/k1\rceil \times (k1/k2)$

```
1 affine.for %arg1 = m1 to n2 step k2:
2   // Loop-body-2
```

Figure 5: Loop unrolling transformation pattern: the code segments at the top and bottom can be interchanged if they satisfy the required conditions in the middle.

a leading operator, *combine*, to bind the two loop operations (line 2 in Listing 8). Here, we assume that all integrated rewriting rules and transformation patterns are correct. Whether the input code meets the transformation pattern conditions is automatically verified using the Z3 SMT solver [20, 21].

HEC analyzes the graph representation illustrated in Section 4.1 to assess the eligibility of each `for` loop. For each eligible loop, HEC constructs a dynamic rule $lhs \Leftrightarrow rhs$. For instance, Listing 7 and Listing 8 show the *lhs* and *rhs* of dynamic rewriting rules in Listing 6 unrolling, respectively. A selection of dynamic rewriting patterns is shown in Table 2.

Similarly, Listing 4 is verified to be equivalent to Listing 1 through dynamic rewriting. Specifically, the *for* loop in line 4 of Listing 4 is identified as an eligible computation node for loop tiling, and customized dynamic rewriting rules for tiling are established to achieve equivalence with Listing 1.

Extensibility: HEC supports extensibility through a systematic approach for incorporating new transformation patterns. For control flow transformations, users must formalize the transformation pattern along with its correctness conditions (similar to those in Table 2), while datapath transformations require the corresponding static rewriting rules. To ensure soundness, HEC employs a dual validation strategy: static dat-

apath transformation rules are derived from mathematically proven algebraic identities such as De Morgan's laws and arithmetic associativity, making them sound by construction, while dynamic rules for control flow transformations undergo formal verification using the Z3 SMT solver to ensure mathematical correctness. For instance, the loop unrolling condition specified in Table 2 is mathematically verified to guarantee preservation of the iteration space, thereby maintaining semantic equivalence between the original and transformed code.

HEC limits Z3 usage to verifying dynamic rule pattern conditions rather than incorporating it into entire saturation process, thereby avoiding scalability bottlenecks. Our testing confirms that the dynamic rule generation, including Z3 checks, typically completes in under one second, a negligible fraction of the overall verification runtime, which is predominantly consumed by e-graph saturation.

4.3 HEC Verification Runner

```
%0 = arith.index_cast %arg0 : i32 to index
affine.for %arg2 = 0 to %0
  %1 = affine.load %arg1[%arg2] : memref<?xf64>
```

```
#map1 = affine_map<()[s0] -> (((s0 floordiv 2) floordiv 3) * 6)>
#map2 = affine_map<()[s0] -> ((s0 floordiv 2) * 2)>
#map3 = affine_map<()[s0] -> ((s0 floordiv 2) * 2 + ((s0 mod 2) floordiv 3) * 3)>

affine.for %arg2 = 0 to #map1()[%0] step 6
  %1 = affine.load %arg1[%arg2] : memref<?xf64>
  %2 = affine.apply affine_map<(d0) -> (d0 + 1)>(%arg2)
  %3 = affine.load %arg1[%2] : memref<?xf64>
  %4 = affine.apply affine_map<(d0) -> (d0 + 2)>(%arg2)
  %5 = affine.load %arg1[%4] : memref<?xf64>
  %6 = affine.apply affine_map<(d0) -> (d0 + 1)>(%4)
  %7 = affine.load %arg1[%6] : memref<?xf64>
  %8 = affine.apply affine_map<(d0) -> (d0 + 4)>(%arg2)
  %9 = affine.load %arg1[%8] : memref<?xf64>
  %10 = affine.apply affine_map<(d0) -> (d0 + 1)>(%8)
  %11 = affine.load %arg1[%10] : memref<?xf64>
affine.for %arg2 = #map1()[%0] to #map2()[%0] step 2
  %1 = affine.load %arg1[%arg2] : memref<?xf64>
  %2 = affine.apply affine_map<(d0) -> (d0 + 1)>(%arg2)
  %3 = affine.load %arg1[%2] : memref<?xf64>
affine.for %arg2 = #map2()[%0] to #map3()[%0] step 3
  %1 = affine.load %arg1[%arg2] : memref<?xf64>
  %2 = affine.apply affine_map<(d0) -> (d0 + 1)>(%arg2)
  %3 = affine.load %arg1[%2] : memref<?xf64>
  %4 = affine.apply affine_map<(d0) -> (d0 + 2)>(%arg2)
  %5 = affine.load %arg1[%4] : memref<?xf64>
affine.for %arg2 = #map3()[%0] to %0
  %1 = affine.load %arg1[%arg2] : memref<?xf64>
```

Figure 6: MLIR Nested Unrolling Case Study: The bottom MLIR code is produced by applying nested unrolling with factors of 2 and 3 to the top MLIR code.

In this section, we will elaborate more details about e-graph verification runner and use a specific example to illustrate the verification process.

Initial E-graph Construction: HEC takes the graph representation as input. It gathers functional information and constructs the corresponding e-graph. The algorithm for e-graph construction is detailed in Algorithm 1. Before construction, it builds the topological order for each operation based on the dependence relationship. Subsequently, HEC incrementally inserts operation nodes into the e-graph following the topological order of the graph nodes. Specifically, operations are inserted into the e-graph from the leaf nodes to the root nodes to ensure that child nodes are processed beforehand. Each insertion generates an identifier representing the e-nodes and connects these e-nodes with their respective input identifier.

Algorithm 1 E-graph Construction

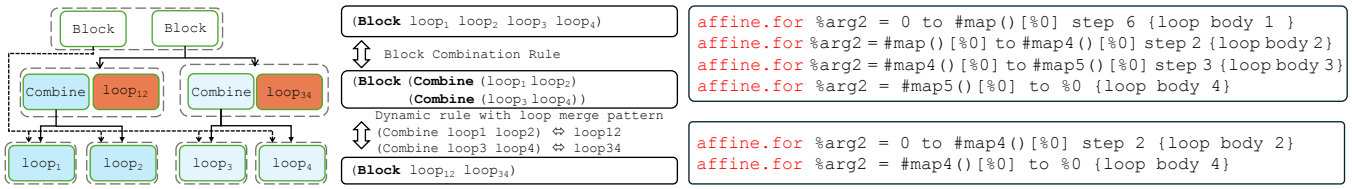
Input: Vertex list V from parsing netlist
Output: : E-graph G_e
1: Initialize *vmap* as empty HashMap
2: **for** each node v in TopoSort(V) **do** ▷ From leaf to root
3: $in_id \leftarrow []$
4: **for** each input i for node v **do**
5: in_id.pushback($vmap[i]$) ▷ All children are inserted
6: **end for**
7: $id \leftarrow G_e$.insert(v, in_id) ▷ Insert v to e-graph
8: $vmap[v] = id$
9: **end for**

HEC Verification: To demonstrate the verification procedure, we use the example shown in Figure 6. The bottom MLIR code presents a nested loop unrolled with factors of 2 and 3, derived from the top MLIR code. The verification process is depicted in Figure 7. For simplicity, the *for* loop structures in Figure 4 are simplified to single-loop operations.

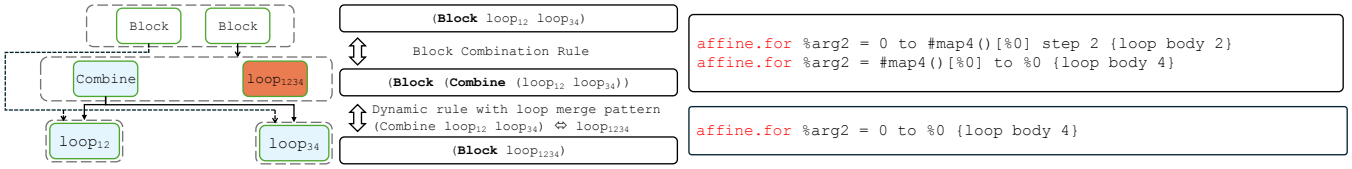
Initially, the ruleset includes only the predefined static rules without any dynamic rules. If the differences between the two MLIR codes fall within the scope addressable by static rewriting rules, the e-graph runner reports the codes as functionally equivalent. However, if the MLIR codes involve control flow transformations, the e-graph runner cannot affirm equivalence. In such cases, the graph representation is forwarded to the rule generator to apply the control flow transformation patterns.

Figure 7a illustrates the e-graph exploration during the initial iteration. The rule generator analyzes the graph representation and creates dynamic rules for candidate operations. HEC then creates a pseudo *combine* node to bind candidates *for* loop operations. The *combine* nodes are created automatically for each pair of candidate nodes of dynamic rules. Finally, HEC unifies the e-classes of *combine* node and the fused loop operations using dynamic rewriting rules. For example, the pairs of *for* loop structures from lines 5–17 and lines 21–27 in the unrolled MLIR code of Figure 6 (indicated by dotted lines in Figure 7a) are eligible for the loop unrolling transformation pattern. The e-graph runner creates a pseudo *combine* node as their parent e-node and builds dynamic rules to merge the *combine* node with $loop_{12}$ into the same e-class (indicated by solid lines). Consequently, $loop_1$/$loop_2$ and $loop_3$/$loop_4$ are merged into two loops named $loop_{12}$ and $loop_{34}$, respectively. The MLIR codes for both the original loops and the unrolled version are displayed on the right side of Figure 7a.

However, the e-graph runner is still unable to ascertain the equivalence between the two input codes. An additional iteration is necessary for confirmation. Similar to the approach in



(a) In the first iteration, HEC inserted pseudo *Combine* nodes to bind leaf $loop_1/loop_2$ and $loop_3/loop_4$. Then, the original sub-expression, indicated by a dotted line, is transformed into a new expression, represented by a solid line. Following this, the e-graph runner implements dynamic rules for $loop_1$/$loop_2$ and $loop_3$/$loop_4$ (highlighted by light blue nodes), merging them into $loop_{12}$ and $loop_{34}$ (illustrated with dark red nodes). However, HEC cannot verify the equivalence in the first iteration.



(b) In the second iteration, the rule generator introduces a new rule, merging $loop_{12}$ and $loop_{34}$ into $loop_{1234}$, similar to the procedure in the initial iteration. Subsequently, the e-graph runner validates the equivalence between the two input MLIR codes.

Figure 7: E-graph verification for the motivating example provided in Figure 6. The e-graph operates through two iterations, with the rule generator introducing dynamic rules to the e-graph runner during each iteration. Each *loop* operator represents a *for* loop structure, as depicted in Figure 4, for brevity purposes.

the first iteration, the rule generator proposes a new dynamic rule for merging $loop_{12}$ with $loop_{34}$ into a single $loop_{1234}$. The illustration of the second iteration is shown in Figure 7b. Consequently, the two MLIR codes in Figure 6 are unified into the same e-class. It is worth noting that Figure 7b shows only one possible verification path for clarity. In practice, the e-graph simultaneously explores multiple combination patterns, such as merging $loop_2$ and $loop_3$ into $loop_{23}$, which could lead to alternative paths like $loop_1$, $loop_2$, $loop_3$, $loop_4$ $\rightarrow$ $loop_1$, $loop_{23}$, $loop_4$ $\rightarrow$ $loop_{123}$, $loop_4$ $\rightarrow$ $loop_{1234}$. The e-graph's congruence closure property ensures that all valid transformation sequences leading to equivalent programs are automatically discovered and unified within the same e-class. If the e-graph runner still cannot determine the equivalence and the rule generator fails to create new dynamic rules for further evaluation, HEC will conclude that the input MLIR codes are not equivalent. It is important to note that both the e-graph verification runner and the rule generator utilize the graph representation as their input. During each iteration, HEC is equipped with an inverter, which converts the e-graph back into the graph representation to facilitate further processing in the subsequent iteration. Besides, the verification flow is fully automated. HEC offers an interface for adding control flow transformation patterns, facilitating broader equivalence verification through an expanded set of rewriting rules and verification patterns.

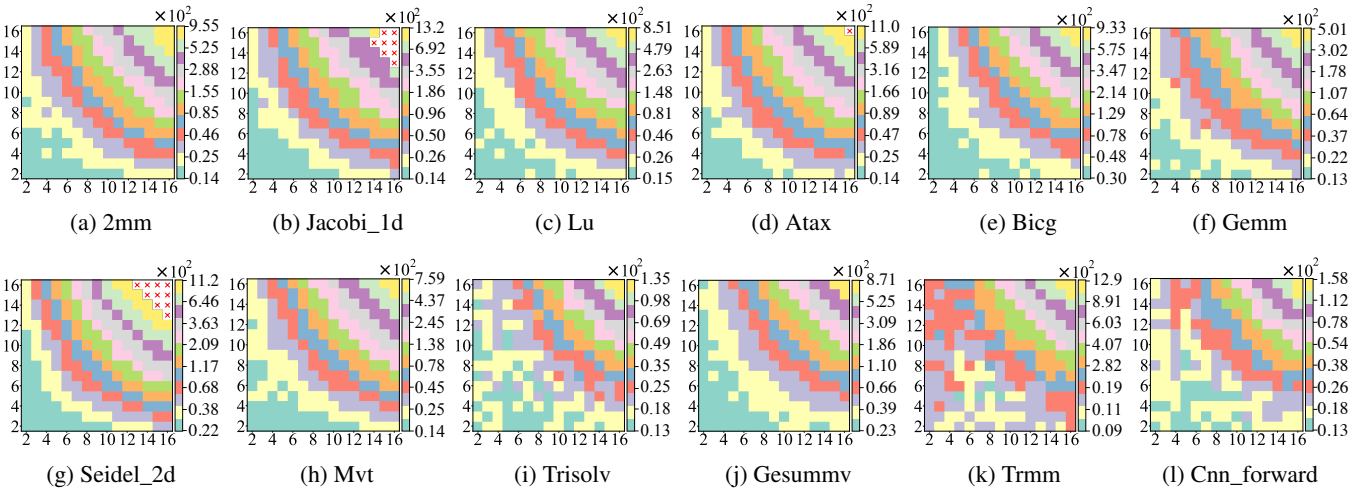
Completeness and Soundness: HEC, like any verification system based on rewriting rules, is inherently incomplete with respect to the universe of all possible code transformations. If an equivalence exists but requires a transformation rule not included in our framework, or if the saturation process is terminated due to computational limits before full exploration is complete, HEC may fail to establish equivalence even when it exists, resulting in false negatives (flagging equivalent programs as nonequivalent).

HEC guarantees soundness through a dual validation mechanism that ensures it will never produce false positives (declaring two programs equivalent when they are not). For static datapath transformation rules, soundness is achieved by deriving rules from mathematically proven algebraic identities such as De Morgan's laws and arithmetic associativity, making them sound by construction. For dynamic control flow transformation rules, pattern conditions are formally verified using the Z3 SMT solver to ensure mathematical correctness.

5 Evaluation

The experiments were conducted on a system equipped with an Intel(R) Xeon(R) Gold 6418H CPU. This CPU has 48 physical cores, operates at a maximum speed of 4000 MHz, and the system has 1024 GB of RAM. We implemented transformations on selected benchmarks using the official MLIR compiler, `mlir-opt`, with LLVM version 18.0.0.

There are many existing transformation validation tools in the literature, including CompCert [30, 31], Alive/Alive2 [33, 34], PolyCheck [7], and MLIR-TV [6], each providing verification capabilities at different levels of the compilation stack. CompCert focuses on C-level compiler correctness, Alive/Alive2 target LLVM IR optimizations, PolyCheck handles affine code transformations for C-level compilation, while MLIR-TV is designed for MLIR-level verification. Among these tools, only MLIR-TV provides verification at



(a) 2mm (b) Jacobi_1d (c) Lu (d) Atax (e) Bicg (f) Gemm

(g) Seidel_2d (h) Mvt (i) Trisolv (j) Gesummv (k) Trmm (l) Cnn_forward

Figure 8: Heatmap of Runtime for Different Nested Unrolling Configurations. The x-axis and y-axis represent the unrolling factors, while the colorbar on the right illustrates the end-to-end verification runtime (seconds).

the MLIR level, which is most relevant to our work. However, according to our testing, MLIR-TV fails to support the affine dialect and lacks coverage for affine operations that are central to high-level synthesis and the transformations evaluated in our benchmarks. This limitation renders a direct comparison with HEC is challenging.

Table 3: List of benchmarks.

Benchmark	Description	Complexity
GEMM [40]	General Matrix Multiply	$O(n^3)$
LU [40]	LU Decomposition	$O(n^3)$
2MM [40]	Two Matrix Multiplications	$O(n^3)$
ATAX [40]	Matrix Transpose Vector Multiplication	$O(n^2)$
BiCG [40]	Biconjugate Gradient Method	$O(n^2)$
GESUMMV [40]	SUM of Matrix Vector Multiplications	$O(n^2)$
MVT [40]	Matrix Vector Transpose	$O(n^2)$
TRISOLV [40]	Triangular Solver	$O(n^2)$
TRMM [40]	Triangular Matrix Multiply	$O(n^3)$
CNN_Forward [50]	CNN Forward Function	$O(n^7)$
Jacobi 1D [40]	Jacobi 1D iterative method	$O(n \cdot t)$
Seidel 2D [40]	Gauss-Seidel method	$O(n^2 \cdot t)$

5.1 Control Flow Transformation Verification

We evaluate the performance of HEC using the loop unrolling and tiling transformations on the MLIR benchmark. Specifically, we verify the equivalence between the input code within the selected benchmarks, reporting the end-to-end runtime, the number of dynamic rules implemented, and the number of e-classes. The benchmarks used in this evaluation are selected from Polybench [40] and are enhanced with an additional benchmark, CNN-Forward, from Polybench-NN [50]. We utilize Polygeist [35] to convert the C code into MLIR, focusing specifically on the kernel portions for evaluation. These benchmarks are implemented as low-level kernels to facilitate hardware acceleration. The benchmarks, including description and time complexity, are shown in Table 3.

Comprehensive results are provided in Table 4, where all selected results have been successfully verified. The benchmarks for Seidel_2d and Jacobi_1d exhibit a specific type of error, classified as a **Loop Boundary Check Error**, which will be discussed in detail in Section 5.4.

First, HEC consistently verifies the correctness of transformations within one minute for most cases, except for nested unrolling. Second, the tiling transformation, which primarily adjusts loop parameters without altering code size, shows stable results across various tiling factors (represented as T2 to T64). Additionally, the number of e-classes has a significant impact on verification runtime, particularly in cases of nested unrolling. In these cases, the size of the e-graph predominantly determines the duration of verification.

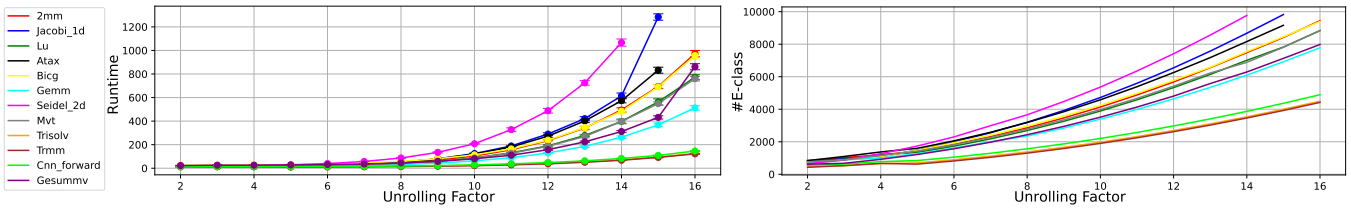
5.2 Unrolling Verification Runtime

Since the unrolling transformation increases code size, unlike loop tiling, we specifically evaluate the verification runtime for nested unrolling as a scalability test. We assess the runtime performance of HEC across various nested unrolling factors. The results are illustrated in Figure 8, where each sub-graph is a heatmap. The x and y axes of the heatmap represent unrolling factors from 2 to 16. The color of each pixel indicates the verification runtime by seconds, as detailed by the colorbar on the right. Cases marked with an "X" indicate that they either exceeded the time limit.

Figure 8 shows that the verification runtime for most benchmarks completes within 20 minutes. E-graph saturation dominates the runtime, accounting for over 90% in runtime.

Table 4: Runtime, dynamic rule numbers, e-classes numbers comparison under various tiling (T), unrolling setup (U) configurations. For benchmark Jacobi_1d and Seidel_2d, mlir-opt tool generates the Loop Boundary errors, which will be elaborated in Section 5.4. Note that these are standard benchmarks used widely in MLIR-based compilation evaluations.

	Equivalence Checking	Metric	Base	T2~64	U8	U16	U32	U64	T16-U8	U16-T8	U8-U4	U16-U8
2MM	✓	Runtime(s)	N/A	7.3	6.8	7.8	10.9	25.3	6.8	7.9	27.8	173.8
		#Dynamic Rules	N/A	2	2	2	2	2	4	4	6	6
		#E-classes	156	198	427	683	1195	2219	469	725	1733	5069
Jacobi_1d	**Loop Boundary Bug Identified**	Runtime(s)	N/A	6.9	6.9	7.5	11.2	28.9	6.7	7.8	22.4	221.3
		#Dynamic Rules	N/A	1	2	2	2	2	3	3	6	6
		#E-classes	111	121	428	732	1340	2556	438	742	1832	5936
Lu	✓	Runtime(s)	N/A	6.9	6.7	6.9	9.6	21.4	6.7	7.3	23.5	155.1
		#Dynamic Rules	N/A	1	2	2	2	2	3	3	6	6
		#E-classes	110	119	364	604	1084	2044	373	613	1642	4718
Atax	✓	Runtime(s)	N/A	6.8	9.5	7.2	10.6	27.4	12.3	7.9	30.4	215.0
		#Dynamic Rules	N/A	2	3	3	3	3	6	4	9	9
		#E-classes	109	128	557	692	1252	2372	584	623	1930	5599
Bicg	✓	Runtime(s)	N/A	22.2	20.1	21.8	24.0	37.8	25.8	25.9	40.6	191.3
		#Dynamic Rules	N/A	2	2	2	2	2	5	3	6	6
		#E-classes	139	158	513	666	1178	2202	540	597	1735	5143
Gemm	✓	Runtime(s)	N/A	6.8	6.9	6.8	8.5	17.3	13.2	7.7	20.9	101.5
		#Dynamic Rules	N/A	1	2	2	2	2	3	3	6	6
		#E-classes	90	100	308	516	932	1764	318	526	1415	4078
Seidel_2d	**Loop Boundary Bug Identified**	Runtime(s)	N/A	14.5	15.8	16.7	22.0	51.0	14.7	13.3	36.5	380.1
		#Dynamic Rules	N/A	1	1	1	1	1	2	1	3	3
		#E-classes	214	246	563	907	1595	3017	584	246	2125	6723
Mvt	✓	Runtime(s)	N/A	6.8	6.3	6.9	9.2	20.6	6.1	6.0	25.7	144.7
		#Dynamic Rules	N/A	2	2	2	2	2	4	2	6	6
		#E-classes	91	131	339	579	1059	2019	356	131	1653	4683
Trisolv	✓	Runtime(s)	N/A	6.8	7.2	6.3	7.0	9.3	6.2	6.7	17.7	36.0
		#Dynamic Rules	N/A	1	1	1	1	1	2	2	3	3
		#E-classes	82	92	214	334	574	1054	224	344	900	2400
Gesummv	✓	Runtime(s)	N/A	19.5	19.1	19.3	21.4	31.1	17.8	18.3	30.3	131.5
		#Dynamic Rules	N/A	1	1	1	1	1	2	2	3	3
		#E-classes	172	181	402	618	1050	1914	411	627	1404	4372
Trmm	✓	Runtime(s)	N/A	6.7	5.8	6.2	6.6	9.2	6.2	8.4	16.4	31.6
		#Dynamic Rules	N/A	1	1	1	1	1	2	2	3	3
		#E-classes	72	93	201	321	561	1041	222	342	896	2381
CNN_forward	✓	Runtime(s)	N/A	8.1	7.4	7.5	8.6	11.5	7.1	6.8	21.0	45.3
		#Dynamic Rules	N/A	1	1	1	1	1	2	1	3	3
		#E-classes	140	222	273	401	657	1169	343	222	983	2591



(a) Runtime

(b) Number of e-class

Figure 9: Runtime and e-graph complexity w.r.t nested unrolling complexity under the same unrolling factors.

Additionally, we present detailed results in Figure 9 for the diagonal data sample from Figure 8. In this figure, the x-axis denotes the nested unrolling data sample (specifically, the unroll_k_unroll_k, where k is the unrolling factor), and the y-axis displays the runtime and the number of e-classes.

This runtime is exponential rather than linear. Firstly, unrolling transformations modify code size, introducing scalability challenges, which are not present in other transformations like tiling and loop fusion. Secondly, the diagonal data sample involves nested unrolling, where the code size increases quadratically with each increase in the unrolling factor. This quadratic growth in code size significantly impacts runtime. Notably, it is uncommon for implementations to adopt such large unrolling factors due to the substantial increase in chip area it entails. From the perspective of power, performance, and area (PPA), such high unrolling factors are generally avoided, making the case of nested unrolling particularly rare.

5.3 Datapath Transformation Evaluation

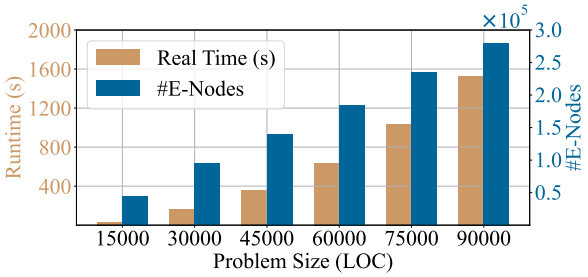


Figure 10: Runtime and e-nodes number for HEC across representative problem sizes ranging from 15,000 to 90,000 LOC.

HEC is designed to manage both static and dynamic transformations. In this section, we focus on the datapath transformation capabilities of HEC. Specifically, we generated over 150 benchmarks incorporating various datapath transformations. These benchmarks vary in size, measured by lines of code (LOC), to assess the framework scalability for datapath transformation. We conducted experiments to measure both the runtime and the number of e-nodes generated for different benchmark sizes. The results are illustrated in Figure 10. In this figure, the x-axis represents the problem size measured in the sum of LOC for input benchmark, with representative datapoints ranging from 15,000 to 90,000, while the y-axis depicts the runtime (left) and the number of e-nodes (right).

Notably, all verification tasks were successfully completed within a 40-minute timeframe. For instance, the largest benchmark, encompassing 108,012 LOC, required a verification runtime of 2,305 seconds. This indicates that HEC maintains consistent performance even as the problem size increases significantly. Additionally, the analysis of the number of e-nodes reveals a linear growth trend with respect to LOC, as illustrated in Figure 10. This linear relationship suggests that the complexity of the e-graph structure scales predictably with the size of the input code, ensuring manageable computational overhead. Together, these findings highlight the capability of HEC to efficiently verify large-scale benchmarks while maintaining structural integrity and performance.

5.4 Mining Bugs in PolyBenchC MLIR

The MLIR affine dialect [1] provides a suite of standard optimization transformations. We selected the representative kernels to showcase the application of these transformations. We will highlight two representative cases where the optimized HLS programs fail to preserve semantic equivalence with the original programs, and our tool has successfully detected and reported these discrepancies as bugs. In the following sections, we will elaborate on these two cases as case studies.

Case study 1: Loop Boundary Check Error: We begin with an example in Listing 9 and its unrolled version (Listing 10), which is unrolled by a factor of 2 using `mlir-opt`. The two code segments appear equivalent. The first loop in Listing 10 from lines 9 to 13 handles the majority of the operations, while the second loop from lines 16 to 18 processes the remainder. Together, both loops make up the entire code shown in Listing 9. However, this transformation contains a flaw in the loop boundary checks. Specifically, if we consider the scenario where `%0` equals 5, Listing 9 will not execute since the loop start value (15) exceeds its end value (10). In contrast, Listing 10 executes once for the second loop spanning from line 16 to line 19. This error occurs with all values of `%0` less than 10.

Our tests indicate that these bugs are not confined to the specific case study presented. In general, for any loop defined with start and end conditions, `mlir-opt` preserves semantic equivalence only when the loop end value is guaranteed to exceed the loop start value. If this guarantee is not met and the loop is unrolled by a factor of k, the transformed loop may inadvertently alter the loop boundaries, causing unintended iterations. For instance, such errors are observed in the

[1] https://mlir.llvm.org/docs/Dialects/Affine/

```
#map = affine_map<(d0) -> (d0 + 10)>
#map1 = affine_map<(d0) -> (d0 * 2)>
func.func @kernel(%arg0: i32, %arg1: memref<?xf64>) {
  %0 = arith.index_cast %arg0 : i32 to index
  // Loop range: from %0+10 to %0*2
  // Program should not execute when %0 is 5 (15->10)
  affine.for %arg2 = #map(%0) to #map1(%0) {
    %1 = affine.load %arg1[%arg2] : memref<?xf64>
  }
  return
}
```

Listing 9: Case study 1

```
#map = affine_map<(d0) -> (d0 + 10)>
#map1 = affine_map<()[s0] -> (s0 + (s0 floordiv 2) * 2)>
#map2 = affine_map<(d0) -> (d0 + 1)>
#map3 = affine_map<(d0) -> (d0 * 2)>
func.func @kernel(%arg0: i32, %arg1: memref<?xf64>) {
  %0 = arith.index_cast %arg0 : i32 to index
  // Loop range: from %0+10 to %0+(%0 floordiv 2)*2
  // This loop should not execute when %0 is 5 (15->9)
  affine.for %arg2 = #map(%0) to #map1()[%0] step 2 {
    %1 = affine.load %arg1[%arg2] : memref<?xf64>
    %2 = affine.apply #map2(%arg2)
    %3 = affine.load %arg1[%2] : memref<?xf64>
  }
  // Loop range: from %0+(%0 floordiv 2)*2 to %0*2
  // This loop will execute once when %0 is 5 (9->10)
  affine.for %arg2 = #map1()[%0] to #map3(%0) {
    %1 = affine.load %arg1[%arg2] : memref<?xf64>
  }
  return
}
```

Listing 10: Loop boundary check error: Loop unrolling

```
func.func @testing2(%arg0: memref<10xi32>, %arg1: memref
    <10xi32>) {
  %cst = arith.constant 1 : i32
  // Replace elements indexed from 1 to 10 with %arg0[0]
  affine.for %arg2 = 1 to 10 {
    %1 = affine.load %arg0[%arg2 - 1] : memref<10xi32>
    affine.store %1, %arg0[%arg2] : memref<10xi32>
  }
  // Elements indexed from 1 to 10 are incremented by 1
  affine.for %arg2 = 1 to 10 {
    %1 = affine.load %arg0[%arg2] : memref<10xi32>
    %2 = arith.addi %1, %cst : i32
    affine.store %2, %arg0[%arg2] : memref<10xi32>
  }
  // Finally, memory in %arg0 will be replaced to
  // %arg0[0], %arg0[0]+1, %arg0[0]+1, ...
  return
}
```

Listing 11: Case study 2

```
func.func @testing2(%arg0: memref<10xi32>, %arg1: memref
    <10xi32>) {
  %cst = arith.constant 1 : i32
  affine.for %arg2 = 1 to 10 {
    // Load data from %arg0[%arg2 - 1]
    %0 = affine.load %arg0[%arg2 - 1] : memref<10xi32>
    affine.store %0, %arg0[%arg2] : memref<10xi32>
    %1 = affine.load %arg0[%arg2] : memref<10xi32>
    %2 = arith.addi %1, %cst : i32
    // Elements are incremented by 1
    affine.store %2, %arg0[%arg2] : memref<10xi32>
  }
  // Finally, memory in %arg0 will be replaced to
  // %arg0[0], %arg0[0]+1, %arg0[0]+2, ...
  return
}
```

Listing 12: Memory RAW violation: Loop fusion

Jacobi_1d and Seidel_2d benchmarks listed in Table 3.

Case study 2: Memory Read-After-Write Violation: Listing 11 and Listing 12 illustrate another instance of semantic non-equivalence following a transformation—loop fusion. In the provided code, the divergence in memory state between the two functions arises from differences in the execution order of the loop body operations after loop fusion. In the first function, two separate loops handle the operations: the first loop copies values from one element to the next, and the second loop increments these values. As a result, the final state of the array consists of the original element `%arg0[0]` followed by incremented versions of this element for all subsequent indices. Specifically, the array becomes `%arg0[0]`, `%arg0[0] + 1`, `%arg0[0] + 1`, ..., `%arg0[0] + 1`, with every element after the first set to `%arg0[0] + 1`. However, in the fused version, each iteration of the single loop performs both copy and increment operations adjacently. This introduces a dependency violation on the increment operation of the previous iteration, leading to a linearly increasing sequence based on the first element. Finally, the array becomes `%arg0[0]`, `%arg0[0] + 1`, `%arg0[0] + 2`, ..., `%arg0[0] + 10`. Such transformations in the loop structure affect not only the computational dependencies but also the overall memory state of `%arg0`, resulting in non-equivalence of the final output across the two function versions.

In these case studies, we identify two types of transformation errors introduced by the `mlir-opt` compiler. Code unrolling leads to additional loop body executions, while loop fusion transformation changes the sequence of data dependencies, both affecting the intended behavior and memory state. Relying on `mlir-opt` for code transformations without subsequent equivalence checks can result in various adverse outcomes. Minor issues might include data corruption, while more severe implications could lead to complete system failures, potentially disrupting critical operations and compromising system integrity. Although code transformations can enhance performance, the improper application of these transformations without equivalence checking can introduce errors that disrupt control flow and alter semantic behavior.

6 Conclusions

This paper introduces HEC, a novel equivalence checking tool. HEC leverages the dynamic rule generation to effectively verify complex control flow transformations. By integrating both static datapath rewriting rules with dynamic, code-specific rules, HEC provides a flexible verification solution. Our experimental results demonstrate its efficiency, verifying various code transformations within an efficient runtime. Finally, HEC successfully identifies a few critical compilation errors, covering functional incorrectness and data hazard violations.

Acknowledgment – This work is supported by National Science Foundation (NSF) under CCF2350186, CCF2403134, CCF2349670, and CCF2349461 awards and the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research's Advanced Computing Technologies Competitive Portfolios program at Pacific Northwest National Laboratory (PNNL).

References

[1] IREE, September 2019.

[2] ISA. https://repo.or.cz/w/isa.git, 2024.

[3] Nicolas Bohm Agostini, Serena Curzel, David Kaeli, and Antonino Tumeo. Soda-opt an mlir based flow for co-design and high-level synthesis. In *Proceedings of the 19th ACM International Conference on Computing Frontiers*, pages 201–202, 2022.

[4] Leo Bachmair and Harald Ganzinger. Rewrite-based equational theorem proving with selection and simplification. *Journal of Logic and Computation*, 4(3):217–247, 1994.

[5] Leo Bachmair and Harald Ganzinger. Equational reasoning in saturation-based theorem proving. *Automated deduction—a basis for applications*, 1:353–397, 1998.

[6] Seongwon Bang, Seunghyeon Nam, Inwhan Chun, Ho Young Jhoo, and Juneyoung Lee. Smt-based translation validation for machine learning compiler. In *International Conference on Computer Aided Verification*, pages 386–407. Springer, 2022.

[7] Wenlei Bao, Sriram Krishnamoorthy, Louis-Noël Pouchet, Fabrice Rastello, and Ponnuswamy Sadayappan. Polycheck: Dynamic verification of iteration space transformations on affine programs. *ACM SIGPLAN Notices*, 51(1):539–554, 2016.

[8] David Cao, Rose Kunkel, Chandrakana Nandi, Max Willsey, Zachary Tatlock, and Nadia Polikarpova. Babble: Learning better abstractions with e-graphs and anti-unification. *Proceedings of the ACM on Programming Languages*, 7(POPL):396–424, 2023.

[9] Chen Chen, Guangyu HU, Cunxi Yu, Yuzhe Ma, and Hongce Zhang. E-morphic: Scalable equality saturation for structural exploration in logic˜ synthesis. *Design Automation Conference (DAC)*, 2025.

[10] Chen Chen, Guangyu Hu, Dongsheng Zuo, Cunxi Yu, Yuzhe Ma, and Hongce Zhang. E-syn: E-graph rewriting with technology-aware cost functions for logic synthesis. pages 1–6, 2024.

[11] Jianyi Cheng, Samuel Coward, Lorenzo Chelini, Rafael Barbalho, and Theo Drane. Seer: Super-optimization explorer for hls using e-graph rewriting with mlir. *arXiv preprint arXiv:2308.07654*, 2023.

[12] Jianyi Cheng, Samuel Coward, Lorenzo Chelini, Rafael Barbalho, and Theo Drane. Seer: Super-optimization explorer for high-level synthesis using e-graph rewriting. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 1029–1044, 2024.

[13] Circuit ir compilers and tools (CIRCT)., 2024.

[14] Jason Cong, Jason Lau, Gai Liu, Stephen Neuendorffer, Peichen Pan, Kees Vissers, and Zhiru Zhang. Fpga hls today: successes, challenges, and opportunities. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 15(4):1–42, 2022.

[15] Jason Cong, Bin Liu, Stephen Neuendorffer, Juanjo Noguera, Kees Vissers, and Zhiru Zhang. High-level synthesis for fpgas: From prototyping to deployment. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 30(4):473–491, 2011.

[16] Samuel Coward, George A Constantinides, and Theo Drane. Automatic datapath optimization using e-graphs. In *2022 IEEE 29th Symposium on Computer Arithmetic (ARITH)*, pages 43–50. IEEE, 2022.

[17] Samuel Coward, George A Constantinides, and Theo Drane. Automating constraint-aware datapath optimization using e-graphs. In *2023 60th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2023.

[18] Samuel Coward, Emiliano Morini, Bryan Tan, Theo Drane, and George A Constantinides. Datapath verification via word-level e-graph rewriting. In *2023 Formal Methods in Computer-Aided Design (FMCAD)*, pages 92–100. IEEE, 2023.

[19] Shail Dave, Youngbin Kim, Sasikanth Avancha, Kyoungwoo Lee, and Aviral Shrivastava. Dmazerunner: Executing perfectly nested loops on dataflow accelerators. *ACM Transactions on Embedded Computing Systems (TECS)*, 18(5s):1–27, 2019.

[20] Leonardo De Moura and Nikolaj Bjørner. Efficient e-matching for smt solvers. In *Automated Deduction–CADE-21: 21st International Conference on Automated Deduction Bremen, Germany, July 17-20, 2007 Proceedings 21*, pages 183–198. Springer, 2007.

[21] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[22] Nachum Dershowitz. A taste of rewrite systems. *Functional Programming, Concurrency, Simulation and Automated Reasoning: International Lecture Series 1991–1992 McMaster University, Hamilton, Ontario, Canada*, pages 199–228, 2005.

[23] David Detlefs, Greg Nelson, and James B Saxe. Simplify: a theorem prover for program checking. *Journal of the ACM (JACM)*, 52(3):365–473, 2005.

[24] Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. Taso: optimizing deep learning computation with automatic generation of graph substitutions. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 47–62, 2019.

[25] Chandan Karfa, Kunal Banerjee, Dipankar Sarkar, and Chittaranjan Mandal. Verification of loop and arithmetic transformations of array-intensive behaviors. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 32(11):1787–1800, 2013.

[26] Yi-Hsiang Lai, Yuze Chi, Yuwei Hu, Jie Wang, Cody Hao Yu, Yuan Zhou, Jason Cong, and Zhiru Zhang. Heterocl: A multi-paradigm programming infrastructure for software-defined reconfigurable computing. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 242–251, 2019.

[27] Yi-Hsiang Lai, Ecenur Ustun, Shaojie Xiang, Zhenman Fang, Hongbo Rong, and Zhiru Zhang. Programming and synthesis for software-defined fpga acceleration: status and future prospects. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 14(4):1–39, 2021.

[28] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *International symposium on code generation and optimization, 2004. CGO 2004.*, pages 75–86. IEEE, 2004.

[29] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. Mlir: Scaling compiler infrastructure for domain specific computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 2–14. IEEE, 2021.

[30] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

[31] Xavier Leroy, Sandrine Blazy, Daniel Kästner, Bernhard Schommer, Markus Pister, and Christian Ferdinand. Compcert-a formally verified optimizing compiler. In *ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress*, 2016.

[32] Bruce W Leverett, Roderic Geoffrey Galton Cattell, Steven O Hobbs, Joseph M Newcomer, Andrew Henry Reiner, Bruce R Schatz, and William A Wulf. An overview of the production quality compiler-compiler project. *Computer*, 13(8):38–49, 1980.

[33] Nuno P Lopes, Juneyoung Lee, Chung-Kil Hur, Zhengyang Liu, and John Regehr. Alive2: bounded translation validation for llvm. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 65–79, 2021.

[34] Nuno P Lopes, David Menendez, Santosh Nagarakatte, and John Regehr. Provably correct peephole optimizations with alive. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 22–32, 2015.

[35] William S Moses, Lorenzo Chelini, Ruizhe Zhao, and Oleksandr Zinenko. Polygeist: Raising c to polyhedral mlir. In *2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 45–59. IEEE, 2021.

[36] Charles Gregory Nelson. *Techniques for program verification*. Stanford University, 1980.

[37] Greg Nelson and Derek C Oppen. Fast decision procedures based on congruence closure. *Journal of the ACM (JACM)*, 27(2):356–364, 1980.

[38] Robert Nieuwenhuis and Albert Oliveras. Proof-producing congruence closure. In *International Conference on Rewriting Techniques and Applications*, pages 453–468. Springer, 2005.

[39] Pavel Panchekha, Alex Sanchez-Stern, James R Wilcox, and Zachary Tatlock. Automatically improving accuracy for floating point expressions. *Acm Sigplan Notices*, 50(6):1–11, 2015.

[40] Louis-Noël Pouchet et al. Polybench: The polyhedral benchmark suite. https://github.com/MatthiasJReisinger/PolyBenchC-4.2.1, 437:1–1, 2012.

[41] Louis-Noël Pouchet, Emily Tucker, Niansong Zhang, Hongzheng Chen, Debjit Pal, Gabriel Rodríguez, and Zhiru Zhang. Formal verification of source-to-source transformations for hls. In *Proceedings of the 2024 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 97–107, 2024.

[42] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *Acm Sigplan Notices*, 48(6):519–530, 2013.

[43] KC Shashidhar, Maurice Bruynooghe, Francky Catthoor, and Gerda Janssens. Verification of source code transformations by program equivalence checking. In *Compiler Construction: 14th International Conference, CC 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005. Proceedings 14*, pages 221–236. Springer, 2005.

[44] Gus Henry Smith, Andrew Liu, Steven Lyubomirsky, Scott Davidson, Joseph McMahan, Michael Taylor, Luis Ceze, and Zachary Tatlock. Pure tensor program rewriting via access patterns (representation pearl). In *Proceedings of the 5th ACM SIGPLAN International Symposium on Machine Programming*, PLDI '21. ACM, June 2021.

[45] Atefeh Sohrabizadeh, Cody Hao Yu, Min Gao, and Jason Cong. Autodse: Enabling software programmers to design efficient fpga accelerators. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 27(4):1–27, 2022.

[46] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. Equality saturation: a new approach to optimization. In *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 264–276, 2009.

[47] Philippe Tillet, Hsiang-Tsung Kung, and David Cox. Triton: an intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, pages 10–19, 2019.

[48] Ecenur Ustun, Ismail San, Jiaqi Yin, Cunxi Yu, and Zhiru Zhang. Impress: Large integer multiplication expression rewriting for fpga hls. In *2022 IEEE 30th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 1–10. IEEE, 2022.

[49] Ecenur Ustun, Cunxi Yu, and Zhiru Zhang. Equality saturation for datapath synthesis: A pathway to pareto optimality. In *2023 60th ACM/IEEE Design Automation Conference (DAC)*, pages 1–2. IEEE, 2023.

[50] Hrishikesh Vaidya, Akilesh B, Abhishek Patwardhan, and Ramakrishna Upadrasta. Polybench-nn. https://github.com/IITH-Compilers/PolyBench-NN, 2018.

[51] Alexa VanHattum, Rachit Nigam, Vincent T Lee, James Bornholt, and Adrian Sampson. Vectorization for digital signal processors via equality saturation. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 874–886, 2021.

[52] Sven Verdoolaege, Gerda Janssens, and Maurice Bruynooghe. Equivalence checking of static affine programs using widening to handle recurrences. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 34(3):1–35, 2012.

[53] Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. Egg: Fast and extensible equality saturation. *Proceedings of the ACM on Programming Languages*, 5(POPL):1–29, 2021.

[54] Jackson Woodruff, Thomas Koehler, Alexander Brauckmann, Chris Cummins, Sam Ainsworth, and Michael FP O'Boyle. Rewriting history: Repurposing domain-specific cgras. *arXiv preprint arXiv:2309.09112*, 2023.

[55] Nan Wu, Yuan Xie, and Cong Hao. Ironman: Gnn-assisted design space exploration in high-level synthesis via reinforcement learning. In *Proceedings of the 2021 on Great Lakes Symposium on VLSI*, pages 39–44, 2021.

[56] Yichen Yang, Phitchaya Phothilimthana, Yisu Wang, Max Willsey, Sudip Roy, and Jacques Pienaar. Equality saturation for tensor graph superoptimization. *Proceedings of Machine Learning and Systems*, 3:255–268, 2021.

[57] Hanchen Ye, Cong Hao, Jianyi Cheng, Hyunmin Jeong, Jack Huang, Stephen Neuendorffer, and Deming Chen. Scalehls: A new scalable high-level synthesis framework on multi-level intermediate representation. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 741–755. IEEE, 2022.

[58] Jiaqi Yin, Zhan Song, Chen Chen, Qihao Hu, and Cunxi Yu. Boole: Exact symbolic reasoning via boolean equality saturation. *Design Automation Conference (DAC)*, 2025.

[59] Cunxi Yu, Mihir Choudhury, Andrew Sullivan, and Maciej Ciesielski. Advanced datapath synthesis using graph isomorphism. In *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 424–429. IEEE, 2017.

[60] Yihong Zhang, Yisu Remy Wang, Oliver Flatt, David Cao, Philip Zucker, Eli Rosenthal, Zachary Tatlock, and Max Willsey. Better together: Unifying datalog and equality saturation. *Proc. ACM Program. Lang.*, 7(PLDI), jun 2023.

[61] Ruizhe Zhao, Jianyi Cheng, Wayne Luk, and George A Constantinides. Polsca: Polyhedral high-level synthesis with compiler transformations. In *2022 32nd International Conference on Field-Programmable Logic and Applications (FPL)*, pages 235–242. IEEE, 2022.

Para-ksm: Parallelized Memory Deduplication with Data Streaming Accelerator

Houxiang Ji[Ψ] Minho Kim[‡] Seonmu Oh[‡] Daehoon Kim[*] Nam Sung Kim[Ψ]
[Ψ]*University of Illinois Urbana-Champaign*
[‡]*Daegu Gyeongbuk Institute of Science and Technology*
[*]*Yonsei University*

Abstract

To tame the rapidly rising cost of memory in servers, hyperscalers have begun deploying memory deduplication features, such as Kernel Same-page Merging (`ksm`), for some of their services. Nonetheless, `ksm` incurs a datacenter tax significant enough to notably degrade performance of co-running applications, which hinders its wider and more aggressive deployment. Meanwhile, the server-class CPU has started to integrate various on-chip accelerators to effectively reduce datacenter taxes. One of such accelerators is Data Streaming Accelerator (DSA), which can offload the two most taxing functions of `ksm`, page comparison and checksum computation, from CPU. In this work, we demonstrate that `ksm` offloading these two functions to DSA (`DSA-ksm`) can reduce the performance degradation of co-running applications caused by `ksm` from 1.6–5.8× to 1.0–1.6×. However, we uncover that `DSA-ksm`, which naïvely replaces CPU-based functions with their DSA-based counterparts, yields significantly lower rates of memory deduplication than `ksm` due to the long latency of offloading these functions through on-chip PCIe. To address this shortcoming, we redesign `ksm` to exploit DSA's batching capability (`Para-ksm`). It facilitates a given function to operate on multiple pages per offload, rather than a single page as `ksm` does, thereby amortizing the long offloading latency. Compared to `ksm`, `Para-ksm` increases the amount of memory deduplication per CPU cycle used for `ksm` by 31–50% while decreasing the performance degradation to 1.3–2.7×.

1 Introduction

Memory technology scaling has stagnated, while the memory manufacturing cost has steadily increased, leading to a rise in the cost per bit [28]. At the same time, hyperscale applications demand increasingly larger memory capacities. These make memory account for 30–50% of the total hardware cost of a server [18, 38]. Meanwhile, virtualization techniques, such as Virtual Machines (VMs) and containers, are widely employed by hyperscalers to provide performance scalability and isolation for serving hyperscale applications. However, they often inefficiently use the given memory capacity, as they duplicate code and data across VMs and containers. Past work shows that redundant or duplicated memory pages consume 11%–86% of memory capacity, depending on the applications and operating systems [2, 21].

OS memory deduplication features, such as Kernel Same-page Merging (`ksm`), have been shown to effectively utilize memory capacity by significantly reducing duplicated memory pages [20, 31]. For example, Meta employs `ksm` to reduce memory pressure caused by running many Instagram worker processes on each server with limited memory capacity [35]. Nonetheless, recent work has reported that `ksm` not only imposes a high datacenter tax (*e.g.*, consuming 14–65% of total CPU cycles in our evaluation), but also pollutes the cache hierarchy, both of which are significant enough to degrade the performance of co-running applications [14]. Especially, page comparison (`memcmp`) and checksum computation (`xxhash`) in `ksm` are the two most taxing functions, collectively accounting for 38% of the CPU cycles consumed by `ksm` according to our analysis (§3).

To effectively reduce rising datacenter taxes, the latest server-class CPU has started integrating various on-chip accelerators capable of offloading common functions in the low levels of the datacenter software stack from CPU cores. For example, four on-chip accelerators have been integrated into the Intel Xeon Scalable Processor since its 4th generation. Compared to comparable off-chip accelerators, such as SmartNIC (SNIC) connected to CPU via off-chip PCIe [9, 19, 23–25], these on-chip accelerators offer several advantages, including reduced offloading latency, streamlined memory management, improved virtualization support, owing to their tight integration with CPU through on-chip PCIe and other supporting

[*]This work was supported in part by a grant from PRISM, one of the seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA, by grants from Institute of Information & Communications Technology Planning & Evaluation (IITP) funded by the Korea government (MSIT) (No. 2018-0-00503, RS-2024-00396013, No.RS-2024-00459797), and by a grant from the MSIT, Korea, under the Global Scholars Invitation Program (RS-2024-00456287) supervised by the IITP.

subsystems [40]. One of such on-chip accelerators is Data Streaming Accelerator (DSA), integrated to offload data movement and transformation functions, including memory copy, memory comparison, and checksum computation.

In this work, we not only explore the unique capabilities of DSA for `ksm` but also redesign `ksm` to efficiently offload `memcmp` and `xxhash` from CPU, significantly reducing the datacenter tax and performance degradation of co-running applications. We make the following specific contributions.

Contribution-1: Identifying opportunities in offloading `ksm` to DSA (§3–4). First, we demonstrate that offloading `memcmp` and `xxhash` to DSA, with a specific operating mode, can reduce the CPU cycle consumption for these functions by 6.6× and 6.0×, respectively, while obviating cache pollution. Although CPU may use non-temporal load instructions such as `MOVNTDQA` to prevent these functions from polluting the cache hierarchy, we show that they incur even more CPU cycle consumption than temporal load instructions. Second, we present `DSA-ksm`, which directly replaces CPU-based `memcmp` and `xxhash` with their DSA-based counterparts in `ksm`, reducing the performance degradation of co-running applications incurred by `ksm` from 1.6–5.8× to 1.0–1.6×. Nonetheless, we also uncover that these DSA-based functions take nearly 3× longer to return their results than their CPU-based counterparts although they yield CPU to other application processes during that time. Such long offloading latency makes `DSA-ksm` offer notably lower rates of memory deduplication than `ksm`. Lastly, we identify that leveraging the DSA's batching capability, which allows `memcmp` and `xxhash` to operate on multiple pages per offload, can significantly amortize the offloading latency (*e.g.*, by 81–83% with a batching size of 8), but it requires a significant redesign of `ksm`.

Contribution-2: Redesigning `ksm` to efficiently exploit the DSA's batching capability (§5). `ksm` scans pages to find those that can be merged while maintaining two Red-Black (RB) trees: stable and unstable trees, which store pointers to merged and previously scanned pages, respectively. It first picks a candidate page and compares it with pages in these trees (named tree pages) using `memcmp`. Then, based on the result of comparing the candidate and current tree pages, it determines the next tree page to compare with the candidate page and whether to compute the checksum of the candidate page using `xxhash`. That is, `ksm` is inherently a serial algorithm, and, consequently, `DSA-ksm` cannot exploit the DSA's batching capability, which requires selecting multiple tree pages to compare with the candidate page or multiple candidate pages to compare with a tree page. To tackle the shortcoming of `DSA-ksm`, we propose `Para-ksm`, a significant redesign of `ksm` that allows `memcmp` and `xxhash` to operate on multiple pages in parallel (or offload these functions to DSA for multiple pages in a batch). Specifically, we explore two algorithms: `Para-ksmC`, which compares multiple candidate pages with a single tree page, and `Para-ksmT`, which speculatively compares a single candidate page with multiple tree pages concurrently. However, we focus on `Para-ksmC` in this work, as we find that `Para-ksmT` significantly wastes DSA resources due to its speculative nature. `Para-ksmC` increases the amount of memory deduplication per CPU cycle used for `ksm` by 31–50% while decreasing the performance degradation of co-running applications incurred by `ksm` from 1.6–5.8× to 1.3–2.7×.

2 Background

2.1 Memory Deduplication

Memory deduplication has been developed to consolidate duplicate pages in memory, thereby improving memory utilization of a given memory capacity. One widely adopted implementation of memory deduplication is Kernel Same-page Merging (`ksm`), a feature integrated into Linux since kernel version 2.6. `ksm` is a Content-Based Page Sharing (CBPS) approach that scans the anonymous memory regions of processes to identify pages with the same contents, merges them, and subsequently reclaims the freed space.

The `ksm` daemon, `ksmd`, operates on the memory regions specified by the `madvise` system call and is controlled through parameters exposed in `sysfs` under `sys/kernel/mm/ksm`, such as `pages_to_scan`, `sleep_millisecs`, and `max_page_sharing` [37]. The key data structures of `ksm` are two Red-Black (RB) trees: stable tree and unstable tree. The stable tree stores pointers to all merged pages. If a process modifies a merged page, the Copy-on-Write (CoW) mechanism creates a copy of the page. In contrast, the unstable tree keeps pointers to scanned pages that have not yet been merged. Henceforth, we refer to the pages pointed to by the stable and unstable trees as *stable tree pages* and *unstable tree pages*, respectively. Both trees sort and organize the tree pages based on their contents, using the page content as the index to insert and look up a given page.

Figure 1 illustrates a single `ksm` scan over an advised memory region. During the scan, `ksmd` sequentially selects each

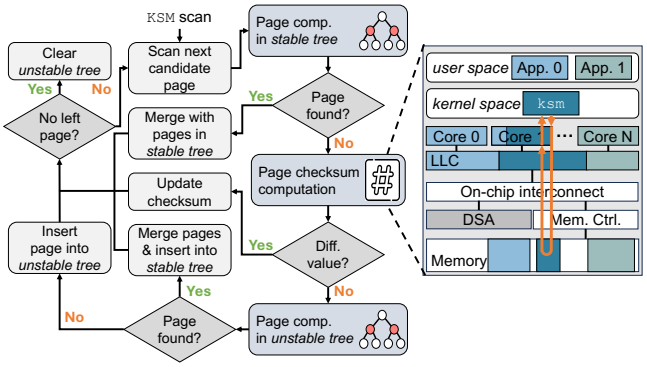


Figure 1: `ksm` scan process (left) and page checksum computation running on CPU (right).

candidate page and searches for a matching page in the stable tree. If a match is found, the candidate page is merged into the stable tree, and its memory region is freed. If no match is found in the stable tree, `ksmd` computes a checksum of the candidate page's content and compares it with the checksum from the previous scan. A mismatch between these two checksums indicates that the page has been modified between two scans, prompting `ksmd` to proceed to the next candidate page. If the checksums match, `ksmd` searches for a matching page in the unstable tree. If a match is found, the candidate page is inserted into the stable tree, the matched page is removed from the unstable tree to free the associated memory region, and the attribute of the candidate page is changed to read-only with CoW protection. If no match is found, the candidate page is inserted into unstable tree for future scans.

2.2 Data Streaming Accelerator

Hardware architecture. Figure 2 overviews the architecture of DSA. A CPU core interacts with DSA by directly submitting work descriptors to ❶ memory-mapped registers of DSA, referred to as portals. A work descriptor encapsulates the key information required to set up the offloading operation, such as the operation type, one or more source memory region addresses, and the completion record address. The operational unit of DSA is ❷ Group, comprising Work Queues (WQs) and Processing Engines (PEs). The number of WQs and PEs within a group is flexibly configurable by a user through the software interface of DSA. ❸ The Group arbiter dispatches the descriptors at the heads of WQs to available PEs, considering the relative priority of WQs set for QoS control. Upon receiving a work descriptor, ❹ a PE reads and operates on one or more source memory regions specified in the work descriptor through the cache-coherent on-chip interconnect, and updates the destination memory region if necessary and the completion record in host memory. The virtual address is used to access host memory, assisted by ❺ the on-device Address Translation Cache (ATC). ATC handles the address translations, sends requests to the IOMMU on the CPU, and handles page faults in coordination with the OS. This new feature allows both CPU and DSA to access shared memory regions in the virtual address space without memory pinning.

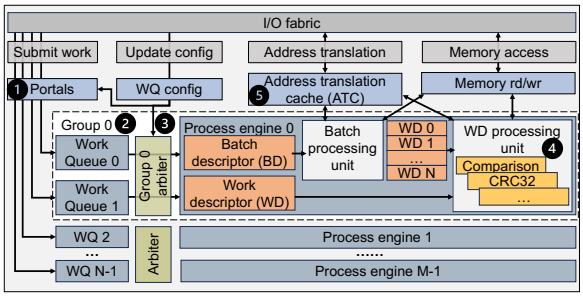


Figure 2: Architecture overview of DSA.

Software stack. The Intel Data Accelerator Driver (IDXD) is a kernel-mode driver responsible for initializing and managing DSA [10]. It provides two interfaces: a character device (`cdev`) interface for data-plane operations and a `sysfs` interface for control-plane configuration. In user space, applications can use the `libaccel-config` library to configure DSA and submit work descriptors to portals, exposed as `cdev` through `mmap`. However, there is no dedicated library existing for kernel-space DSA configuration and usage.

3 Opportunity in Offloading `ksm` to DSA

Prior work has reported that `ksm` incurs significant CPU cycle consumption when invoked [4, 14], disrupting the execution of co-running applications. In this section, we first break down the CPU cycles consumed by `ksm` to identify its CPU-intensive functions. Second, we demonstrate the potential of offloading these CPU-intensive functions to DSA.

3.1 CPU-intensive Functions of `ksm`

To show the considerable CPU cycle consumption by `ksm`, we pick a period that `ksm` co-runs with `Redis` on a CPU core. When `ksm` is invoked, we measure not only the overall CPU utilization but also the CPU utilization contributed by `ksm` and `Redis` separately. Figure 3 (left) shows that `ksm` increases CPU utilization by 14–65%, which is significant enough to degrade `Redis` performance when co-running.

Delving into the CPU utilization by `ksm`, we analyze the breakdown of CPU cycles by each function of `ksm`. Figure 3 (right) shows that `memcmp` and `xxhash`—which compares two pages and computes a 32-bit checksum value of a page—consume 21% and 17% of CPU cycles, respectively. These two functions perform simple operations compared to other functions of `ksm`. However, as they operate on thousands of pages per invocation, they collectively account for 38% of the CPU cycles consumed by `ksm`. Moreover, to operate on these pages, `memcmp` and `xxhash` need to bring them into the cache hierarchy, which incurs cache pollution [14, 15]. Although non-temporal or cache-bypassing load instructions, such as `MOVNTDQA`, can be used to implement these two functions, they are known to give lower bandwidth and higher latency

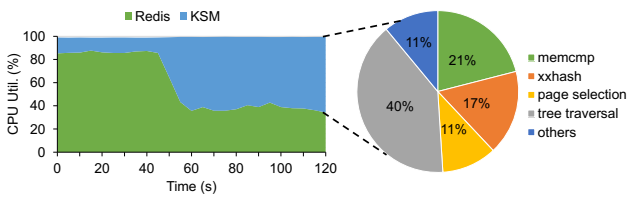


Figure 3: A snapshot of CPU utilization in a system where `CPU-ksm` co-runs with `Redis` under the YCSB-d workload and a detailed breakdown of CPU cycles consumed by `ksm`'s functions.

```
u32 dsa_xxhash(src, PAGE_SIZE, 0){
   struct dsa_hw_desc desc = { };
   struct dsa_completion_record comp;

   // ① Descriptor preparation
   desc.opcode = DSA_OPCODE_CRCGEN;
   desc.src_addr = src;
   desc.xfer_size = PAGE_SIZE;
   comp.status = 0;
   desc.completion_addr = (uintptr_t)&comp;
   desc.flags = IDXD_OP_FLAG_RCR|IDXD_OP_FLAG_CRAV;

   // ② Descriptor submission
   enqcmd(wq_portal, &desc);

   // ③ Descriptor completion check
   wait_for_interrupt(&comp.done);
   wake_up(ksm);
   return comp.result
}
```

Figure 4: Pseudo-code of DSA-based `xxhash` using asynchronous mode.

than temporal load instructions, proportionally consuming more CPU cycles. In contrast, DSA can perform these two functions while consuming far fewer CPU cycles than CPU without causing cache pollution.

3.2 Comparison between CPU and DSA: CPU Cycle Consumption

Among many DSA native operations, we can use 'Comparison' and 'CRC Generation' to implement `memcmp` and `xxhash`, respectively. Figure 4 depicts the pseudo-code of DSA-based `xxhash`. ① A CPU core first prepares a 64B work descriptor specifying (1) 'CRC Generation' as the operation to perform, (2) the address of a source page for checksum computation, (3) 4KB as the transfer size, (4) the address to write the checksum value as the completion record, and other fields. ② Then the CPU core submits the work descriptor to DSA by executing a single instruction, `ENQCMD`. When DSA receives the work descriptor, its processing engine directly operates on the source page without transferring the page into the cache hierarchy. After completing the checksum computation, it writes the checksum value to the completion record address. DSA supports two operating modes: synchronous and asynchronous modes. In synchronous mode, after submitting a work descriptor to DSA, the process continuously polls the completion record address until it receives the value returned by DSA. ③ In asynchronous mode, however, the process yields the CPU core to another process instead of polling, and it resumes execution upon receiving an interrupt, *i.e.*, a notification of work completion, from DSA.

Figure 5 compares the CPU cycle consumption by DSA-based `memcmp` and `xxhash` in kernel space with that of CPU-based counterparts. 'CPU-t' and 'CPU-nt' denote CPU-based `memcmp` and `xxhash` implemented using temporal and non-temporal load instructions, respectively. To avoid cache invalidation overhead from non-temporal loads, the associated pages are placed at random memory addresses and are not present in the CPU cache hierarchy. 'DSA-s' and 'DSA-a' represent DSA-based `memcmp` and `xxhash` implemented using synchronous and asynchronous modes, respectively. Our findings from Figure 5 are as follows. First, DSA-a-based `memcmp` and `xxhash` consume 6.6× and 6.0× fewer CPU cycles than their CPU-t-based counterparts, respectively, as they consume CPU cycles only for preparing and submitting a work descriptor and retrieving the completion record. Second, CPU-nt does not cause cache pollution as DSA, but it consumes 1.1–3.2× more CPU cycles than CPU-t. We use `MOVNTDQA` as CPU-nt in kernel space, which requires full FPU state, thereby increasing CPU cycle consumption compared to CPU-t. Lastly, DSA-s-based `memcmp` and `xxhash` consume CPU cycles comparable to their CPU-t-based counterparts, respectively. Since they do not reduce a meaningful amount of CPU cycle consumption compared to their CPU-t-based counterparts, we use DSA-a-based `memcmp` and `xxhash` to implement DSA-based `ksm` in this work.

4 `DSA-ksm`: CPU-DSA Cooperative `ksm`

We have demonstrated that `memcmp` and `xxhash` consume a significant number of CPU cycles when `ksm` is invoked, and that they can be offloaded to DSA to considerably reduce the CPU cycle consumption compared to those executed by CPU. Since DSA-based `memcmp` and `xxhash` decrease the consumption of CPU cycles without incurring cache pollution compared to their CPU-based counterparts, they can significantly reduce the performance degradation of co-running applications (§6). In this section, we present CPU-DSA cooperative `ksm` (`DSA-ksm`), where DSA performs `memcmp` and `xxhash` (collectively referred to as data-plane functions hereafter) while CPU executes the remaining control-plane functions (*e.g.*, a function selecting the next candidate page to scan) to minimize the performance degradation of applications co-running with `ksm`. This work differs from prior approaches that offload these data-plane functions to off-chip PCIe accelerators [7, 14], because it not only exploits innovations in the on-chip PCIe accelerator ecosystem [40] but also advances the `ksm` algorithm to fully capitalize on the capability of DSA.

Workflow. We first implement DSA-based `memcmp` and `xxhash` using the APIs provided by the IDXD driver (§2.2) and then build a kernel-space library integrating these func-

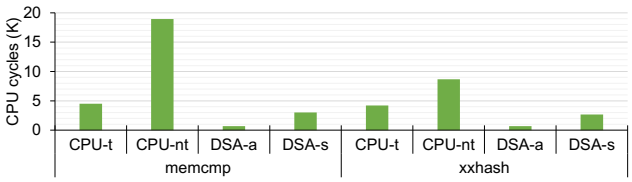


Figure 5: CPU cycle consumption of CPU and DSA-based `memcmp` and `xxhash`.

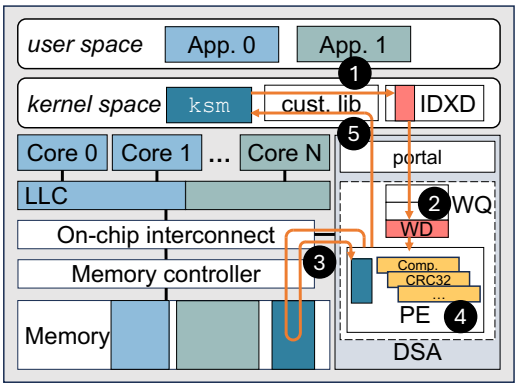


Figure 6: DSA-ksm workflow.

tions, enabling them to be invoked directly from ksm running in the kernel space. In contrast to prior work [7, 14], DSA-ksm leverages DSA's virtual memory support, eliminating the need for memory pinning and virtual-to-physical address translation, thereby simplifying kernel integration and reducing overhead.

Figure 6 depicts the workflow of DSA-ksm. To execute either memcmp or xxhash, ❶ DSA-ksm invokes the corresponding function from the library, first preparing a work descriptor that includes the virtual addresses of memory pages to operate on. ❷ Subsequently, it submits the work descriptor to a work queue (WQ) through a DSA portal. After the successful submission, the process sleeps to yield the CPU core to another process (*e.g.*, a co-running application) until it receives an interrupt from DSA. Receiving the work descriptor from the WQ, a processing engine (PE) ❸ reads the page(s) specified in the work descriptor from host memory into its internal buffer and ❹ conducts the operation specified in the work descriptor. As soon as DSA completes the operation on the entire page(s), ❺ it writes the completion record comprising the result to the corresponding address and sends an interrupt, which wakes up the process. Then, resuming the execution, the function reads the completion record and returns the result to DSA-ksm. To minimize the performance cost of handling interrupts, we choose to sleep for a fixed amount of time, wake up, and poll the completion record, observing that the completion latency exhibits little variance.

Alleviation of long offloading latency. Although DSA-based memcmp and xxhash considerably decrease the consumption of CPU cycles without incurring cache pollution compared to their CPU-based counterparts, we find that using DSA decreases the memory deduplication rate. Figure 7 shows that when the batch size is 1 (submitting one work descriptor per offload), DSA-based memcmp and xxhash present 2.6× and 2.7× longer latency than CPU-based counterparts, respectively. The increased latency is attributed to the time taken to submit a work descriptor and receive the notification of completion, denoted by 'submission' and 'waiting', respectively. While the waiting latency includes the execution time of a given function in DSA, both submission and waiting latencies for offloading a function operating on a 4KB page are dominated by the communication time between the CPU core and DSA over on-chip PCIe [11]. Compared to SNIC-based offloading over off-chip PCIe [14] , DSA still achieves lower latency (*e.g.*, 6μs versus 17μs for xxhash), as off-chip PCIe incurs longer round-trip latency [40] and additional RDMA stack overhead.

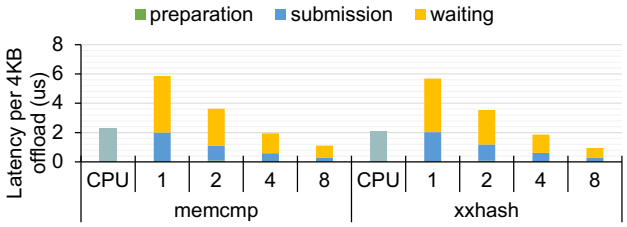


Figure 7: Latency of memcmp and xxhash running on CPU and DSA with varying batching size.

To reduce the latency, instead of submitting a work descriptor one at a time, we can submit a batch descriptor (BD) to DSA [17], which specifies the starting address of an array of work descriptors and the number of work descriptors in the array. The memory regions that these work descriptors specify for DSA to operate on do not need to be contiguous. The batch descriptor is submitted to a WQ through a portal in the same way as a work descriptor. However, receiving the batch descriptor, a batch processing unit fetches all the work descriptors in the array in a single PCIe transaction, forwards them to the processing unit for execution, and writes the completion records of the work descriptors in another PCIe transaction.

Figure 7 demonstrates that larger batch sizes decrease the average processing latency per work descriptor. For instance, a batch size of 8 offers 81% and 83% lower latency than the batch size of 1 for memcmp and xxhash, respectively. This notable latency reduction is primarily attributed to the reduced cost of (1) communication between CPU and DSA for submitting work descriptors and receiving completion records over PCIe, and (2) context switching between the ksm process and the application process. Note that the PCIe communication latency is dominated by other overheads rather than data transfer time when the data-transfer size is small [17]. To benefit from DSA's batching capability, ksm should be able to select multiple tree pages to compare with the candidate page or multiple candidate pages to compare with a tree page. However, it is not designed to do so as it is a serial algorithm.

5 Para-ksm

DSA batch processing effectively amortizes the long latency of offloading functions from CPU (§4), increasing the rate of memory deduplication. However, the vanilla ksm implementation inherently lacks parallelism to leverage DSA batch processing. To address this shortcoming, we propose Para-ksm,

a parallelized ksm optimized for efficient batch processing. Para-ksm significantly accelerates the deduplication process by batching data-plane functions associated with candidate page and further exploring batching strategies for tree page.

5.1 Sequential Processing in ksm

memcmp and xxhash are used to identify matching pages during the searches in stable and unstable trees, and to verify whether page contents have been modified between scans. We identify two **B**ottlenecks that impede the parallel processing of these data-plane functions: **(B1)** single candidate page selection and **(B2)** ordered comparisons in tree search, which stem from the properties of the trees employed by ksm.

Single candidate page selection. As illustrated in Figure 1, ksm selects one candidate page at a time from the advised memory region. It then searches for the matching page by sequentially comparing candidate page with tree pages and computes a checksum of its content if necessary. This design naturally leads to the sequential execution of data-plane functions, significantly limiting parallelism. The root cause lies in the rebalancing process of the RB trees used in ksm. An RB tree, as a type of self-balancing binary search tree (BST), maintains balance by performing recoloring and/or rotations after each node insertion or deletion, keeping a longest-to-shortest path ratio of at most two. This rebalancing ensures efficient search operations in the RB tree with a time complexity of $O(\log N)$, where N is the number of nodes in the tree. However, rebalancing prevents batch processing of candidate pages because inserting even one candidate page can modify the tree structure, invalidating comparisons already performed for other candidate pages in the same batch.

Ordered comparisons in tree search. ksm manages the memory pages using two RB trees, unstable tree and stable tree, which store pointers to pages and organize them in a ***lexicographical*** order based on page contents. This ordering enables efficient binary searches within the trees to find matching pages by avoiding unnecessary comparisons. However, since each comparison determines the next search direction, comparisons along the search path must be performed sequentially. For example, DSA-ksm typically needs to submit four separate WDs sequentially to complete a search in a four-level RB tree, as the selection of the tree page in each WD depends on the previous comparison result returned by DSA. As shown in Figure 7, this sequential approach can result in higher latency per function than CPU-based counterparts, compromising the memory-saving performance of DSA-ksm.

5.2 Para-ksmC: Candidate Page Batching

To eliminate **B1**, Para-ksm enhances vanilla ksm by incorporating candidate page batching and redesigning tree search and insertion (Para-ksmC). Para-ksmC selects N consecutive candidate pages from the advised memory region, groups them into a single batch, and conducts data-plane functions over the batch in a parallelized manner. These parallelized memory comparisons and checksum computations are offloaded to DSA using batch descriptors to exploit DSA's batch processing capabilities. As the batch processing of xxhash functions on candidate pages is straightforward, this section focuses on how Para-ksmC supports and leverages batch processing of memcmp functions in the renovated search and insertion design. In particular, we use unstable tree search and insertion as an example in this section, since the operations in the stable tree are structurally identical.

5.2.1 Search in Para-ksmC

Para-ksmC selects 256 consecutive memory pages to build a candidate page batch based on a thorough parameter search (§6.5). Since candidate pages are pre-allocated before ksm execution and their volume is typically sufficient, Para-ksmC does not wait to accumulate a full batch of 256 candidate pages. If fewer pages remain during a ksm scan pass, Para-ksmC immediately offloads all remaining pages to the DSA, avoiding unnecessary delays. Before starting the tree search, Para-ksmC creates a search_result for each candidate page in the batch, which encapsulates the addresses of candidate page as well as the addresses of its predecessor and successor in the tree. The predecessor and successor of a candidate page are defined as follows:

$$predecessor(x,S) = \max\{s \in S \mid s < x\} \quad (1)$$

$$successor(x,S) = \min\{s \in S \mid s > x\} \quad (2)$$

, where S is the set of compared tree pages.

The addresses of predecessor and successor are initialized as NULL in search_result. Para-ksmC leverages a key **P**roperty of RB trees: **(P1)** rebalancing does not alter the predecessor or successor of a candidate page. Based on **P1**, Para-ksmC uses search_result to track the addresses of predecessor and successor, as each candidate page is always inserted as a child of either its predecessor or successor. This approach eliminates repeated searches for a proper insertion spot after tree rebalancing caused by other candidate pages from the batch. Further details on the usage of search_result are provided in Section 5.2.2.

Figure 8 illustrates how Para-ksmC performs tree search in a three-level tree as an example. For simplicity, numerical values in nodes represent page contents, and the arrows indicate

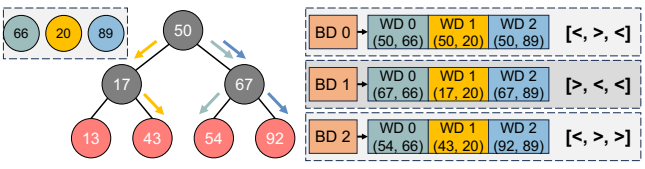


Figure 8: Search in Para-ksmC.

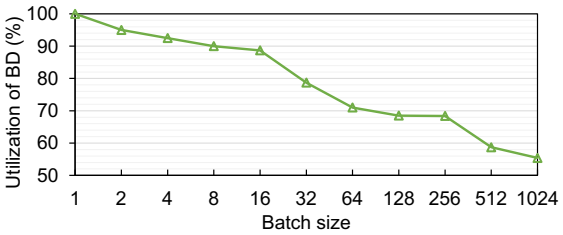


Figure 9: Utilization of BD in `Para-ksmC` with varying candidate page batch sizes.

search paths for candidate pages, with each path highlighted in the same color as the corresponding candidate page. The search starts at the root node. `Para-ksmC` prepares a list of WDs, each containing the address of a candidate page and the address of a tree page pointed to by the root node. A BD is then created to point to the WD list and submitted to DSA for batch processing of `memcmp` functions. Based on the comparison results returned from DSA, the WDs are updated with the addresses of the next selected tree pages for further comparisons with candidate pages. The search for each candidate page terminates when a matching page is found or when a leaf node is reached. During the search, predecessor and successor addresses in the `search_result` are dynamically updated based on comparison outcomes. If the encountered tree page is smaller than the candidate page, predecessor address is updated to the address of the tree page; if larger, successor address is updated. If the tree page and candidate page are the same, both successor and predecessor addresses are set to the address of this matching tree page. While `Para-ksmC` offloads `memcmp` operations to DSA efficiently in a batching manner, it preserves the original comparison paths of `CPU-ksm` to maintain correctness, without altering the tree traversal logic.

Unlike `CPU-ksm`, which interleaves page comparison and tree insertion for each page, `Para-ksmC` decouples the two stages by first conducting comparisons and then inserting pages in the batch, achieving similar per-page comparison and insertion times as `CPU-ksm`. To enable this, `Para-ksmC` processes all candidate pages in the batch simultaneously by deferring tree operations, such as insertion or deletion, until searches for all candidate pages in the batch are complete. This approach removes the dependency on sequential candidate page selection, preventing tree rebalancing from interfering with the search paths of other candidate pages within the batch. However, this design introduces a potential issue dubbed ***comparison skewness***. More specifically, during the search, the number of comparisons required for different candidate pages within the same batch can vary notably, leading to underutilization of BD and diminishing the benefits of batch processing. For example, in a batch of three candidate pages, one candidate page may find its matching page after five comparisons, while the other two require ten. As a result, `Para-ksmC` submits ten BDs to complete the comparisons for the entire batch, even though only two WDs remain in the WD list after the fifth BD. We examined the BD utilization on `memcmp` functions across various batch sizes, where utilization is defined as the ratio of the number of WDs in the list to the preset candidate page batch size. Figure 9 presents the utilization of BD when `Para-ksmC` co-runs with `Graph500` (§6). The utilization remains above 90% when batch sizes are below 16. However, as the batch size increases, utilization drops sharply, falling below 60% at the batch size of 512. This underutilization of BD indicates an increasing skewness in larger batches, reducing the benefits of batch processing on DSA and highlighting the need for a careful batch size selection to maximize the effectiveness of `Para-ksmC`.

5.2.2 Insertion in `Para-ksmC`

After completing the search for all candidate pages in the batch, `Para-ksmC` proceeds to insert them into the unstable tree. `Para-ksmC` tracks the predecessor and successor of each candidate page in the `search_result` (§5.2.1) and uses a hash table to group `search_results` based on the hash value of the predecessor address as shown in Figure 10. We leverage another property of RB trees: **(P2)** two candidate pages either share the same pair of (predecessor, successor) or have distinct non-overlapping pairs. By **P2**, we know that the `search_results` in the same group share the same successor. After adding all `search_results` into the hash table, `Para-ksmC` traverses the hash table to insert candidate pages as outlined in algorithm 1.

If a group in the hash table contains only one `search_result`, the candidate page is inserted as either the left child of successor or the right child of predecessor, using the addresses recorded in `search_result`. Since groups have disjoint (predecessor, successor) pairs, inserting a candidate page in one group does not affect insertions in other groups. For groups with multiple `search_results`, `Para-ksmC` sorts them in descending order based on the content of candidate pages pointed to by each `search_result`, using the CPU-based `memcmp` function. If the same pages are found in the group, they are merged and inserted into stable tree directly,

Algorithm 1: unstable tree insertion in `Para-ksmC`.

```
1  for group ∈ hash_table do
2      if len(group) > 1 then
3          compare_and_reorder(group)
4          update_succ(group)
5      for search_result ∈ group do
6          cand_page = search_result[0]
7          (pred, succ) = search_result[1]
8          if pred.right == NIL then
9              pred.right = cand_page
10         else
11             succ.left = cand_page
12         Rebalance the tree.
```

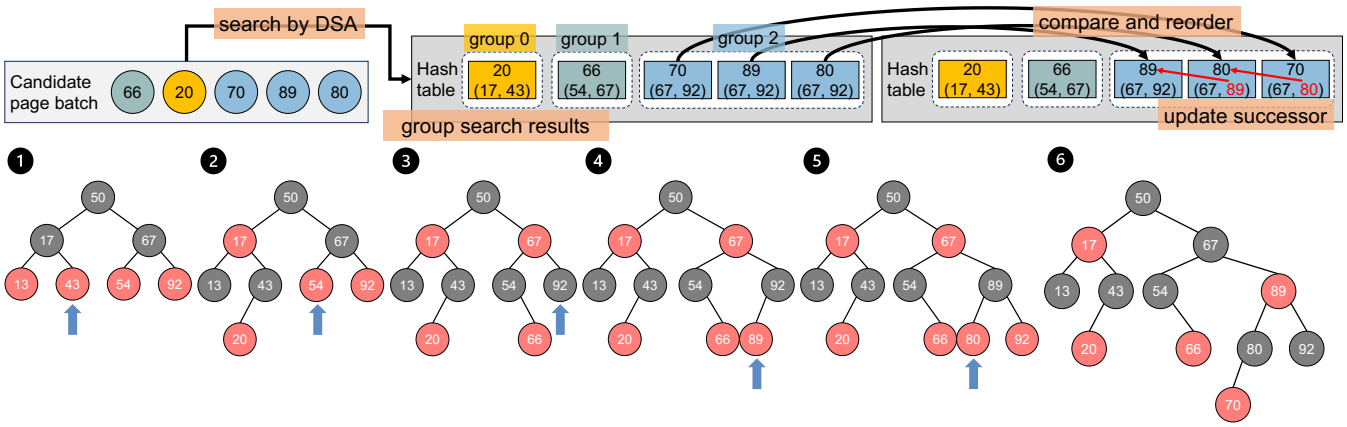


Figure 10: Insertion in `Para-ksmC`. Blue arrows point to the insertion locations in unstable tree.

and their corresponding `search_results` are removed. The successor address in each remaining `search_result` is then updated to the candidate page address in the preceding `search_result` in the sorted group. This intra-group sorting and successor update ensure that `search_result` records the correct predecessor and successor, remaining unaffected by any rebalancing triggered by earlier insertions. Figure 10 (lower) illustrates a step-by-step example of candidate page insertion into unstable tree in `Para-ksmC`. After each candidate page insertion, the unstable tree rebalances through recoloring (e.g., ❶ to ❷) and/or rotation (e.g., ❹ to ❺). Nonetheless, with our careful design, `Para-ksmC` can always identify the correct insertion point for each candidate page regardless of how the tree structure is changed.

5.3 `Para-ksmT`: Tree Page Batching

To mitigate **B2**, `Para-ksm` introduces ***speculative page comparison*** for batched tree pages (`Para-ksmT`). Instead of waiting for each comparison result to select the next tree page to be compared with candidate page, `Para-ksmT` speculatively compares the candidate page with tree pages across the next M levels and offloads all comparisons in a single BD as shown in Figure 11. Specifically, `Para-ksmT` performs an M-level inorder traversal starting from a specified node to aggregate tree pages. ❶ The inorder traversal ensures that tree pages

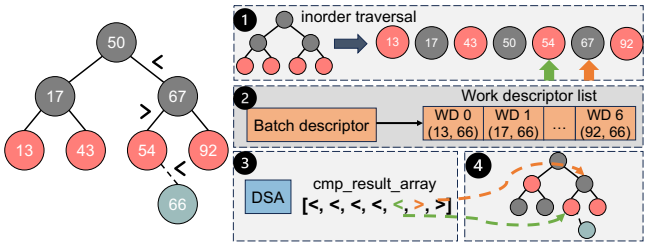


Figure 11: Search and insertion in vanilla `ksm` (left) and `Para-ksmT` (right). Green and orange arrows point to the predecessor and successor, respectively.

are visited in ascending order based on their contents, and their addresses are stored sequentially in a `tree_page_array`. `Para-ksmT` creates a WD list, where each WD includes the candidate page address and a tree page address from `tree_page_array` in sequential order. ❷ A BD pointing to the WD list is then created and submitted to DSA to execute `memcmp` functions in one offload. Upon completion, ❸ the comparison results are retrieved from the completion record and stored in the `cmp_result_array` in the same order as the WDs in the WD list. Since `cmp_result_array` preserves the same order as `tree_page_array`, predecessor (successor) is easily located in `tree_page_array` by the index of the last negative (first positive) result in the `cmp_result_array`. If predecessor (successor) is not found, `Para-ksmT` checks the left (right) child of the smallest (largest) tree page and inserts candidate page if a spot is available. If both exist, `Para-ksmT` checks the right child of predecessor and/or the left child of successor and inserts candidate page if the child is absent. If no spots are available for insertion, `Para-ksmT` continues the search on the subtrees until reaching the leaf nodes. Figure 11 (right) illustrates an example search and insertion in `Para-ksmT`. With M set to three, `Para-ksmT` speculatively compares the candidate page with seven tree pages in the next three levels in parallel and offloads seven page comparisons in a single BD to DSA. In contrast, `DSA-ksm` requires three sequential page comparisons and submits three separate WDs to DSA.

While `Para-ksmT` leverages batch processing, its memory deduplication rate remained limited. Our evaluation indicates that `Para-ksmT` achieves 9.2% of `CPU-ksm` 's deduplication rate, slightly higher than `DSA-ksm`. This underwhelming performance is caused by the overhead of unnecessary comparisons introduced by speculative page comparisons, resulting in a significant waste of DSA computation resources. After traversing M levels, `Para-ksmT` offloads $2^M - 1$ page comparisons to DSA while `DSA-ksm` only offloads M page comparisons. That is, only M out of $2^M - 1$ comparisons contribute to identifying duplicate pages. In addition to extra computation in DSA, speculative traversal also incurs extra data move-

Table 1: Hardware and software configurations.

Hardware	Description
CPU	Intel® Xeon 8460Y+ CPUs @2.0 GHz, 40 cores and 2.625 MB LLC per core, Hyper-Threading disabled
Memory	8-Ch. w/ 8 32GB DDR5-4800 DRAM modules
DSA	One group with one 64-entry WQ and four PEs
SmartNIC	NVIDIA BF-3 [26], PCIe 5.0, DDR5-5200
Software	**Description**
OS (kernel)	Ubuntu 18.04.6 LTS (Linux kernel 6.2.15)
`ksm`	`sleep_between_scan` = 20ms, `free_mem_thres` = 80 `pages_to_scan` = [64, 2048] # adjusted by *ksmtuned*
`Para-ksmC`	`cand_page_batch_size` = 256
Virtual machine	Hypervisor: QEMU-KVM v2.11.1 Ubuntu Cloud 22.04, 1 Virtual Core, 6GB memory

ment. Our evaluation shows that when $M = 6$, this results in an extra 2% of system DRAM bandwidth consumption. Moreover, `Para-ksmT` lacks support for batch processing of `xxhash` functions, further restricting its effectiveness compared to `Para-ksmC`. We consider `Para-ksmT` a complementary optimization to `Para-ksmC`, potentially beneficial in environments with abundant computational resources to handle offloaded functions. However, due to the limited availability of DSA devices in our setup, the following sections focus on evaluating `Para-ksmC`.

6 Evaluation

6.1 Evaluation Setup

System Setup. Table 1 summarizes the hardware and software configurations used in our experiments. CPU core frequency is locked at 2.0 GHz, and hyper-threading (HT) is disabled to ensure consistent performance measurements across runs. The on-chip DSA is configured with one group consisting of a 64-entry WQ and four processing engines. Candidate page batch size in `Para-ksmC` is set to 256 based on the evaluation in Section 6.5. We reproduce `STYX` [14] on NVIDIA BF-3 [26] for a comparison with `Para-ksmC`.

Workloads. We evaluate three representative workloads: `Liblinear` [6], `Graph500` [36], and `Redis` [30] with Yahoo! Cloud Serving Benchmark (`YCSB`) [5]. `Liblinear` is a widely used library for large-scale linear regression, processing datasets with millions of instances and features. We run the Support Vector Regression (SVR) from `Liblinear` on the `SUSY` dataset [39]. `Graph500` is a high-performance computing benchmark designed to stress memory subsystems with its graph generation and traversal. We use `Graph500` to generate a graph of scale 23 and edge factor of 16, and run breadth-first search (BFS) on it. `Redis` is a high-performance, in-memory Key-Value Store (KVS) database widely adopted in real-world applications requiring low-latency data access. `YCSB` is a benchmarking framework to evaluate the performance of KVS databases, providing four workloads: (a) `update heavy` (50% read and 50% update), (b) `read heavy` (95% read and 5% update), (c) `read only` (100% read) and (d) `read latest` (95% read and 5% insert), with a uniform distribution for key values.

Methodology. We launch 40 VMs, each assigned one virtual core (vCPU) and 6 GB RAM, pinned to a dedicated physical core. For `Liblinear` and `Graph500`, each VM runs the workload independently. For `Redis`, we organize the 40 VMs into 10 groups, each comprising 3 VMs as `Redis` clients and 1 VM as the `Redis` server to handle the requests. Given the memory-intensive nature of `Redis` client operations, we employed Intel's Cache Allocation Technology (CAT) [12] to partition the LLC. Specifically, one cache way is reserved for each `Redis` server, while the remaining cache ways are shared among the clients to minimize interference between client and server processes. We configure the vanilla `ksm` running on CPU (`CPU-ksm`) and `Para-ksmC` to achieve comparable memory savings and provide a fair evaluation of their impact on the co-running workloads.

6.2 Application Performance

We adopt execution time as the performance metric for `Liblinear` and `Graph500`, and 99th-percentile (p99) latency for `Redis`, as user-serving datacenter applications like `Redis` must meet strict tail latency requirements [22, 27]. Figure 12 shows the performance degradation of `Liblinear`, `Graph500`, and `Redis` on systems that deploy `CPU-ksm`, `DSA-ksm`, `STYX`, and `Para-ksmC`, normalized to a system without running `ksm` (`no-ksm`). Performance degradation refers to the increase in execution time for `Liblinear` and `Graph500`, or the increase in p99 latency for `Redis`, compared to `no-ksm`.

On average (geometric mean), `CPU-ksm` incurs 3.3× performance degradation over `no-ksm`. `YCSB` workloads on `Redis` experience higher degradation, with an average of 3.7×, while `Liblinear` and `Graph500` have an average of 2.1× execution time increase, as the tail latency is more sensitive to the CPU contention caused by the intensive data-plane functions running on the CPU. In contrast, `DSA-ksm`, `STYX` and `Para-ksmC` incur 1.3×, 1.4× and 2.1× performance degradation compared to `no-ksm`, respectively. That is, `DSA-ksm`, `STYX` and `Para-ksmC` reduce the performance degradation by 2.5×, 2.4× and 1.6× compared to `CPU-ksm`. Both `DSA-ksm` and `Para-ksmC` alleviate CPU contention by offloading data-plane functions (*i.e.*, `memcmp` and `xxhash`) to DSA, freeing CPU cycles for co-running applications and reducing the cache pollution as shown in Section 6.3. `Para-ksmC` exhibits higher performance degradation than `DSA-ksm` because it retains intra-group page comparisons on the CPU and incurs additional overhead (*e.g.*, successor update) during search and insertion (§5.2).

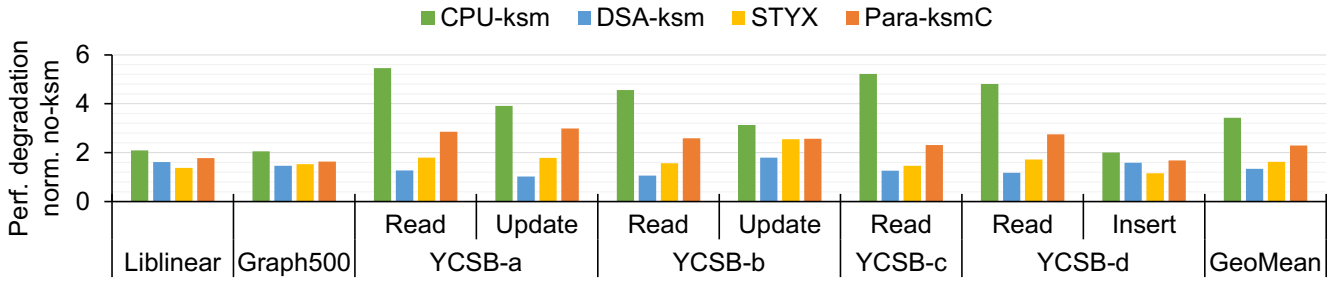


Figure 12: Workload performance degradation under CPU-ksm, DSA-ksm, STYX and Para-ksmC, normalized to no-ksm.

6.3 Impact on CPU Resource

To disclose how DSA-ksm and Para-ksmC mitigate the impact of CPU-ksm on the co-running applications, we investigate their CPU cycle consumption and LLC miss rates of server CPU in the system.

6.3.1 CPU Cycle Consumption

Both DSA-ksm and Para-ksmC conserve the server CPU cycles by offloading the CPU-intensive data-plane functions to DSA. We identify the intervals during which ksm and application co-run on a server CPU core and measure the server CPU cycles consumed by ksm within these intervals. The total CPU cycles consumed by ksm are summed and divided by the total server CPU cycles during co-running periods to calculate the average CPU utilization. The reported values for YCSB workload are geometric averages across four YCSB workloads.

Table 2 shows that CPU-ksm consumes significant CPU cycles when it co-runs with applications, with an average CPU utilization of 48%, consistent with our observations in Section 3.1. On average, DSA-ksm and Para-ksmC reduce the CPU utilization of ksm by 85% and 36%, respectively, compared to CPU-ksm. STYX achieves a comparable level of CPU cycle reduction as DSA-ksm, as both approaches succeed in offloading the memcmp and xxhash from host CPU. The saved CPU cycles can be used for applications, reducing the performance degradation caused by the deduplication feature. Para-ksmC consumes more CPU cycles than DSA-ksm because it does not offload intra-group memcmp to DSA and performs extra operations (*e.g.*, hashing on search_results and successor update) to ensure correct insertion in trees. Despite the higher CPU cycle consumption than DSA-ksm, Para-ksmC delivers more effective and efficient memory deduplication, as demonstrated in Section 6.4.

Table 2: CPU utilization of CPU-ksm, DSA-ksm, STYX and Para-ksmC when they co-run with different workloads.

	Liblinear	Graph500	YCSB	GeoMean
CPU-ksm	44.5%	39.2%	51.7%	48.2%
DSA-ksm	5.8%	5.2%	8.5%	7.3%
STYX	5.4%	5.7%	5.4%	5.5%
Para-ksmC	37.6%	29.8%	29.8%	31.0%

6.3.2 LLC Miss Rate

By offloading data-plane functions, DSA-ksm and Para-ksmC not only conserve CPU cycles but also alleviate cache pollution. We evaluate the cache pollution by measuring the LLC miss rates of the server CPU every second while running CPU-ksm, DSA-ksm, STYX, and Para-ksmC.

Table 3 reports the average LLC miss rates across workloads. CPU-ksm leads to 7%–114% higher LLC miss rate than no-ksm as ksm brings cold pages into CPU caches for comparison or checksum calculation when invoked. Liblinear and Graph500 exhibit smaller LLC miss rate increases as they are more memory-intensive than Redis. DSA-ksm and Para-ksmC reduce the LLC miss rate by 28% and 18% on average compared to CPU-ksm. The reduction is achieved because DSA reads data directly from host memory to its processing engines, bypassing the CPU cache hierarchy when executing the offloaded functions (❸ in Figure 6). However, cache pollution is not fully eliminated in DSA-ksm and Para-ksmC, as only the two dominant data-plane functions, memcmp and xxhash, are offloaded, while control-plane functions remain on the CPU.

6.4 Deduplication Performance

After evaluating the impact on application performance and CPU resources, we focus on the deduplication performance of the memory deduplication feature in this section. Specifically, we examine the deduplication effectiveness and efficiency of

Table 3: LLC miss rate of systems (no-ksm, CPU-ksm, DSA-ksm, STYX and Para-ksmC).

	Liblinear	Graph500	YCSB	GeoMean
no-ksm	52.5%	8.9%	20.7%	21.3%
CPU-ksm	56.4%	11.9%	42.5%	30.5%
DSA-ksm	52.2%	9.0%	22.9%	22.1%
STYX	53.5%	9.5%	21.7%	21.9%
Para-ksmC	52.7%	10.4%	28.4%	25.0%

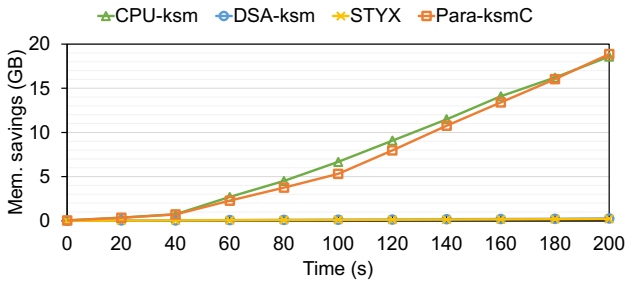


Figure 13: Memory saving under `CPU-ksm`, `DSA-ksm`, `STYX` and `Para-ksmC` when `Liblinear` runs.

`CPU-ksm`, `DSA-ksm`, and `Para-ksmC`.

6.4.1 Deduplication Effectiveness

Deduplication effectiveness is measured as memory saving achieved at a given time while the deduplication feature co-runs with workloads. Although metrics like the number of scanned or saved pages can reflect the deduplication progress, memory savings offers a more direct and comprehensive measure of deduplication effectiveness, as it captures both the amount and usefulness of eliminated duplicate pages. We pick a 200-second window during which the deduplication feature runs alongside the workload and measure the memory savings every 20 seconds.

Figure 13 shows the memory saving over the selected period using `Liblinear` as a representative workload, reflecting similar trends observed across other workloads. `DSA-ksm` achieves only 12% of `CPU-ksm`'s memory saving after the first 20 seconds, with the gap widening over time. The deduplication effectiveness of `DSA-ksm` is hindered by the long execution time of DSA-based `memcmp` and `xxhash` functions (Figure 7), resulting from the communication overhead between CPU core and DSA. `STYX` exhibits slightly lower deduplication effectiveness than `DSA-ksm` in the same window, due to even longer offloading latency that further delays memory savings. Although both `DSA-ksm` and `STYX` eventually converge toward the memory savings of `CPU-ksm`, their short-term effectiveness is constrained by offloading inefficiencies. The limited deduplication effectiveness of `DSA-ksm` highlights the necessity of exploiting the batching capability of DSA to amortize the offloading overhead.

`Para-ksmC` achieves memory savings comparable to `CPU-ksm` by redesigning `ksm` and leveraging the batching capability of DSA. During the period, `Para-ksmC` delivers 20% less memory savings than `CPU-ksm` after the first 100 seconds and eventually narrows the gap to under 1% by the end. The significant improvement in deduplication effectiveness of `Para-ksmC` over `DSA-ksm` stems from the redesign of vanilla `ksm` that enables `memcmp` and `xxhash` to operate on multiple pages in parallel. This software enhancement allows functions to be offloaded to DSA in a batch, substantially reducing offloading overhead and consequently boosting deduplication effectiveness.

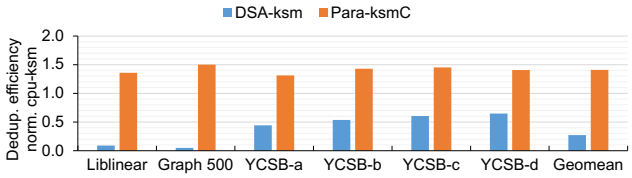


Figure 14: Deduplication efficiency of `DSA-ksm` and `Para-ksmC` normalized to `CPU-ksm` across workloads.

6.4.2 Deduplication Efficiency

Deduplication efficiency is defined as the memory savings per 1K CPU cycles consumed by the deduplication features (*i.e.*, `CPU-ksm`, `DSA-ksm`, and `Para-ksmC`). This metric offers a more comprehensive evaluation by accounting for both CPU cycle consumption and memory savings, reflecting how efficiently deduplication features utilize CPU cycles. Similar to Section 6.4.1, we select a 200-second window during which the deduplication features and user applications run concurrently. The deduplication efficiency is calculated by dividing the total memory savings by the CPU cycles consumed by the deduplication features during this period.

Figure 14 presents the deduplication efficiency of `DSA-ksm` and `Para-ksmC` normalized to `CPU-ksm` across workloads. `DSA-ksm` achieves only 5%–65% deduplication efficiency compared to `CPU-ksm`. This reduced efficiency is primarily because `DSA-ksm` offloads each function individually in DSA asynchronous mode. Excessive context switching and CPU-DSA communication overhead overwhelm the CPU cycle savings from function offloading, leading to lower overall efficiency. In contrast, `Para-ksmC` achieves 1.3×–1.5× higher deduplication efficiency than `CPU-ksm`. Its superior efficiency over `DSA-ksm` is attributed to the batch processing on DSA, which consolidates multiple functions into a single BD instead of processing individual WDs. Batch processing amortizes the submission overhead across multiple functions, reduces the frequency of context switches, and minimizes the software overhead for checking completion records. Although `Para-ksmC` introduces additional operations to vanilla `ksm`, the benefits of batch processing outweigh these overheads, yielding significantly higher efficiency.

6.5 Impact of Batch Size

In this section, we investigate how the deduplication performance of `Para-ksmC` changes under different batch size configurations. We measure the memory saving every 20 seconds and CPU cycles consumed by `Para-ksmC` over a 160-second period during which `Para-ksmC` co-runs with `Graph500`.

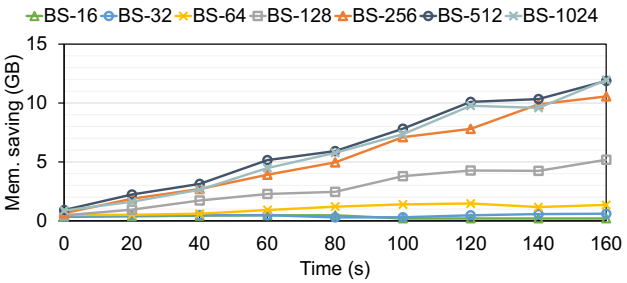


Figure 15: Memory saving achieved by `Para-ksmC` under varying batch sizes (BS) while co-running with `Graph500`.

Figure 15 shows memory saving achieved by `Para-ksmC` with batch size ranging from 16 to 1024, where 1024 is the maximum batch size supported by DSA. As the batch size doubles from 16 to 256, `Para-ksmC` achieves 3.0×, 2.2×, 3.9× and 2.1× increases in memory savings over the preceding batch size. However, beyond the batch size of 256, the improvements plateau, with only 1.1× and 1.0× higher memory savings as the batch size doubles from 256 to 512 and 512 to 1024, respectively. Two factors drive this plateau. First, DSA 's limited function processing capability becomes a bottleneck for larger batches, increasing the latency of DSA-based functions and degrading deduplication performance. Second, the comparison skewness issue, as shown in Figure 9, exacerbates the underutilization of BD at large batch sizes, reducing the overall efficiency of batch processing. We also calculate the deduplication efficiency of `Para-ksmC` across various batch sizes normalized to the efficiency achieved at a batch size of 16, as shown in Figure 16. The deduplication efficiency rises steadily up to 2.2× at a batch size of 256, then slightly declines at larger batch sizes. This decline results from DSA's processing limitations and the effects of comparison skewness, as discussed above.

7 Related Work

Memory deduplication. Memory deduplication has proven to be an efficient technique for reducing server memory requirements, particularly in virtualization environments [7, 8, 29, 31, 32]. Numerous software and hardware techniques have been proposed to enhance the performance and scalability of `ksm`, the widely used memory deduplication mechanism.

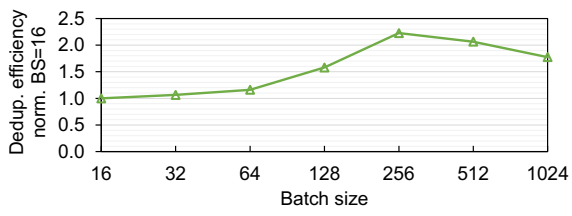


Figure 16: Deduplication efficiency of `Para-ksmC` across different batch sizes, normalized to the efficiency at the batch size of 16, while co-running with `Graph500`.

On the software side, UPM [29] enhances `ksm` by using user-provided hints to identify memory regions likely to contain duplicates, thereby improving accuracy and reducing overhead. XLH [21] extends the memory deduplication process by introducing cross-layer I/O-based hints, enabling earlier identification and exploitation of page-sharing opportunities. CMD [4] classifies pages into different groups and performs comparisons only between pages within the same group, thereby minimizing futile comparisons. However, these software techniques still rely on the CPU for data-plane functions, and thus cannot eliminate CPU overhead. `Para-ksm` enhances `ksm` by introducing greater parallelism, offering an orthogonal approach that can be combined with the aforementioned software techniques.

On the hardware side, PageForge [34], a pioneering design, integrates a dedicated hardware component into the memory controller to execute `memcmp` and leverages the Error Correcting Code (ECC) engine to perform `xxhash` functions. Although PageForge reduces CPU overhead, it requires complex and challenging modifications to memory controller hardware that are infeasible without vendor support. In contrast, `Para-ksm` can operate seamlessly on processors with on-chip DSA accelerators without any hardware modification.

Kernel function offloading. Kernel function offloading has been popular for its success in alleviating CPU workload by shifting intensive OS functions and stacks to specialized hardware accelerators, such as GPUs, FPGAs, SmartNICs (SNICs), and dedicated processors. PCIe-based FPGAs and SNICs stand out as promising candidates for offloading due to their flexible programmability. LineFS [16] offloads distributed system functions, while FlexTOE [33] offloads the TCP stack to SNICs. Pigasus [41] employs FPGA-based SNICs to accelerate the intrusion detection and prevention systems (IDS/IPS), two demanding stateful network functions. Performance-related kernel functions are offloaded to FPGAs in a prior work [1] and hXDP [3] offloads Linux's eXpress Data Path programs written in eBPF to FPGAs.

Specific to memory deduplication, Catalyst [7] employs GPUs to accelerate computationally intensive functions from `ksm`, while STYX [14] harnesses the capabilities of RDMA-capable SNICs to mitigate the overhead of `ksm`. However, as a non-coherent off-chip device accessed over PCIe, STYX introduces longer offloading latency, requires memory pinning,address translation, and involves substantial kernel modifications (∼1300 LoC). In contrast, `Para-ksm` utilizes a coherent on-chip DSA, achieving lower offloading latency without the need for memory pinning, and integrates more easily into the OS (∼300 LoC). Compared to another recent work [13], which only offloads `memcmp` to DSA and uses the CPU to filter comparisons to offset offloading latency, `Para-ksm` achieves lower CPU utilization by offloading both

memcmp and xxhash, and improves deduplication throughput by redesigning ksm and leveraging the batching capability of DSA. Recent work [15] on CXL Type-2 devices demonstrates efficient kernel function offloading with reduced communication overhead enabled by the cache-coherent CXL interconnect. However, the current implementation of genuine CXL Type-2 devices relies on a resource-constrained FPGA for computation, which is less capable than DSA and requires extra programming effort. Furthermore, Para-ksm's software enhancements to ksm are adaptable to existing hardware acceleration techniques for memory deduplication, offering the potential for higher throughput and improved efficiency.

8 Conclusion

In this paper, we first identified the CPU-intensive functions, memcmp and xxhash, in the memory deduplication feature, ksm, and demonstrated the potential of offloading them to the on-chip accelerator DSA to reduce CPU cycle consumption. Second, we proposed DSA-ksm, which directly replaces CPU-based memcmp and xxhash with DSA-based counterparts in ksm. While DSA-ksm significantly reduced the performance degradation on the co-running applications incurred by ksm, it offered notably lower rates of memory deduplication than ksm due to long offloading latency. We found that DSA's batching capability can effectively amortize this latency by enabling memcmp and xxhash to operate on multiple pages per offload. Based on this observation, we lastly proposed Para-ksm, which redesigns ksm to replace its sequential processing of memcmp and xxhash on a single page with parallel processing on multiple pages, enabling the Para-ksm to fully leverage DSA's batching capability. We showed that Para-ksm increased the amount of memory deduplication per CPU cycle by 31-50% compared to ksm while decreasing the performance degradation of co-running applications incurred by ksm from 1.6–5.8× to 1.3–2.7×.

References

[1] Pabudi T Abeyrathne, S Devapriya Dewasurendra, and Dhammika Elkaduwa. Offloading specific performance-related kernel functions into an fpga. In Proceedings of the 2021 IEEE 30th International Symposium on Industrial Electronics (ISIE'21), 2021.

[2] Sean Barker, Timothy Wood, Prashant Shenoy, and Ramesh Sitaraman. An empirical study of memory sharing in virtual machines. In Proceedings of the 2012 USENIX Annual Technical Conference (ATC'12), 2012.

[3] Marco Spaziani Brunella, Giacomo Belocchi, Marco Bonola, Salvatore Pontarelli, Giuseppe Siracusano, Giuseppe Bianchi, Aniello Cammarano, Alessandro Palumbo, Luca Petrucci, and Roberto Bifulco. hxdp: Efficient software packet processing on fpga nics. In Proceedigs of the 14th USENIX Symposium on Operating Systems Design and Implementation (NSDI'20), 2020.

[4] Licheng Chen, Zhipeng Wei, Zehan Cui, Mingyu Chen, Haiyang Pan, and Yungang Bao. Cmd: classification-based memory deduplication through page access characteristics. In Proceedings of the 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE'14), 2014.

[5] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC'10), 2010.

[6] Rong-En Fan, Kai-Wei Chang, Cho-Jui Hsieh, Xiang-Rui Wang, and Chih-Jen Lin. Liblinear: A library for large linear classification. Journal of machine Learning research, 2008.

[7] Anshuj Garg, Debadatta Mishra, and Purushottam Kulkarni. Catalyst: Gpu-assisted rapid memory deduplication in virtualization environments. In Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE'17), 2017.

[8] Yunfei Gu, Yihui Lu, Chentao Wu, Jie Li, and Minyi Guo. Cksm: An efficient memory deduplication method for container-based cloud computing systems. In Proceedings of the 2024 IEEE International Parallel and Distributed Processing Symposium (IPDPS'24), 2024.

[9] Intel. Intel® FPGA SmartNIC N6000-PL Platform. https://www.intel.com/content/www/us/en/products/details/fpga/platforms/smartnic/n6000-pl-platform.html, accessed in 2024.

[10] Intel Corporation. Intel Data Accelerator Driver. https://github.com/intel/idxd, accessed in 2024.

[11] Intel Corporation. Intel Data Streaming Accelerator. https://www.intel.com/content/www/us/en/products/docs/accelerator-engines/data-streaming-accelerator.html, accessed in 2024.

[12] Intel Corporation. Introduction to Cache Allocation Technology in the Intel® Xeon® Processor E5 v4 Family. https://www.intel.com/content/www/us/en/developer/articles/technical/introduction-to-cache-allocation-technology.html, accessed in 2024.

[13] Houxiang Ji, Minho Kim, Seonmu Oh, Daehoon Kim, and Nam Sung Kim. Cooperative memory deduplication with intel data streaming accelerator. IEEE Computer Architecture Letters, 2025.

[14] Houxiang Ji, Mark Mansi, Yan Sun, Yifan Yuan, Jinghan Huang, Reese Kuper, Michael M. Swift, and Nam Sung Kim. STYX: Exploiting SmartNIC capability to reduce datacenter memory tax. In Proceedings of the 2023 USENIX Annual Technical Conference (ATC'23), 2023.

[15] Houxiang Ji, Srikar Vanavasam, Yang Zhou, Qirong Xia, Jinghan Huang, Yifan Yuan, Ren Wang, Pekon Gupta, Bhushan Chitlur, Ipoom Jeong, and Nam Sung Kim. Demystifying a cxl type-2 device: A heterogeneous cooperative computing perspective. In Proceedings of the 2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO'24), 2024.

[16] Jongyul Kim, Insu Jang, Waleed Reda, Jaeseong Im, Marco Canini, Dejan Kostić, Youngjin Kwon, Simon Peter, and Emmett Witchel. Linefs: Efficient smartnic offload of a distributed file system with pipeline parallelism. In Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP'21), 2021.

[17] Reese Kuper, Ipoom Jeong, Yifan Yuan, Ren Wang, Narayan Ranganathan, Nikhil Rao, Jiayu Hu, Sanjay Kumar, Philip Lantz, and Nam Sung Kim. A quantitative analysis and guidelines of data streaming accelerator in modern intel xeon scalable processors. In Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'24), 2024.

[18] Taehyung Lee, Sumit Kumar Monga, Changwoo Min, and Young Ik Eom. Memtis: Efficient memory tiering with dynamic page classification and page size determination. In Proceedings of the 29th Symposium on Operating Systems Principles (SOSP'23), 2023.

[19] Mellanox Technologies. Mellanox Innova-2 Flex Open Programmable SmartNIC. https://network.nvidia.com/files/doc-2020/pb-innova-2-flex.pdf, accessed in 2024.

[20] Konrad Miller, Fabian Franz, Thorsten Groeninger, Marc Rittinghaus, Marius Hillenbrand, and Frank Bellosa. Ksm++: Using i/o-based hints to make memory-deduplication scanners more efficient. In Proceedings of the ASPLOS Workshop on Runtime Environments, Systems, Layering and Virtualized Environments (RESoLVE'12), 2012.

[21] Konrad Miller, Fabian Franz, Marc Rittinghaus, Marius Hillenbrand, and Frank Bellosa. XLH: More effective memory deduplication scanners through cross-layer hints. In Proceedings of the 2013 USENIX Annual Technical Conference (ATC'13), 2013.

[22] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. Scaling memcache at facebook. In Proceedigs of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI'13), 2013.

[23] NVIDIA. ConnectX NICs. https://www.nvidia.com/en-us/networking/ethernet-adapters/, accessed in 2024.

[24] NVIDIA. NVIDIA BlueField-3 Networking Platform. https://resources.nvidia.com/en-us-accelerated-networking-resource-library/datasheet-nvidia-bluefield, accessed in 2024.

[25] NVIDIA. NVIDIA Ethernet SuperNICs. https://www.nvidia.com/en-us/networking/products/ethernet/supernic/, accessed in 2024.

[26] NVIDIA Corporation. NVIDIA BlueField-3 DPU. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/documents/datasheet-nvidia-bluefield-3-dpu.pdf, accessed in 2023.

[27] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. Shenango: Achieving high cpu efficiency for latency-sensitive datacenter workloads. In Proceedigs of the 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI'19), 2019.

[28] Patel, Dylan and Xie, Myron and Chu, Wega. Going Vertical: Gate All Around, 3D DRAM, 3D NAND – Kokusai Electric IPO. https://semianalysis.com/2023/10/15/going-vertical-gate-all-around-3d/, accessed in 2024.

[29] Wei Qiu, Marcin Copik, Yun Wang, Alexandru Calotoiu, and Torsten Hoefler. User-guided page merging for memory deduplication in serverless systems. In Proceedings of the 2023 IEEE International Conference on Big Data (BigData'23), 2023.

[30] Redis Labs. Redis: The Real-Time Data Platform. https://redis.io, accessed in 2024.

[31] Divyanshu Saxena, Tao Ji, Arjun Singhvi, Junaid Khalid, and Aditya Akella. Memory deduplication for serverless computing with medes. In Proceedings of

the 17th European Conference on Computer Systems (Eurosys'22), 2022.

[32] Prateek Sharma and Purushottam Kulkarni. Singleton: system-wide page deduplication in virtual environments. In Proceedings of the 21st international symposium on High-Performance Parallel and Distributed Computing (HPDC'12), 2012.

[33] Rajath Shashidhara, Tim Stamler, Antoine Kaufmann, and Simon Peter. {FlexTOE}: Flexible {TCP} offload with {Fine-Grained} parallelism. In Proceedings of the 19th USENIX Symposium on Networked Systems Design and Implementation (NSDI'22), 2022.

[34] Dimitrios Skarlatos, Nam Sung Kim, and Josep Torrellas. Pageforge: a near-memory content-aware page-merging architecture. In Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'17), 2017.

[35] Stefan Roesch. Kernel Samepage Merging (KSM) Usage at Meta and future Improvements. https://lpc.events/event/17/contributions/1625/attachments/1320/2649/KSM.pdf, accessed in 2024.

[36] Toyotaro Suzumura, Koji Ueno, Hitoshi Sato, Katsuki Fujisawa, and Satoshi Matsuoka. Performance characteristics of graph500 on large-scale distributed environment. In Proceedings of the 2011 IEEE International Symposium on Workload Characterization (IISWC'11), 2011.

[37] The Kernel Development Community. Kernel Samepage Merging. https://docs.kernel.org/admin-guide/mm/ksm.html, accessed in 2024.

[38] Johannes Weiner, Niket Agarwal, Dan Schatzberg, Leon Yang, Hao Wang, Blaise Sanouillet, Bikash Sharma, Tejun Heo, Mayank Jain, Chunqiang Tang, and Dimitrios Skarlatos. Tmo: Transparent memory offloading in datacenters. In Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'22), 2022.

[39] Daniel Whiteson. SUSY. UCI Machine Learning Repository, accessed in 2024. DOI: https://doi.org/10.24432/C54606.

[40] Yifan Yuan, Ren Wang, Narayan Ranganathan, Nikhil Rao, Sanjay Kumar, Philip Lantz, Vivekananthan Sanjeepan, Jorge Cabrera, Atul Kwatra, Rajesh Sankaran, Ipoom Jeong, and Nam Sung Kim. Intel accelerators ecosystem: An soc-oriented perspective: Industry product. In Proceedings of the 2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA'24), 2024.

[41] Zhipeng Zhao, Hugo Sadok, Nirav Atre, James C Hoe, Vyas Sekar, and Justine Sherry. Achieving 100gbps intrusion prevention on a single server. In Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI'20), 2020.

A Artifact Appendix

Abstract

This artifact provides instructions for reproducing the key experimental results reported in the paper. Specifically, it covers two key sets of experiments: (1) application performance degradation on systems running `CPU-ksm`, `DSA-ksm`, and `Para-ksmC`, normalized to a system without `ksm` (`no-ksm`) (Figure 12); and (2) memory deduplication performance, including both deduplication effectiveness (Figure 13) and deduplication efficiency (Figure 14).

Scope

The artifact enables validation of two main claims: (i) offloading data-plane functions of `ksm` to DSA (`DSA-ksm` and `Para-ksm`) significantly reduces performance degradation of co-running applications; and (ii) after an elaborated redesign of `ksm`, `Para-ksm` leverages DSA's batching capability by processing multiple pages per offload, effectively amortizing offload latency. Compared to vanilla `ksm`, `Para-ksm` achieves comparable memory deduplication effectiveness while significantly improving deduplication efficiency.

Contents

The artifact includes: (i) a `README.md` file containing instructions for building and installing the kernel with `Para-ksm` support, and for running the experiments using provided scripts; (ii) the `linux-paraksm` directory containing the Linux kernel source code modified with the `Para-ksm` implementation; and (iii) the `scripts` directory containing scripts to reproduce experimental results and generate the corresponding figures.

Hosting

The artifact is hosted on GitHub at https://github.com/ece-fast-lab/ATC-2025-Paraksm.git, on the `main` branch. Users can obtain the artifact by cloning the repository and checking out the latest commit (`98716ac`) to reproduce the results.

Requirements

The experiments require a server with 4th-generation (or newer) Intel Xeon Scalable Processor equipped with DSA accelerator. The `Para-ksm` implementation is based on Linux kernel version 6.2.15 and has been tested on Ubuntu 18.04.6 LTS. QEMU-KVM version 2.11.1 is used for virtual machine management, with guest VMs running Ubuntu Cloud 22.04.

A.1 Installation

Clone the artifact repository as follows:

```
$ git clone https://github.com/ece-fast-lab/ATC
    -2025-Paraksm.git
```

The Linux kernel source code with `Para-ksm` implementation can be found in `linux-paraksm` directory. Follow the instructions in `README.md` to build and install the kernel. A system reboot is required to use the compiled kernel.

A.2 Experiment workflow

After installing the kernel, experiments can be executed as follows:

```
$ cd /scripts/fig12-14/
$ bash run.sh
```

The `run.sh` script executes all benchmarks by default; specific experiments can be selected by commenting or uncommenting lines within the script. All generated results will be automatically saved. The figures (Figure 12, 13 and 14) can be plotted and saved as PNG files by running `plot.sh` in the `/scripts/fig12-14/` directory. Additional details are provided in the `README.md` file in the root directory.

DSA-2LM: A CPU-Free Tiered Memory Architecture with Intel DSA

Ruili Liu[♠◇*], Teng Ma[♡*+], Mingxing Zhang[♠+], Jialiang Huang[♠♡], Yingdi Shan[♠], Zheng Liu[♡]
Lingfeng Xiang[♣], Zhen Lin[♣], Hui Lu[♣], Jia Rao[♣], Kang Chen[♠], Yongwei Wu[♠]
[♠]Tsinghua University, [♡]Alibaba Group
[♣]The University of Texas at Arlington, [◇]University of Electronic Science and Technology of China

Abstract

Tiered Memory is critical to manage heterogeneous memory devices, such as Persistent Memory or CXL Memory. Existing works make difficult trade-offs between optimal data placement and costly data movement. With the advent of Intel Data Streaming Accelerator (DSA), a CPU-free hardware to move data between memory regions, data movement can be up to 4× faster than a single CPU core. However, the fine memory movement granularity in Linux kernel undermines the potential performance improvement. To this end, we have developed DSA-2LM, a new tiered memory system that adaptively integrates DSA into page migration. The proposed framework integrates fast memory migration workflow and adaptable concurrent data paths with well-tuned DSA configurations. Experimental results show that, compared to three representative tiered memory works: MEMTIS, TPP and NOMAD, DSA-2LM can achieve 20%, 30% and 16% performance improvement under real-world applications.

1 Introduction

Data-intensive applications such as data processing, machine learning and graph processing are experiencing rising memory demands. It couples with the increasing cost of DRAM and limited DRAM capacity have made memory a major expense. Non-DRAM memory devices, such as Compute Express Link (CXL) [33, 40, 50]/Non-volatile Memory (NVM) [37], appear as a larger memory tier with a low per GB price. However, these technologies expose higher latencies than main memory, potentially degrading performance if data is sub-optimally placed in memory hierarchy.

To place hot/cold data in the correct fast/slow tier, the architecture of tiered memory involves three primary stages: detecting hot data, selecting data to be migrated, and executing the data migration. NOMAD [44] presents that a significant amount of time is consumed by page copying during the memory migration process. Despite the implementation of MEMTIS [29], which effectively mitigates page copying using histogram algorithm [29], the measured migration time spent on copying remains around 49%. One solution is to enhance the accuracy of the data hotness detection. Nonetheless, achieving high precision is associated with substantial CPU overhead. Even when tasks are offloaded to hardware solutions such as processor event-based sampling (Intel PEBS [48]), the performance degradation is still recorded at 20% [44]. This is attributed to the increased frequency of migration, which consumes CPU resources (11% CPU overhead in Graph500 [34]) and occupies additional bandwidth, interfering with applications, negatively impacting overall end-to-end performance.

A promising way to reduce CPU bottleneck is to offload overhead to hardware accelerators such as the Intel Data Streaming Accelerator (DSA). DSA can transfer data between memory regions without involving CPU and now ships as a standard feature in 4/5th generation Intel Xeon processors [24]. Its benefits are twofold. First, each page-copy request requires submitting only a small descriptor to work queue (WQ), which takes only a few cycles to complete [10]. This eliminates most of the CPU overhead, even in systems that rely on CXL and CPU interactions. Second, DSA provides a bandwidth of about 32 GB/s per device, comparable to four CPU cores [19]. Modern servers typically include four DSA devices per socket, delivering a combined bandwidth of up to 110 GB/s. Hence, DSA is well-suited for tiered memory.

However, using DSA effectively requires careful design. First, current kernel support for DSA relies on the traditional Direct Memory Access (DMA) interface, introducing large overhead for buffer preparation and (un)mapping. It is inadequate for fine-grained page movement. Second, DSA outperforms CPU-based copying only when there is sufficient concurrency to hide high invocation latency, so asynchronous task partitioning and dispatching must be carefully planned to maximize performance by utilizing the multiple cores provided by the CPU socket. Finally, DSA performance is sensitive to many configurations and requires tuning. We address this by developing an adaptive algorithm for different page types, with dynamic batch sizes and WQ settings.

Based on DSA hardware, we design DSA-2LM (DSA-based two-level memory), a new tiered memory system in Linux kernel with flexible interfaces (§3.2). While inheriting hotness detection and page migration logic via `kmigrated` of MEMTIS [29], DSA-2LM introduces several DSA-specific designs. These designs contain fast memory migrating workflow (§3.3) and adaptable concurrent data migration (§3.4) with well-tuned batch size and number of WQs. It avoids

*: Equal Contributions. +: Corresponding Authors. This work is supported by National Key Research & Development Program of China (2022YFB4502004), Natural Science Foundation of China (62141216, 92467102), Tsinghua University Initiative Scientific Research Program, and Alibaba Group through Alibaba Innovative Research Program.

CPU involvement in the critical data copying path, enabling aggressive CPU-free migrations.

To evaluate the performance of DSA-2LM framework, we choose five widely used applications and use traces of industry workloads. The evaluation environment is a commercial pre-market CXL system. The DSA-based acceleration of page copying allows the system to take full advantage of page migration, thereby improving its responsiveness to changes in memory access patterns. Experimental results show that, compared to three representative page management schemes: MEMTIS [29], TPP [33] and NOMAD [36], DSA-2LM achieves up to 81.7%, 52.9% and 15.6% performance improvement under real-world applications. DSA-2LM is available at https://github.com/madsys-dev/DSA-2LM.

2 Motivation and Challenges

2.1 Tiered Memory and Applications

The memory demand of data-intensive applications such as graph processing and machine learning continues to increase [28]. In Microsoft Azure, memory costs ∼50% of the total server costs [30]. ML models are rapidly growing, and are expected to grow 50× in the next five years [33]. With rapid data growth, memory hierarchy should be redesigned.

Tiered memory fulfills the requirements of data-intensive applications. It uses a two-level hierarchy with a fast, expensive tier and a slow, cost-effective tier [15]. It frequently accesses data in the fast tier, moves less-accessed data to slow tier, and maintains an exclusive data placement. Emerging memory technologies such as high bandwidth memory, CXL memory [8, 32, 40, 43, 51], NVM [47], and NVMe SSDs introduce solutions that challenge the traditional memory hierarchy. For example, CXL [8] integrates various memory devices under a single interface, offering larger storage capacities and narrowing the performance gap. It guided researchers to explore state-of-the-art systems like TPP [33], MEMTIS [29], NOMAD [44], and Nimble [46], each designed to optimize data access across different memory tiers.

2.2 Tiered Memory Top-down

Trade-off between CPU and hotness detection algorithm. Both hardware-based and software-based access tracking involve CPU usage [35, 39], a critical performance bottleneck for tiered memory systems. There is no one-fit-all hotness detection algorithm [45]. Achieving high precision and finer monitoring granularity for detecting hot pages requires tiered memory systems to either shorten the monitoring interval or increase the number of counters, both of which demand additional CPU resources.

Interference between migration and applications. To adapt to dynamically changing workloads, tiered memory requires to frequently migrate data between the fast and slow tiers, leading to data copying overhead. The extra CPU utilization for monitoring and migration will interfere with application performance [23], thus a well-designed tiered memory system may not be the best choice. As shown in Figure 1, increasing the sampling interval results in higher CPU utilization and more page migrations, which can degrade application performance eventually.

Page copy is still a problem for tiered memory. One of the critical challenges is the significant CPU overhead associated with page migration and memory copying. This issue has been explored in previous tiered memory works [27, 29, 44]. Nimble reports 30% in the page migration phase. Our empirical tests further highlight this challenge, particularly in scenarios involving mixed workloads (e.g., co-location [17]) and hot start (e.g., serverless [31]). The core reason for this overhead is the CPU's involvement in memory copying, which leads to CPU stalls. To address this, we explore solutions such as offloading memory copy, which could alleviate the CPU's burden and improve system performance.

Figure 2 shows the function sampling results of kernel migrating process captured by perf [2] on MEMTIS. The proportion of migrating pages is 72.87%, while getting page info from index (i.e., `get_pginfo_idx`, which involves kernel's reverse mapping mechanism and update page access statistics) is 29.82% and memory copying (`migrate_page_list`) is 34.43%. About 73.51% of processor cycles in migrating pages are spent on copying pages.

Huge Pages are commonly found in tiered memory. Transparent Huge Page (THP) is utilized to enhance performance in applications [38], such as cloud [20] and database [21]. Most tiered memory works enable THP, so we measure the processing of TPP. Results show that Huge Pages will be the majority if THP [3] is enabled. For PageRank [22] workload, during the demotion, there are 1249067 4KB Pages and 8670 2MB Huge Pages. Thus Huge Pages account for 78.0% memory.

2.3 Observations of DSA

DSA [24] in Intel 4/5th (Sapphire/Granite Rapids) Xeon CPUs can enhance data copy and transformation. DSA hardware comprises interfaces for host communication, work queues (WQs) for descriptor storage, processing engines (PEs) for task execution, and arbiters for quality of service [25]. DSA devices appear as single root complex integrated endpoints, compatible with PCIe. Users submit 64B descriptors via MMIO portals; these include operation details like operation type, address, and batching list. Descriptors are stored in WQs, which can be either a dedicated WQ for single-client use or a shared WQ for multiple clients without synchronization.

DSA's address translation cache interacts with the IOMMU, supporting coherent shared memory with CPU cores, eliminating the need for memory pinning. The software architecture includes the Intel Data Accelerator Driver (IDXD) [7] for DSA

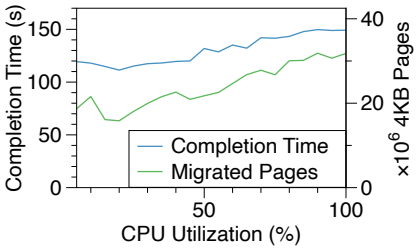


Figure 1: Analysis of hotness detection.

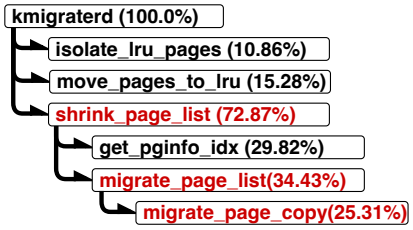


Figure 2: Profiling of migrate process.

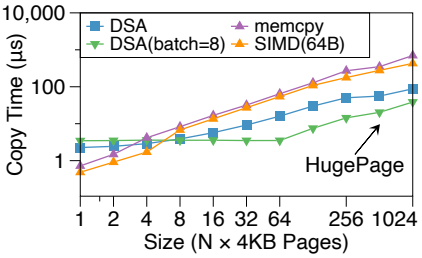


Figure 3: DSA/Memcpy comparisons.

management [19]. Applications exploit `libaccel-config` API to use DSA in user space, and IDXD allows WQ portal address exposed to user space via `mmap`.

The performance of DSA is higher than CPU. While copying data between local/remote NUMA nodes, a DSA can attain a bandwidth of up to 32 GBps, whereas an individual CPU core achieves a maximum bandwidth of only 9 GBps. Moreover, the time required for data copying with a DSA is consistently less than that of a CPU when the size of the data exceeds 32 KB. Furthermore, memory copying increases CPU utilization, leading to interference with ongoing tasks due to significant contention for CPU resources [42].

Not all memory copy sizes benefit from DSA. Figure 3 shows that using DSA performs higher performance than CPU copying when the transfer size is above 32 KB, where each descriptor is submitted and completed before sending another. The submitted descriptors are processed in a pipelined fashion [25], so CPU copying is faster than DSA in small transfer size (< 32 KB). As the transfer size increases, the submission overhead of DSA can be disregarded, thereby showcasing its performance advantage compared to CPU copying.

Number of WQs and batch size of DSA. Figure 4/5 shows that the number of WQs significantly influences maximal throughput. The number of WQs is positively correlated with increased throughput. The throughput attains its maximum when configured with four WQs. Users should carefully consider transfer size, as it affects scalability differently.

The DSA-facilitated batching of descriptors serves to reduce the cost associated with offloading. As shown in Figure 6, the performance improvement plateaus as the batch size reaches 8. Additionally, DSA enhances throughput in a non-linear manner, and it depends on the transfer size.

2.4 DSA v.s., DMA

DMA and DSA operate at different abstraction levels. DMA provides a general interface that various underlying engines, such as DSA or Host Bridge can support, and these interfaces abstract memory operations, letting developers work without dealing with complex hardware drivers [49]. Directly manipulating DSA driver functions (i.e., IDXD) offers less abstraction. However, this approach balances ease of use with advanced functionality and performance. Utilizing DSA directly, without dependence on the DMA interface, presents two principal advantages:

Elimination of memory pinning overhead. Most devices using DMA cannot tolerate page faults, requiring guest buffers to be pinned in host memory and mapped in the IOMMU page table before DMA access. Consequently, DMA becomes problematic in tiered memory scenarios, a major limitation, especially when hosting multiple virtual machines. In contrast, DSA supports the use of either physical or virtual addresses. The use of virtual addresses that are shared with processes running on the CPU is called shared virtual memory. Therefore, address translation does not require IOMMU page table mapping or memory pinning.

Significant performance improvement. In the submitting descriptor phase, DSA shows better performance than DMA. In DSA-2LM, the DSA descriptors are designed as per-cpu variables, so there is no dynamic descriptor (de)allocation. In contrast, using DMA interfaces should wait for the descriptor to be ready. We measure single copy operation for one 4KB page shown in Figure 7. DMA operation spends an extra 88*ns* in total to (un)map DMA buffer, prepare memory copy and submit DMA request. In contrast, only 6*ns* time is required to submit DSA descriptor. The major expensive phase is waiting for the completion of each DMA/DSA operation. According to evaluations in Section 2.3, utilizing features such as DSA batching and multi-channel DSA utilization further boost performance compared to the previous generation DMA engines such as Crystal Beach DMA (CBDMA) and Intel I/O Acceleration Technology. DSA provides 2.1× greater throughput on average and up to 8× higher throughput than CBDMA [25].

2.5 Challenges and Design Principles

Above all, there are three primary challenges: (1) How to directly use DSA in kernel-space for migrating pages and cooperating with tiered memory system (2) How to let both 4KB Page and 2MB Huge Page enjoy the benefit of DSA, and (3) How to dynamically choose appropriate batch size and number of WQs for each descriptor to maximum performance without harming application performance. Thus, we should redesign the tiered memory system considering the features of DSA from three aspects.

Use DSA directly for page migration in kernel space. Utilizing the CPU for page migration will increase CPU footprint, thereby affecting the performance. Section 2.3 shows that using DSA can alleviate CPU overhead and exhibit superior performance in page migration. For in-kernel program-

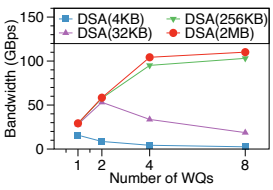


Figure 4: The effects of different number of WQs.

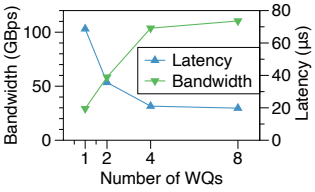


Figure 5: The bandwidth and latency of multi-WQ.

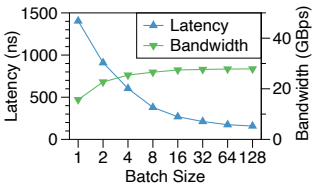


Figure 6: The effects of batch size.

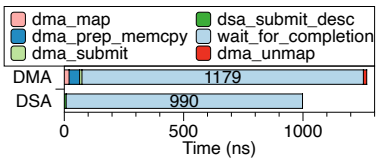


Figure 7: The performance breakdown of DSA and DMA.

ming, we bypass DMA interface and directly use physical address in submitted descriptors. Our evaluation confirms that using DSA in kernel space does not trigger any page fault.

Use asynchronous mode of DSA properly. Handling hardware interrupts in user space can be tricky. Intel officially recommends using `UMONITOR/UMWAIT` instructions after submitting descriptors to reduce power consumption of the CPU core [11, 25, 49]. However, this approach does not achieve true asynchronous mode, as the CPU cannot schedule other tasks while in a low-power state. Intel DTO [1] adopts the `sched_yield` syscall to proactively relinquish the CPU to avoid busy polling, but this sacrifices some responsiveness. Hence, such user-space programming models are not applicable in kernel space. DSA-2LM must carefully design and utilize the advantage of handling interrupts in kernel space, achieving genuine and efficient asynchronous waiting.

Aggregate data path for both small/large size page migration. Section 2.3 presents that only large-sized page migration can benefit from DSA. Naive design is using two separate data paths for page migration. For 4KB Pages, DSA-2LM uses the original `copy_page` method. For 2MB HugePages, DSA-2LM uses DSA-oriented copying functions. A separate data path means extra overhead and small-sized pages cannot benefit from DSA. We need a new data path with adaptable algorithm to aggregate small and large pages.

Carefully tuning parameters for DSA. As we discuss in Section 2.3, using different values of batch size and number of WQs will significantly influence the copying rate. For a page list to be migrated, we should tune the value of parameters to maximize performance. So we need to know the performance changes and adjust the parameters dynamically according to the page number and each page size in the page list.

3 Design of DSA-2LM

3.1 Architecture Overall

DSA-2LM consists of three steps: track page access using hardware performance counter, choose hotness pages, and migrate pages in the background with DSA (Figure 8). (1) DSA-2LM uses the page histogram mechanism, which is the same as MEMTIS [29], to record the access of pages. It samples memory access in the page granularity using PEBS, and updates the hotness distribution of all allocated pages periodically. (2) DSA-2LM ensures that the size of the hot set is

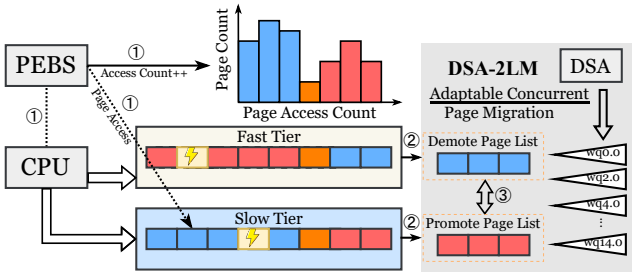


Figure 8: DSA-2LM overall.

maintained near the capacity of the fast tier, thereby enabling the fast tier to accommodate all hot pages. It chooses hot/cold pages according to its access count higher/lower than the thresholds T_{hot} and T_{cold}. These thresholds are dynamically adjusted to fit into the total pages. Then selected hot/cold pages are moved to promotion/demotion page list. (3) The per-node kernel threads responsible for migration perform the migration operation including demotion and promotion. Such a kernel thread will be woken up periodically in a time interval (e.g., 200 ms). DSA-2LM decides to migrate pages according to the available space in the fast tier. It detects the status of DSA, and it migrates the page list using available DSA devices in the kernel space. DSA-2LM uses an adaptable concurrent migration algorithm to copy page lists that include 4KB Pages and 2MB Huge Pages.

Compared to MEMTIS, DSA-2LM inheriting its hotness detection algorithm, page placement strategies and relying on the `kmigrated` kernel thread for page migration. Additionally, DSA-2LM improves MEMTIS's sampling algorithm to consistently utilize the most recent samples, thus avoiding using outdated samples. Crucially, DSA-2LM introduces several key improvements that are specific to DSA. Inspired by Nimble [46], DSA-2LM introduces a concurrent page migration strategy. DSA-2LM applies the DSA adaptive migration algorithm to optimize page copying. Because DSA is CPU-free, DSA-2LM shortens the wake-up interval of the kmigrated thread, adopting a more aggressive migration strategy without incurring extra overhead.

3.2 Implementation in Kernel

An extra moudle is implemented under the `misc/exp` directory, which exposes two new APIs: `dsa_multi_copy_pages` and `dsa_copy_page_lists`. The functionality of DSA-2LM

can be toggled via a sysAPI interface, and various statistical parameters are presented through `procfs` for monitoring purposes. The state-of-the-art tiered memory systems like MEMTIS [29] and TPP [33] are based on the 5.x kernel. To ensure compatibility with existing systems, the overall implementation is based on Linux kernel version 5.13/5.15. The modifications to the memory management subsystem and DSA acceleration module implementation comprise approximately 2K LoC, while the backporting of the IDXD and IOMMU drivers from kernel 6.4 introduces around 8K LoC.

3.3 Migration Workflow

DSA-2LM creates a migration kernel thread for each memory tier to handle promotion and demotion operations in the background. The system maintains two lists: a promotion list containing hot pages for the slow tier and a demotion list containing cold pages for the fast tier.

Whenever processing a memory access sample, DSA-2LM compares the page's hotness factor (H_i) to the threshold T_{hot}. If the page is determined as hot, it is moved to the promotion list. Also, the migration kernel thread periodically performs cooling by halving a page's access count for every N sampled events. This process can cause some pages to transition to cold states, in which case they are moved to the demotion list.

In the slow tier, the migration thread checks for hot pages and determines if there is available space in the fast tier. If both conditions are met, it promotes hot pages to the fast tier. Conversely, when the available memory in the fast tier drops below a predefined free-space threshold (set at 3% of the fast tier's total size for future page allocations and promotions), the demotion operation is initiated.

This demotion process involves selecting victim pages from the fast tier. Initially, cold pages ($H_i \leq T_{\text{cold}}$) are demoted back to the slow tier. It stops until sufficient free space is reclaimed. Otherwise, cold pages are demoted until the necessary free space is achieved. This strategy allows DSA-2LM to retain as many cold pages as possible within the fast tier.

3.4 Adaptable Concurrent Migration

After promotion/demotion page list is prepared, the migration thread uses DSA to copy pages containing mixed 4KB Pages and 2MB Huge Pages.

Before migrating (see Algorithm 1), the total number of available WQs must be initialized. `limit_chans` specifies the total number of available DSAs used for page migration and should be configured properly according to configurations (typically a power-of-two value with DSA-2LM defaulting to 8). In DSA-2LM, each DSA device enables four PEs and one WQ, so DSA device and WQ are one-to-one. DSA-2LM enumerates each WQ, updating the count of available WQs when DSA-2LM is initialized first or `limit_chans`/`dsa_state` are updated through sysfs API.

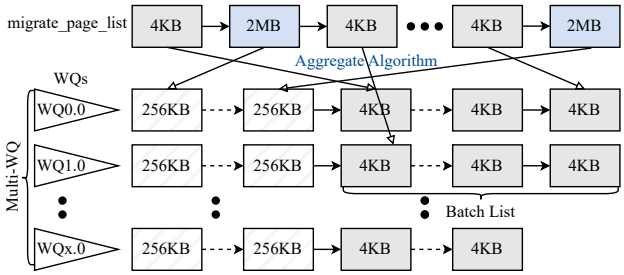


Figure 9: Workflow of adaptable page migration.

Algorithm 1 DSA Migrate Page List

Input: *page_list*: 4KB and 2MB mixed page pair lists to be migrated (A page pair include source/destination page)

1: get `idxd_desc/dsa_hw_desc/dsa_completion_record/completion` pointers from per-cpu `struct` variables
2: **repeat**
3: *nr_base_pages* ← 0
4: **for** each unsolved *page_pair* in *page_list* **do**
5: **if** *page_pair* is THP/HugePage **then**
6: *transfer_size* ← 2MB / `limit_chans`
7: **for** each available WQ *i* **do**
8: Prepare a `memmove` DSA descriptor for (*page_pair*, *transfer_size*, WQ *i*)
9: Submit a DSA descriptor
10: **else**
11: *nr_base_pages* ← *nr_base_pages* + 1
12: *batch_size* ← *nr_base_pages* / `limit_chans`
13: *batch_list* ← ∅
14: **for** each unsolved 4KB *page_pair* in *page_list* **do**
15: Prepare a `memmove` DSA descriptor *desc* for *page_pair*
16: *batch_list* ← *batch_list* ∪ *desc*
17: **if** number of *batch_list* fits *batch_size* **then**
18: Choose a WQ *t* in round robin
19: Prepare a `batch` DSA descriptor for (*batch_list*, *batch_size*, WQ *t*);
20: Submit a DSA descriptor
21: *batch_list* ← ∅
22: **for** each available WQ *i* **do**
23: Yield and wait for WQ *i* completion
24: **until** each page pair in *page_list* has been migrated

Aggregate data path. When the page order is greater than 3 (i.e., 32 KB), utilizing DSA for copying pages achieves lower latency compared to using the CPU. For non-contiguous pages, we employ batch processing to reduce additional overhead. In case of contiguous pages, such as 2MB Huge Pages, we leverage multi-WQ parallel copying to enhance efficiency.

Combining these strategies altogether, Figure 8 and Algorithm 1 show the workflow of handling the promotion/demotion page list. In the first pass, DSA-2LM picks the 2MB Page and splits it into `limit_chans` (e.g., 8) subpages whose size is 2MB / `limit_chans` (e.g., 256 KB), and assigns subpages to each WQ in round robin. In the second pass, DSA-2LM calculates the average transfer size (i.e., `batch_size`) for each WQ based on the remaining 4KB Pages, and distributes them into batch lists in each WQ. According to the "shortboard effect", DSA-2LM balances the total transfer size submitted to each WQ so that the difference among WQs does

not exceed the size of a single 4KB page. If the `batch_size` is less than two in a corner case, DSA-2LM directly submits the remaining 4KB pages to each WQ.

Concurrent migration with both batch and multi-WQ. According to `limit_chans` of batch lists, DSA-2LM initializes the necessary DSA data structures for processing memory copy, including `idxd_desc`, `dsa_hw_desc`, and `dsa_completion_record`. These data structures are created as per-cpu global variables in the initialization phase, to avoid the preparation overhead that exists in DMA processing.

Subsequently, for each WQ, DSA-2LM submits DSA descriptors, ensuring any errors encountered are handled appropriately. Since adopting asynchronous mode, DSA-2LM leverages the Linux kernel's `completion` mechanism to await completion of copy requests. Once the `status` field in the `dsa_completion_record` is set by hardware, it triggers an MSI-X interrupt that awakens the `idxd_wq_thread`, which then scans all pending DSA descriptors. For each completed descriptor, `idxd_wq_thread` invokes the corresponding callback function, thereby signaling the `completion`. Consequently, the migration thread is unblocked from `wait_for_completion_timeout` and proceeds with the subsequent operations. DSA-2LM sets a timeout for each submitted descriptor, and returns fails if it occurs. This workflow significantly enhances the efficiency and speed of page copying operations in the kernel.

4 Evaluations

4.1 Evaluation Configurations

Hardware configuration. Our testbed is a dual-socket server equipped with 4th Gen Xeon Platinum CPU (48×2 physical cores), where each socket has 1 TB (256 GB×4) DDR5 DRAM. One of the sockets is connected to a 64 GB ASIC-based Montage Technology CXL device. We disabled Intel Hyper-Threading and fixed the CPU frequency to 3.2 GHz. We set CPU affinity for workloads and used only 32 cores of one socket. Similar to prior works, we used a DRAM NUMA node (load/store: 112 ns) and a CPU-less node with CXL memory (load/store: ∼300 ns), and THP [16] is enabled. DSA-2LM can also support two NUMA nodes without CXL. We configured the CXL-attached memory to `system-ram` mode. It allows load/store access to CXL memory in kernel space.

Software configuration. We use different methods to equitably constrain the fast tier. For AutoNUMA, TPP and NOMAD, we changed the kernel boot argument (memmap GRUB option [5]) to limit the fast tier size. The fast tier size is set as 16/32 GB. For MEMTIS, we used a memory `cgroup` interface to control the size of the fast tier. The ratios of fast/slow tier memory size were established from 1:2 to 1:16. The proportion of the fast tier size was adjusted from 33% (1/3) to 5.9% (1/17) of the resident set size (RSS) for each benchmark.

Workload	RSS	Description
Graph500	68.0 GB	Graph generation and breadth-first search of some random vertices [18].
PageRank	12.3 GB	Compute the PageRank score with an iterative method [18]. (Twitter dataset [26])
XSBench	63.4 GB	Monte Carlo neutron transport algorithm [41].
BTree	38.3 GB	In-memory index lookup benchmark [4].
Pandas	20.2∼92.6 GB	Data processing with six different queries [12].

Table 1: Workload characteristics.

4.2 Performance of Copy Pages

Two distinct workloads A/B, are employed to assess the impacts of our adaptable copying algorithm. Workload A comprises 1023 4KB pages and a single 2MB huge page, with the smaller 4KB pages constituting the bulk of the workload. Conversely, workload B is characterized by a high page intensity, consisting of 1000 4KB pages and 24 2MB huge pages.

As shown in Figure 10, we measure the bandwidth of continuous page migration with three strategy: 1) CPU: only use CPU to copy pages; 2) DSA (raw): migrating 4KB page with CPU and 2MB page with DSA; and 3) DSA (opt): use adaptable concurrent migration proposed in Section 3.4. For workload B, the bandwidth of DSA-2LM (106.3 GBps) is 14.38/2.19× higher than CPU/DSA (raw) implementations. We conclude DSA-2LM can easily saturate the bandwidth (the peak bandwidth in our evaluation platform is ∼110 GBps).

4.3 Performance Breakdown

We repeat to capture the result of function samples in kernel mode by perf and Figure 11 shows the results. The overall proportion of migrating pages in is dropped from 39.0% to 4.24%, which is only 2.51% compared with the original overhead. This proves that our approach can effectively reduce CPU overhead in the migration data path.

Figure 12 shows copying duration in real-time during the execution of Graph500. The application completes in 142s, with a total copying time of 14.9s when employing CPU-based migration. Conversely, using DSA-based methods DSA-2LM for page migration reduces the total copying time to 1.39s, representing only 9.3% of the baseline duration. In the results reported in DSA-2LM, the copying time accounts for only 0.97% of the execution time, whereas it comprises 10.5% in the context of original tiered memory.

4.4 Real-world Applications

We evaluate five representative workloads in Table 1. Since MEMTIS and TPP limit fast-tier memory size in different ways, it is difficult to reserve the same fast-tier memory size. For a fair comparison, we presented them in separate figures.

The upper part of Figure 13 compares DSA-2LM with TPP, NOMAD, and AutoNUMA. We successfully applied DSA to other state-of-the-art tiered memory systems using similar approaches, denoted as TPP+, NOMAD+, and AutoNUMA+ in Figure 13. We set the fast tier as 16/32 GB.

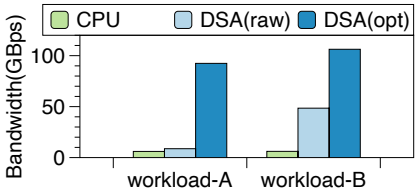


Figure 10: The performance of adaptable copying algorithm.

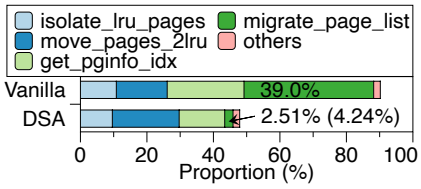


Figure 11: The perf breakdown in DSA-2LM.

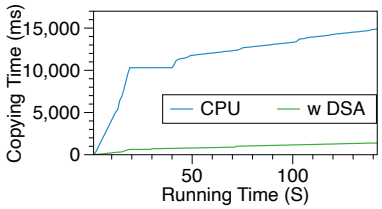


Figure 12: CDF of copying time.

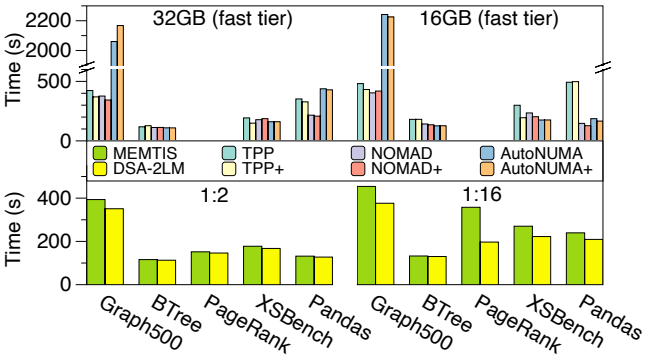


Figure 13: Performance with different applications.

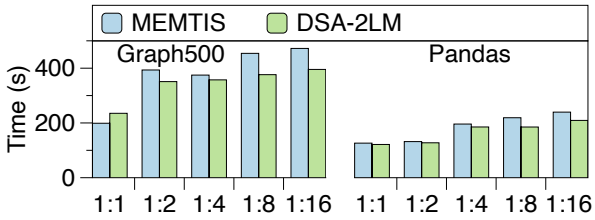


Figure 14: Performance with different ratio.

Note that PageRank's RSS is less than 16 GB, so it is excluded from this comparison. For TPP, Graph500 and XSBench show performance improvements of 14.5%/11.5% and 29.8%/53.9% respectively for the 32/16 GB configuration. For NOMAD, the baseline already outperforms TPP in the majority of cases across four workloads. When DSA is integrated with NOMAD, two-thirds of the cases show 4%∼16% performance improvements. This enhancement is attributed to the increased page migration efficiency, which effectively reduces the migration window and consequently decreases the probability of transactional migration failures. For AutoNUMA, since it employs a more conservative page placement strategy, DSA integration generally yields minimal performance changes. In Pandas, AutoNUMA+ achieves performance improvements of 2.0% and 12.1% for the 32 GB and 16 GB configurations, respectively.

The bottom part of Figure 13 compares DSA-2LM against MEMTIS. When the ratio of fast/slow tier is 1:2, DSA-2LM outperforms MEMTIS in all evaluated benchmarks by 2.5∼12.0% for five applications. When increasing the ratio to 1:16, the performance enhancement averages 28% across all benchmarks. In the best case, DSA-2LM achieves a 1.8× speedup compared to MEMTIS. This significant improvement occurs because the fast tier memory can only accommodate a small subset of hot pages, causing the hottest page set to change rapidly. DSA-based fast page migration enables better adaptation to these dynamic access pattern changes.

We measure Graph500/Pandas as represented applications for further case analysis (Figure 14). When the ratio is 1:1, the performance increase is insignificant. When the ratio is 1:8, the completion time can decrease by 20.68% in Graph500. The effects of DSA will be better with the ratio increasing.

5 Related Works

Compatibility with other tiered memory systems. Nomad [44] employs Transactional Page Migration (TPM), abandoning the page copy if the page is modified during migration. Since DSA can speed up page migration by 5∼10×, which significantly reduces the migration window and consequently decreases the probability of TPM failures. In the case of Colloid [42], the key insight is balancing access latencies across different memory tiers. Then it determines the direction of page migration based on global access latency. In this context, page migration is still necessary, and our work is orthogonal to Colloid. Additionally, recent works like NeoMem [52] have been able to offload the page placement strategy to hardware as well. It would be interesting to integrate DSA-2LM with NeoMem to achieve a fully CPU-free solution.

Intel accelerator use cases. Recent research has demonstrated the effectiveness of Intel accelerators in optimizing storage, networking, and virtualization. For instance, AnolisOS leverages "Memory Fill" capability to pre-zero memory pages, reducing VM startup latency in large-memory configurations by offloading page-zeroing from CPUs to DSA [6, 9]. The Intel IAA plugin for RocksDB provides accelerated compression/decompression in RocksDB [14]. For storage and networking, DSA accelerates CRC/DIF generation and memory comparison through specialized hardware pipelines while maintaining data integrity [13, 19]. DSA-2LM is the first work to utilize DSA in tiered memory.

6 Conclusion

This paper introduces DSA-2LM, an efficient tiered memory system using DSA. It exploits the hardware capabilities of DSA to design an aggregation algorithm for adaptable data paths. DSA-2LM leverages batch/multi-WQ approaches for concurrent migration. Results on the real CXL platform show that applications benefit significantly from DSA.

References

[1] Intel® dsa transparent offload library. https://github.com/intel/DTO. [Accessed 29-05-2025].

[2] Perf Wiki — perf.wiki.kernel.org. https://perf.wiki.kernel.org/index.php/Main_Page. [Accessed 24-06-2024].

[3] Transparent hugepage support. https://docs.kernel.org/admin-guide/mm/transhuge.html. [Accessed 24-06-2024].

[4] Mitosis workload btree. https://github.com/mitosis-project/mitosis-workload-btree, 2020.

[5] Using the memmap kernel option. https://docs.pmem.io/persistent-memory/getting-started-guide/creating-development-environments/linux-environments/linux-memmap, 2020.

[6] Anolis cloud kernel. https://gitee.com/anolis/cloud-kernel, 2022.

[7] idxd driver for intel data streaming accelerator. https://lwn.net/Articles/805291/, 2022.

[8] Intel® agilex™ i-series fpga development kit user guide. https://www.intel.com/content/www/us/en/docs/programmable/683288/current/overview.html, 2022.

[9] Introduce page prezero with dsa support. https://gitee.com/anolis/cloud-kernel/pulls/702, 2022. [Accessed 03-06-2025].

[10] Intel® data streaming accelerator architecture specification. https://www.intel.com/content/www/us/en/content-details/671116/intel-data-streaming-accelerator-architecture-specification.html, 2024.

[11] Intel® data streaming accelerator user guide. https://www.intel.com/content/www/us/en/content-details/759709/intel-data-streaming-accelerator-user-guide.html, 2024.

[12] Polars decision support (pds) benchmarks. https://pola.rs/posts/benchmarks/, 2024.

[13] Tuning guide for user space network stack acceleration with intel ® dsa. https://www.intel.com/content/www/us/en/developer/articles/guide/user-space-network-stack-acceleration-dsa.html, 2024.

[14] Intel® in-memory analytics accelerator plugin for rocksdb storage engine. https://github.com/intel/iaa-plugin-rocksdb, 2025.

[15] Neha Agarwal and Thomas F Wenisch. Thermostat: Application-transparent page management for two-tiered main memory. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 631–644, 2017.

[16] Andrea Arcangeli. Transparent hugepage support. In *KVM forum*, volume 9, 2010.

[17] Yossi Azar, Seny Kamara, Ishai Menache, Mariana Raykova, and Bruce Shepard. Co-location-resistant clouds. In *Proceedings of the 6th Edition of the ACM Workshop on Cloud Computing Security*, pages 9–20, 2014.

[18] Scott Beamer, Krste Asanović, and David Patterson. The gap benchmark suite, 2017.

[19] André Berthold, Constantin Fürst, Antonia Obersteiner, Lennart Schmidt, Dirk Habich, Wolfgang Lehner, and Horst Schirmeier. Demystifying intel data streaming accelerator for in-memory data processing. In *Proceedings of the 2nd Workshop on Disruptive Memory Systems*, pages 9–16, 2024.

[20] Alibaba Cloud. Thp-related performance optimization in alibaba cloud linux. https://www.alibabacloud.com/help/en/ecs/transparent-huge-page-thp-related-performance-optimization-in-alibaba-cloud-linux-2, 2024.

[21] Google Cloud. Cloud sql for postgresql now supports linux huge pages. https://cloud.google.com/blog/products/databases/cloud-sql-postgresql-now-supports-linux-huge-pages, 2024.

[22] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[23] Padmapriya Duraisamy, Wei Xu, Scott Hare, Ravi Rajwar, David Culler, Zhiyi Xu, Jianing Fan, Christopher Kennelly, Bill McCloskey, Danijela Mijailovic, et al. Towards an adaptable systems architecture for memory tiering at warehouse-scale. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pages 727–741, 2023.

[24] Intel. Intel® data streaming accelerator (intel® dsa). https://www.intel.com/content/www/us/en/products/docs/accelerator-engines/data-streaming-accelerator.html. [Accessed 24-06-2024].

[25] Reese Kuper, Ipoom Jeong, Yifan Yuan, Ren Wang, Narayan Ranganathan, Nikhil Rao, Jiayu Hu, Sanjay

Kumar, Philip Lantz, and Nam Sung Kim. A quantitative analysis and guidelines of data streaming accelerator in modern intel xeon scalable processors. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 37–54, 2024.

[26] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. What is twitter, a social network or a news media? In *Proceedings of the 19th International Conference on World Wide Web*, WWW '10, page 591–600, New York, NY, USA, 2010. Association for Computing Machinery.

[27] Woosuk Kwon, Gyeong-In Yu, Eunji Jeong, and Byung-Gon Chun. Nimble: Lightweight and parallel gpu task scheduling for deep learning. *Advances in Neural Information Processing Systems*, 33:8343–8354, 2020.

[28] Andres Lagar-Cavilla, Junwhan Ahn, Suleiman Souhlal, Neha Agarwal, Radoslaw Burny, Shakeel Butt, Jichuan Chang, Ashwin Chaugule, Nan Deng, Junaid Shahid, et al. Software-defined far memory in warehouse-scale computers. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 317–330, 2019.

[29] Taehyung Lee, Sumit Kumar Monga, Changwoo Min, and Young Ik Eom. Memtis: Efficient memory tiering with dynamic page classification and page size determination. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 17–34, 2023.

[30] Huaicheng Li, Daniel S Berger, Stanko Novakovic, Lisa Hsu, Dan Ernst, Pantea Zardoshti, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, et al. Pond: Cxl-based memory pooling systems for cloud platforms. In *ASPLOS 2023*, page 0, 2023.

[31] Zijun Li, Linsong Guo, Jiagan Cheng, Quan Chen, Bing-Sheng He, and Minyi Guo. The serverless computing survey: A technical primer for design architecture. *ACM Computing Surveys (CSUR)*, 54(10s):1–34, 2022.

[32] Teng Ma, Zheng Liu, Chengkun Wei, Jialiang Huang, Youwei Zhuo, Haoyu Li, Ning Zhang, Yijin Guan, Dimin Niu, Mingxing Zhang, et al. {HydraRPC}:{RPC} in the {CXL} era. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*, pages 387–395, 2024.

[33] Hasan Al Maruf, Hao Wang, Abhishek Dhanotia, Johannes Weiner, Niket Agarwal, Pallab Bhattacharya, Chris Petersen, Mosharaf Chowdhury, Shobhit Kanaujia, and Prakash Chauhan. Tpp: Transparent page placement for cxl-enabled tiered-memory. pages 742–755, 2023.

[34] Richard C Murphy, Kyle B Wheeler, Brian W Barrett, and James A Ang. Introducing the graph 500. *Cray Users Group (CUG)*, 19(45-74):22, 2010.

[35] Alan Nair, Sandeep Kumar, Aravinda Prasad, Ying Huang, Andy Rudoff, and Sreenivas Subramoney. Telescope: telemetry for gargantuan memory footprint applications. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*, pages 409–424, 2024.

[36] Stanko Novakovic, Yizhou Shan, Aasheesh Kolli, Michael Cui, Yiying Zhang, Haggai Eran, Boris Pismenny, Liran Liss, Michael Wei, Dan Tsafrir, et al. Storm: a fast transactional dataplane for remote data structures. In *Proceedings of the 12th ACM International Conference on Systems and Storage*, pages 97–108, 2019.

[37] Amanda Raybuck, Tim Stamler, Wei Zhang, Mattan Erez, and Simon Peter. Hemem: Scalable tiered memory management for big data applications and real nvm. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 392–407, 2021.

[38] Jie Ren, Dong Xu, Junhee Ryu, Kwangsik Shin, Daewoo Kim, and Dong Li. Mtm: Rethinking memory profiling and migration for multi-tiered large memory. In *Proceedings of the Nineteenth European Conference on Computer Systems*, pages 803–817, 2024.

[39] Muhammad Aditya Sasongko, Milind Chabbi, Paul HJ Kelly, and Didem Unat. Precise event sampling on amd versus intel: Quantitative and qualitative comparison. *IEEE Transactions on Parallel and Distributed Systems*, 34(5):1594–1608, 2023.

[40] Yan Sun, Yifan Yuan, Zeduo Yu, Reese Kuper, Chihun Song, Jinghan Huang, Houxiang Ji, Siddharth Agarwal, Jiaqi Lou, Ipoom Jeong, et al. Demystifying cxl memory with genuine cxl-ready systems and devices. pages 105–121, 2023.

[41] John R Tramm, Andrew R Siegel, Tanzima Islam, and Martin Schulz. Xsbench-the development and verification of a performance abstraction for monte carlo reactor analysis. *The Role of Reactor Physics toward a Sustainable Future (PHYSOR)*, 2014.

[42] Midhul Vuppalapati and Rachit Agarwal. Tiered memory management: Access latency is the key! In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, pages 79–94, 2024.

[43] Zixuan Wang, Suyash Mahar, Luyi Li, Jangseon Park, Jinpyo Kim, Theodore Michailidis, Yue Pan, Tajana Rosing, Dean Tullsen, Steven Swanson, Kyung Chang Ryoo, Sungjoo Park, and Jishen Zhao. The hitchhiker's

guide to programming and optimizing cxl-based heterogeneous systems, 2024.

[44] Lingfeng Xiang, Zhen Lin, Weishu Deng, Hui Lu, Jia Rao, Yifan Yuan, and Ren Wang. Nomad:{Non-Exclusive} memory tiering via transactional page migration. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 19–35, 2024.

[45] Dong Xu, Junhee Ryu, Kwangsik Shin, Pengfei Su, and Dong Li. {FlexMem}: Adaptive page profiling and migration for tiered memory. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*, pages 817–833, 2024.

[46] Zi Yan, Daniel Lustig, David Nellans, and Abhishek Bhattacharjee. Nimble page management for tiered memory systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 331–345, 2019.

[47] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. An empirical guide to the behavior and use of scalable persistent memory. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 169–182, 2020.

[48] Jifei Yi, Benchao Dong, Mingkai Dong, and Haibo Chen. On the precision of precise event based sampling. In *Proceedings of the 11th ACM SIGOPS Asia-Pacific Workshop on Systems*, pages 98–105, 2020.

[49] Yifan Yuan, Ren Wang, Narayan Ranganathan, Nikhil Rao, Sanjay Kumar, Philip Lantz, Vivekananthan Sanjeepan, Jorge Cabrera, Atul Kwatra, Rajesh Sankaran, et al. Intel accelerators ecosystem: An soc-oriented perspective: Industry product. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, pages 848–862. IEEE, 2024.

[50] Mingxing Zhang, Teng Ma, Jinqi Hua, Zheng Liu, Kang Chen, Ning Ding, Fan Du, Jinlei Jiang, Tao Ma, and Yongwei Wu. Partial failure resilient memory management system for (cxl-based) distributed shared memory. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 658–674, 2023.

[51] Yuhong Zhong, Daniel S Berger, Carl Waldspurger, Ishwar Agarwal, Rajat Agarwal, Frank Hady, Karthik Kumar, Mark D Hill, Mosharaf Chowdhury, and Asaf Cidon. Managing memory tiers with cxl in virtualized environments. In *Symposium on Operating Systems Design and Implementation*, 2024.

[52] Zhe Zhou, Yiqi Chen, Tao Zhang, Yang Wang, Ran Shu, Shuotao Xu, Peng Cheng, Lei Qu, Yongqiang Xiong, Jie Zhang, and Guangyu Sun. NeoMem: Hardware/Software Co-Design for CXL-Native Memory Tiering . In *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1518–1531, Los Alamitos, CA, USA, November 2024. IEEE Computer Society.

Turbocharge ANNS on Real Processing-in-Memory by Enabling Fine-Grained Per-PIM-Core Scheduling

Puqing Wu[1†], Minhui Xie[1†], Enrui Zhao[1], Dafang Zhang[1], Jing Wang[1],
Xiao Liang[2], Kai Ren[2], Yunpeng Chai[1*]
[1]*Renmin University of China*
[2]*Kuaishou*

Abstract

Approximate Nearest Neighbor Search (ANNS) plays a key role in database and AI infrastructure. It exhibits extremely high memory intensity with a ∼1:1 compute-to-memory access ratio. Commodity Processing-in-Memory (PIM) hardware such as UPMEM is promising for overcoming the memory wall in ANNS. However, its reuse of the system DDR bus prevents the CPU and PIM cores from accessing memory simultaneously. This necessitates batch scheduling in existing systems, which, in turn, leads to severe underutilization in two scenarios: 1) inter-batch, where PIM remains idle while the CPU is copying data, and 2) intra-batch, caused by uneven load distribution of PIM cores in a batch.

This paper proposes an efficient PIM-capable ANNS system named PIMANN. We observe that each PIM core has an additional, undocumented, and little-known control interface (originally used for control commands like launching PIM kernels), which could be retrofitted for fine-grained arbitration of DDR bus access. Thus, PIMANN can break the traditional batching scheduling paradigm and adopt a fine-grained, per-PIM-core scheduling paradigm. With this key idea, PIMANN introduces 1) persistent PIM kernel technique to eliminate the idle state between two batches, and 2) per-PU query dispatching technique that dispatches queries based on the real-time load of PIM cores. Experiments show that PIMANN can boost throughput by 2.4-10.4× compared to existing ANNS systems on CPU or GPU. The implementation of PIMANN is available at https://github.com/cds-ruc/PIM-ANNS.

1 Introduction

Nearest Neighbor Search (NNS) finds top-k vectors closest to a query vector in a dataset. Approximate Nearest Neighbor Search (ANNS) balances speed and accuracy, making it suitable for massive high-dimensional data where exact NNS is costly. ANNS has wide applications in various scenarios, including information retrieval tasks such as search [59] and recommendation [66] (e.g., image-based search [7] and content retrieval [19]), or assist large language models through retrieval-augmented generation (RAG) [48].

ANNS is characterized by extremely high memory intensity [34]. The core of ANNS lies in maintaining in-memory data structures to quickly identify potential neighbors. Currently, mainstream ANNS algorithms rely on traversing data points within their respective data structure (i.e., cluster-based [36] or graph-based [68]), resulting in an extremely low compute-to-memory access ratio, close to 1:1.

Constrained by the von Neumann memory wall [53], current ANNS systems based on CPUs [35] or GPUs [71] are increasingly unable to meet the growing demand in both volume and performance [40]. While CPUs can provide TB-scale memory capacity to store large volumes of vectors, their limited DRAM bandwidth (approximately 200 GB/s) limits vector retrieval performance. Existing work [14] shows that modern CPUs, with SIMD instructions equipped, can typically process 5 billion vector operations per second (with a compute bandwidth of around 500 GB/s), far exceeding DRAM bandwidth. On the other hand, while GPUs offer substantial computing capability and high-bandwidth memory, their limited memory capacity (even a recent high-end GPU like H100 has only 100 GB) prevents them from handling storage requirements for billions of vectors with thousands of dimensions.

Processing-in-Memory (PIM) is a classic approach to overcoming the memory wall by performing computations directly within memory, eliminating the need for extensive data transfers between memory and processors. With the release of UPMEM [65], the world's first commercially available real PIM hardware in 2022, this technology is finally no longer limited to software simulations [5, 6, 10, 11, 15, 18, 20, 23, 26, 29, 31] or FPGA verification [4, 5, 32, 37] and is ready for public real use [42, 47, 57]. UPMEM modules can be plugged into general-purpose servers via DIMM interfaces, functioning like DRAM. Each module integrates two levels of memory (WRAM and MRAM) and incorporates over one hundred

† These authors contributed equally.
∗ Corresponding author (ypchai@ruc.edu.cn).

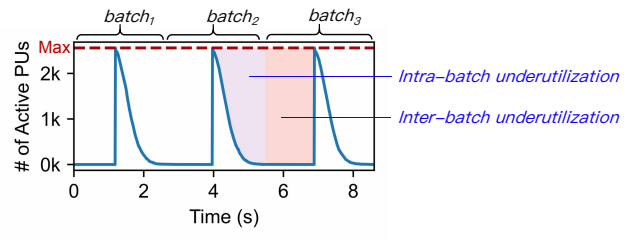


Figure 1: The active PU count over time in a PIM-capble ANNS strawman system. *The red dashed line shows the maximum PU count (2560). In an ideal case, the blue line should closely follow the red dashed line as much as possible.*

RISC-based processing units (PUs[1]), with each PU capable of supporting up to 16 parallel threads. Equipped with 20 UPMEM modules, a single server can provide a total of 2,560 PUs (supporting over 40,000 threads) and deliver an aggregate memory bandwidth of 2 TB/s.

However, due to the architectural differences between UPMEM and CPU/GPU, simply shoehorning existing ANNS systems into UPMEM can only achieve 18.2% of the hardware's theoretical throughput cap (see §2.3), and the number of active PUs is zero for more than 65% of the time (Figure 1). We analyze that this under-utilization stems from the deficiency of **batching paradigm** (§2.3), including the following two key factors:

- *Inter-batch under-utilization, caused by batches' gang scheduling.* Due to UPMEM's reuse of the system DDR bus, it currently does not natively support concurrent memory access by both the CPU and PUs; see §2.2. As a result, existing applications [8, 9, 16, 25, 27, 30, 38, 43, 51, 54] typically rely on batching paradigm, where the CPU first copies all necessary data to UPMEM, and then gang-schedules all PUs to run. This leads to a gap between two batches where PUs are unable to perform computations.
- *Intra-batch under-utilization, caused by PU load unbalance within a batch.* Although UPMEM enjoys a strong aggregated multi-core performance, its single-core capability is rather weak. Unlike CPUs, where multiple cores can access each other through the shared memory architecture, UPMEM hardware restricts PUs from mutual access. Thus, uneven load distribution among PUs may cause some straggler PUs to become the performance bottleneck for the entire batch.

To this end, we propose PIMANN, an ANNS system designed to fully harness UPMEM's potential. Its core idea is breaking the traditional batching scheduling paradigm and adopting a fine-grained, per-PU scheduling paradigm. We observe that the necessity of existing batch gang-scheduling arises from the hypothesis that CPU and PUs can only communicate at the boundary of batches due to their mutually exclusive access to the shared DDR bus. Under this paradigm, once a PIM kernel (i.e., a batch of tasks) is launched, it must be assumed that the PU always retains control of the shared bus until the kernel completes. However, this constraint could be relaxed, with *our key observation that each PU has an additional, undocumented, and little-known control interface (originally used for launching and synchronizing PU programs), which could be retrofitted for fine-grained arbitration of DDR bus access*.

With this key idea, we propose *persistent PIM kernel* (§4.2) to address inter-batch under-utilization. persistent PIM kernel allows continuous sending of individual ANN requests to different PUs for computation in runtime. Unlike the original batch gang-scheduling approach, where there is a pause between two batches, our approach ensures that UPMEM remains uninterrupted throughout the entire process. Specifically, by modifying the UPMEM driver, persistent PIM kernel carefully coordinates the access to the corresponding memory (i.e., MRAM) by either the CPU or PU through each PU's control interface. Due to the limited shared bus bandwidth ($\sim$0.41 GB/s), the CPU or PUs may experience prolonged data stalls when attempting to gain bus control while the other is holding it. To minimize this data stall, PIMANN employs coroutine techniques on the CPU, which hides the bus acquisition time and parallelizes data transmission and computation.

To cope with intra-batch under-utilization, PIMANN introduces *per-PU query dispatching* (§4.3) to dispatch queries to individual PUs based on their real-time load to achieve well load balance. Since PUs follow a share-nothing architecture, PIMANN employs a selective replication data placement scheme, which replicates hot clusters across multiple PUs. With the help of fine-grained per-PU scheduling, PIMANN enables 1) live adjusting data placement without downtime based on the updated hotness information, and 2) dynamically dispatching queries to different replicas based on the real-time load of PUs.

Our evaluation on off-the-shelf UPMEM shows that, PIMANN can improve the throughput by up to 10.4$\times$ compared with Faiss-CPU [22] and by up to 2.4$\times$ compared with a PIM-capable ANNS system based on the batching paradigm. Compared to a pure-GPU-storage system, Faiss-GPU [22], PIMANN can also improve the throughput by up to 3.7$\times$.

2 Background and Motivation

2.1 Approximate Nearest Neighbor Search (ANNS)

By 2030, humanity will enter the YB-scale data era, with more than 90% being unstructured [33]. ANNS is the primary method for retrieving useful information from massive unstructured data. ANNS has wide applications in various

[1]This component is referred to as the data processing unit (DPU) in UPMEM. Here we call it PU to avoid ambiguity with smart NICs.

scenarios. Initially, they were used in traditional information retrieval tasks such as search [59] and recommendation [66] (e.g., image-based search [7] and content retrieval [19]). In recent years, as large language models (LLMs) have emerged as a new frontier of artificial intelligence, ANNS has become a core infrastructure to carry knowledge for LLMs. Through retrieval-augmented generation (RAG) [48], they assist LLMs in retrieving credible, accurate, and timely knowledge, helping to mitigate hallucinations [64].

ANNS exhibits an extremely high memory intensity. First, ANNS needs to repeatedly read large volumes of vectors from memory, causing massive memory access. For example, Alibaba handles approximate searches on billions of data points [69], while Microsoft performs approximate queries on datasets with hundreds of billions of entries [1]. Second, ANNS have an extremely low compute-to-memory access ratio. Typical ANNS algorithms rely on maintaining in-memory data structures to quickly identify potential neighbors. Currently, ANNS algorithms can be broadly classified into two categories, cluster-based [36] and graph-based [68]. Both rely on traversing data points within their respective data structure (i.e., clusters or graphs), resulting in a compute-to-memory access ratio close to 1:1.

In this paper, we explore ANNS systems on UPMEM, with a specific focus on cluster-based algorithms, which are well-suited to UPMEM's massive parallel threading capabilities. Specifically, we focus on the Inverted File with Product Quantization (IVFPQ) algorithm, as it incorporates the Product Quantization (PQ) technique on standard cluster algorithms to further reduce computational overhead. Another category of ANNS algorithms (graph-based methods) are not suitable for UPMEM, because they introduce significant communication overhead when implemented on PIM architectures. This limitation is caused by the constrained communication bandwidth on commodity UPMEM systems—only 0.41 GB/s (from our test results) for both inter-PU and CPU-PIM data transfers. In fact, to the best of our knowledge, there are no existing implementations of graph-based ANNS on PIM devices.

Figure 2 shows an example of the IVFPQ algorithm, which includes two phases: the offline phase and the online phase.

Offline Phase: The offline phase is responsible for preparing the dataset for efficient search by building a cluster-based index. This phase includes two steps: ① Inverted File (IVF) and ② Product Quantization (PQ).

① The IVF step partitions the whole dataset into C clusters using methods such as K-means. Once the clusters are formed, ② the PQ step calculates the residuals, which are defined as the differences between each data point x and its corresponding cluster centroid c. It then compresses these residuals by dividing the residual vectors into M subvectors and then encoding them using a codebook. Each encoded point serves as an identifier that has a one-to-one mapping with a specific codeword in the codebook. In practical implementations, the identifiers are often encoded as 8-bit unsigned

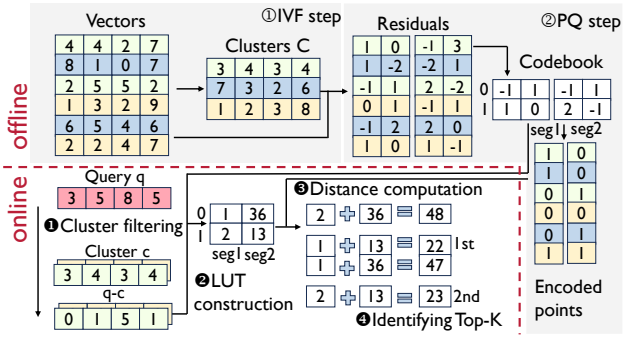


Figure 2: IVFPQ example.

integers (uint8), enabling a compression rate of $\frac{4D}{M}$.

Online Phase: Given a query vector q, the online phase includes 4 steps to find k vectors closest to q. ❶ Cluster filtering, which finds *nprobes* clusters that are closest to q by computing the distances between their centroids and q. ❷ Look-up table (LUT) construction. The LUT serves to store pre-computed data, significantly optimizing the search process by converting computing to memory lookups. Specifically, each element LUT$[j][i]$ represents the distance between the j-th vector in the codebook B_i, where i denotes the i-th segment of residual vectors. To derive the distance from q to a given data point x, one can refer to and sum up the values in LUT as follows:

$$L_2(q,x) = L_2(q-c,r) = \sum_{i=0}^{M-1} \text{LUT}[e_i][i],$$

where r and e. denote the residual and encoded vectors of x, respectively.

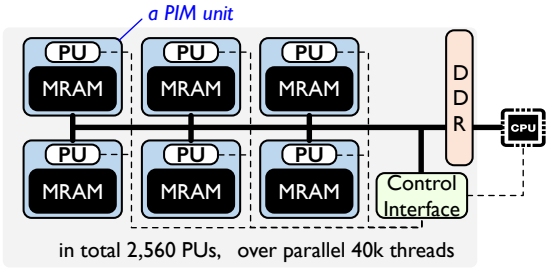
❸ Distance computation. Leveraging LUTs, the distances from q to all data points within the filtered clusters are calculated. ❹ Identifying Top-K, which sorts all computed distances and selects the k vectors with the shortest distances as the approximate nearest neighbors.

2.2 Commodity PIM hardware architecture

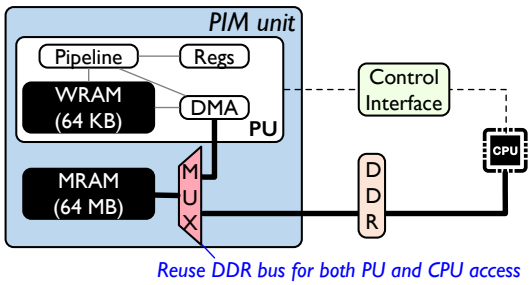
Processing-in-Memory (PIM) is a classic way to deal with the famous "memory wall" problem [6]. For a long time, various PIM architectures have been proposed [17]. However, prior to the release of UPMEM in 2022, these PIM architectures were confined to simulations or FPGA-based verification. UPMEM marked a significant milestone as the world's first commercially available real-world PIM hardware, bridging the gap between theoretical concepts and practical applications. Consequently, our work mainly focuses on leveraging UPMEM to build ANNS system.

In this section, we mainly introduce the following three hardware features of UPMEM.

Multi-core architecture. Figure 3a shows the architecture of the UPMEM-capable system and inner-UPMEM-DIMM



(a) Architecture of UPMEM-capable system.



(b) Inner-PIM-unit details.

Figure 3: Architecture of UPMEM-capable system and inner-PIM-unit details. *The bold line represents DDR bus, and the dashed line represents control interface.*

details. UPMEM are connected to the CPU through DIMM slots. An UPMEM DIMM is made up of 64 PIM units. As shown in Figure 3b, each unit has two main components: a memory bank (MRAM) and a general-purpose processing unit (PU) integrated with the bank. Each PU is a 32-bit in-order RISC-V core featuring a 14-stage pipeline and supporting up to 16 hardware threads. With 20 UPMEM DIMMs, a single server can harness a total of 2,560 PUs, accommodating over 40,000 threads and achieving an aggregate memory bandwidth of 2.5 TB/s. Due to limitations in hardware fabrication, PUs in the current generation of PIM hardware have the following three issues: 1) PU only offers integer-native ALUs; float-related operations are implemented via software. This causes UPMEM to have a much slower FLOPS compared to integer operations. 2) All PUs adhere to a share-nothing architecture, meaning that each PU can only access data stored in its local memory bank. Currently, any communication between PU has to go through the host CPU.

Two-level memory hierarchy. Each PIM unit contains two levels of memory internally, including 1) Working RAM (WRAM), and 2) Main RAM (MRAM). WRAM acts as the faster cache, made of SRAM, with a size of 64KB. MRAM has a substantially larger storage capacity (64MB), but it has slower access speeds compared to WRAM. PU cannot directly access MRAM; it must first bring data from MRAM to WRAM via the on-chip DMA engine.

Bus sharing on MRAM. MRAM receives access from both the CPU and the PU. The CPU is used for sending/collecting task data, while the PU is for PIM computing. Nevertheless, the DIMM-PIM architecture (such as UPMEM), which plugs PIM devices into DIMM slots, inherently lacks the support for simultaneous memory access by the host CPU and the PUs. Specifically, as shown in Figure 3b, each PIM unit reuses the DDR bus for both the PU and host CPU access. Due to the constraints of existing DDR bus protocols, which require deterministic latency behavior [61], simultaneous memory access by the CPU and PU is not feasible. This is because shared access alters the memory access latency, violating the strict timing guarantees of DDR protocols. To address this limitation, UPMEM hardware employs a multiplexer (MUX) for manual access arbitration. Specifically, MUX is a two-state register, namely CPU-side or PU-side, allowing exclusive access to the MRAM by the CPU or the PU, respectively. The state of MUX is controlled by the CPU through the *control interface*. With this design, it simplifies the integration of PIM into traditional systems while adhering to the requirements of DDR bus protocols.

2.3 Motivation

Batching paradigm of current PIM programming model. The PIM programming model adheres to an accelerator model, similar to that of GPUs. Programmers prepare an execution program (referred to as a *kernel*) along with its data in advance and then launch it on the PIM device. The CPU communicates with PIM via the *control interface*, by sending commands such as starting/synchronizing kernels or querying the status of the running kernel (Figure 3b).

Currently, PIM-capable programs [12, 13, 39, 46, 49, 51] typically follow the batching paradigm. This paradigm stems from the limitation that the CPU can only interact with PIM's MRAM at kernel boundaries—either before launching a kernel or after it completes. This restriction arises with the hypothesis that, once PUs are activated, PUs will retain the bus control until the kernel terminates. Thus, the CPU can not read/write PIM's MRAM when PUs are running.

This batching paradigm includes three steps; see Figure 4. First, the CPU gets a buffer ready, filling it with a set of tasks (i.e., a batch) meant for each PIM unit, and sends the buffer to all PIM's MRAM. Next, the CPU launches the PIM kernel, prompting the PIM units to start working on their assigned tasks and generate computing results. Lastly, the CPU collects all the running results and moves them into the host memory.

Impact of batching paradigm on ANNS. We investigate the deficiency of existing batching paradigm on ANNS. We implement a strawman ANNS system based on Faiss [22], by storing clusters on MRAM and offloading the distance computation step (❸ in the Online Phase) to PIM's PUs. Figure 5 shows their performance on the SPACE-1B dataset [3]. We can observe that:

- *Achieving only 18.2% of the theoretical performance.* Figure 5a shows the computing performance of Faiss-CPU

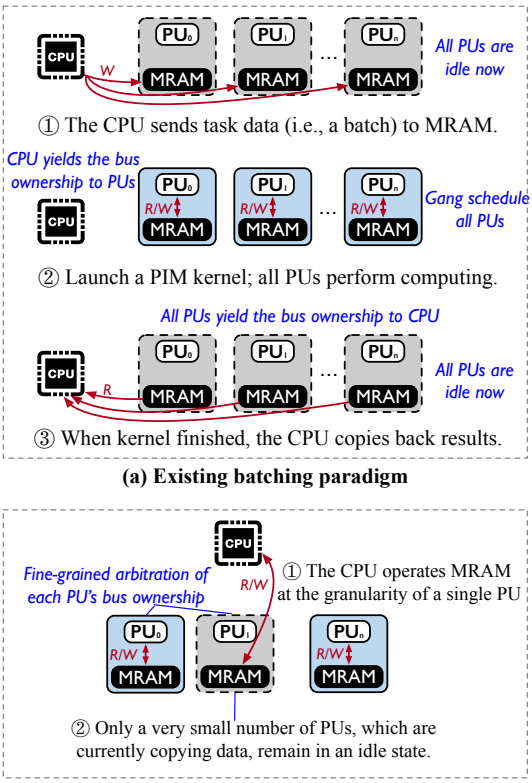


Figure 4: Comparision of existing batching paradigm and PIMANN's per-PU scheduling paradigm.

and PIM-capable strawman system. We observe that: 1) for the strawmen system, the throughput of PIM, although with tens of thousands of threads, is comparable with that of a standard CPU. 2) PIM's performance is far from being fully utilized. If fully leveraged, PIM has the potential to achieve more than 5.5× the throughput. Further analysis reveals that there are two types of under-utilization as follows.

- *Inter-batch under-utilization.* Figure 1 shows the number of active PUs across time. Between two batches, there is a period of over 1.21s during which the number of active PUs is zero. This period accounts for 65% of a single batch's duration. This is because, during this time, the CPU is copying the results of the previous batch from MRAM while transferring the input for the next batch to MRAM. Since the CPU and PUs cannot access the shared bus simultaneously, PUs cannot be activated.
- *Intra-batch under-utilization.* In batch scheduling, a batch of queries is distributed among the PUs. Figure 5b shows the number of tasks assigned to each PU. As shown in this figure, imbalanced workloads across PUs may cause some PUs to idle while others are overloaded, delaying the completion of the entire batch.

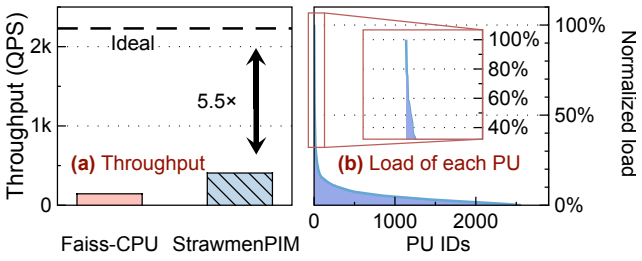


Figure 5: Motivation. *(a) The throughput of a standard CPU ANNS system (Faiss-CPU) and a strawmen PIM-capable system (StrawmenPIM). The dashed line denotes the ideal performance of UPMEM. (b) The load of each PU. The load values are normalized by the heaviest PU.*

3 Key Idea & Challenges

Key Idea: Per-PU scheduling with fine-grained arbitration. Traditional PIM systems use batched gang scheduling where CPUs and PUs operate MRAM alternately, leading to the inefficiencies of inter/intra-batch underutilization (§2.3). These issues are rooted in the assumption that PUs retain control of the shared bus for the entire duration of the PIM kernel's execution, from launch to completion. Differently, this assumption is broken with our key observation that: each PU in UPMEM has an additional control interface (originally used for control commands), which could be retrofitted for sending custom control messages to fine-grainedly coordinate the access of PU and CPU. Building on this insight, we propose a per-PU scheduling paradigm (Figure 4b) by manipulating the ownership of each PU unit's MRAM, determining whether it is currently being read/written by PU/CPU.

Per-PU scheduling offers two advantages. First, it eliminates the idle state between two batches (Figure 1), effectively handling inter-batch underutilization. Each PU can be in one of two states: either performing computation or waiting for the CPU to copy data. Second, without gang scheduling, it enables dynamically dispatching each query to different PUs based on their real-time loads, thus achieving good load balance and avoiding intra-batch underutilization.

Challenges. However, achieving per-PU scheduling is not easy. First, the native driver does not allow reading from or copying data to PIM when PIM is running. Second, due to the limited read/write bandwidth between the CPU and the PU's MRAM, switching a PU's state may cause the CPU to experience a long stall. Third, although per-PU scheduling can dynamically send queries based on the current load of each PU, PUs do not share the same memory space. Therefore, an efficient data placement scheme needs to be designed to pre-distribute clusters across the PUs.

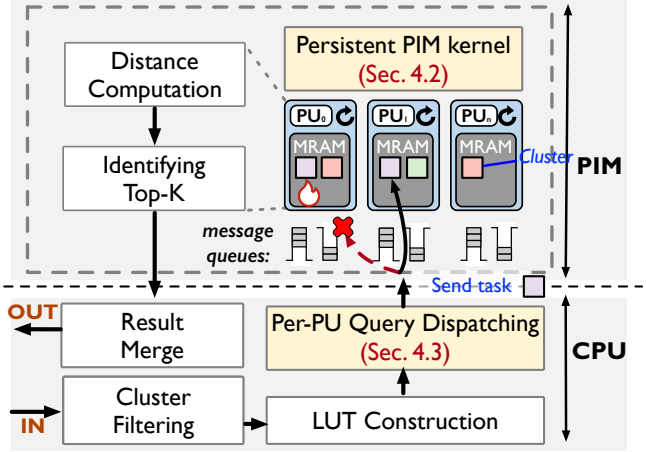


Figure 6: Overview of PIMANN. *This figure shows the workflow of an ANNS query in PIMANN. It supposes PU_0 is overloaded. Our per-PU query dispatching will dispatch this query to the replica on PU_1.*

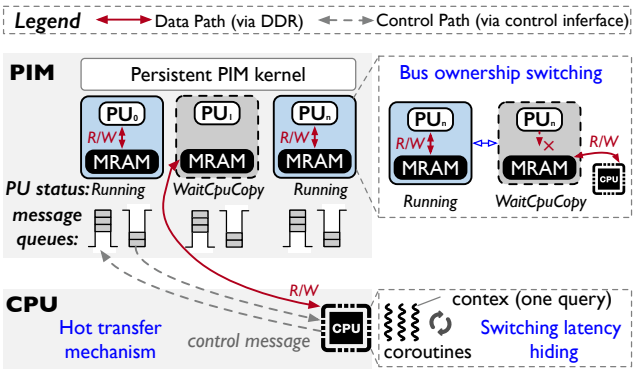


Figure 7: Design of persistent PIM kernel. *Each PU contiguously pulls tasks from the message queue and performs computation. The input/output data of tasks are transferred via the hot transfer mechanism, involving multiple bus ownership switches.*

4 PIMANN Design

4.1 Overview

PIMANN architecture. Figure 6 shows the architecture of PIMANN. PIMANN offloads the most parallelizable distance computation step (see §2.1) of the entire IVFPQ process to UPMEM, while the other steps are still executed by the CPU. Each PU stores multiple IVF clusters and performs the corresponding distance computations.

To mitigate inter-batch underutilization, PIMANN introduces the *persistent PIM kernel* (§4.2), which launches a persistent kernel enabling continuous query processing without idle states between batches. Each PU can independently receive queries from the CPU continuously.

However, due to bus sharing, the native UPMEM driver does not support copying data to/from PIM when PIM is running. Based on our observation that the additional control interface can be retrofitted for fine-grained bus arbitration, we implement a *hot transfer mechanism* to transfer data between the CPU and PU when PIM is running (§4.2.1).

With bus arbitration, each PU can dynamically switch between two states: *Running* (performing computation) or *WaitCPUCopy* (waiting for the CPU to copy data). These two states correspond to the ownership of the shared bus being held by either the PU or the CPU. We discuss how to switch two states safely and efficiently in §4.2.2.

To optimize energy efficiency during idle periods (i.e., when no queries are being processed), we also support transitioning from the persistent kernel to the normal kernel.

To mitigate intra-batch underutilization, we propose *per-PU query dispatching* (§4.3). With the help of bus arbitration, we can dispatch queries to individual PUs based on their real-time load. However, as PUs follow a share-nothing architecture, different PUs cannot access each other's MRAM. Thus, we must tailor an efficient data placement scheme, pre-allocating each PU's clusters in advance. PIMANN leverages a selective replication for data placement (§4.3.1), replicating high-popularity clusters among PUs. It also detects hotness shifts and dynamically adjusts data placement without shutting down the PIM (§4.3.2).

4.2 Persistent PIM kernel

Different from the existing batching paradigm which launches a separate PIM kernel for each batch of queries and lets the CPU copy data between two kernels, PIMANN only launches one persistent PIM kernel for all ANNS queries during system initialization. In this kernel, since we allow dynamic arbitration of per-PU bus during runtime, each PU can independently perform computation or data copying. Thus, the persistent PIM kernel design eliminates the idle copying time introduced by batch gang scheduling.

The main workflow of each PU continuously loops through the following steps:

①Dequeue an ANNS request from the message queue.
②Wait for CPU to copy the task data (e.g., LUT) to MRAM.
③Switch to the *Running* state and perform computing.
④Switch to the *WaitCPUCopy* state.
⑤Wait for the CPU to copy the computing results.

Step ①②⑤ require data transfer from the CPU to the PU while PUs are in the running state; step ③④ require switching the PU's state (i.e., switching ownership of the shared bus). Therefore, we have designed the hot transfer mechanism (§4.2.1) and bus ownership switching scheme (§4.2.2), respectively.

4.2.1 Hot transfer between CPU and PIM

Next, we discuss how CPU and PUs transfer data to each other when PUs are running. This hot transfer mechanism includes two parts, with control and data paths separated: 1) The control path uses a custom *message queue*, implemented through the PU's control interface link and isolated from the DDR bus. 2) The data path allows the CPU to directly read/write MRAM via the DDR bus (but requires holding the bus ownership).

Control path: Message queue implemented on the control interface. To achieve fine-grained per-PU scheduling, the most fundamental function lies in the ability of CPU-PU communication when PIM is running. By carefully reading through the UPMEM driver, we notice that the CPU's read and write access to the WRAM is routed through an additional, undocumented control interface link (§2.2), completely bypassing the DDR bus. This hardware characteristic allows us to implement a message queue directly on WRAM for communication between the PU and the CPU.

This message queue has two limitations, which should be considered when designing our system. First, the message queue has a limited capacity (in the order of a few tens of bytes), In comparison, a common data structure in an ANNS query, LUT, has a typical size of 32 KB. To this end, we only store necessary metadata (e.g., query ID) or small control messages (e.g., ownership switching) in this queue. Other large messages (e.g., LUT) are directly written to MRAM. Second, as PIM does not have the ability to interrupt the CPU, PIMANN maintains dedicated CPU threads to poll all message queues of PUs to get notifications from PIM.

Data path: Exposing MRAM to CPU. The native UPMEM driver does not support exposing MRAM to the CPU for direct read and write operations during PIM execution. We perform a thorough analysis of the driver and make the following two modifications: 1) exposing MRAM to CPU when PIM is running (enabling the CPU to directly read/write MRAM), and 2) constructing a mapping table that maps all variable symbols to their corresponding MRAM addresses (finding where should the CPU read/write).

For 1), all MRAM are viewed as multiple device files in the operation system, and mapped in the virtual memory space. We can obtain the base address of mmap files by implanting a recording function in the PIM startup path. Considering UPMEM adopts memory-level parallelism [44], all writes to MRAM should perform a transpose operation on the data to ensure that the data can be correctly transmitted to the same chip [44].

For 2), the main challenge is the lack of mapping from variable symbols to offsets in the mmap files. Our solution is as follows: before starting the PIM, we first use the standard MRAM copy interface (`dpu_copy_to`) to access the corresponding variables once, recording the corresponding addresses and caching them in the user program.

It should also be noted that the bandwidth of directly writing MRAM by CPU in the running state seems to be relatively low (0.41 GB/s). The reason is that: different from batching paradigm which could easily exploit the bank-level parallelism (8 PUs), our fine-grained per-PU scheduling means that data can be copied to only one PU's MRAM at a time. Although per-8PU scheduling could achieve higher bandwidth, we abandon this approach for two main reasons. First, per-8PU scheduling incurs computation redundancy: if a cluster is split into 8 slices, each slice needs to maintain its own top-k queue, resulting in 8x computation redundancy. Second, per-PU scheduling already has a negligible data transfer time: Although per-PU may suffer from a relatively lower bandwidth, in our implementation, the time taken to copy data is negligible (please refer to the purple bar in Figure 15).

4.2.2 Per-PU bus ownership switching

In this subsection, we present how we safely and efficiently switch the state (i.e., bus ownership) of each PU in PIMANN.

Switching ownership safely. As stated in §2.2, the bus ownership is controlled by the MUX registers; we expose the mapping of each PU units' MUX registers to userspace. Each MUX has two states: CPU-side and PU-side. If there is an inconsistency between the memory access and the MUX state, it may cause abnormal access by either the CPU or PU.

Thus, we next show how PIMANN ensure this consistency. We discuss two cases of PU state switching respectively: 1) *WaitCPUCopy→Running* (i.e., MRAM access ownership switching from CPU to PU), and 2) *Running→WaitCPUCopy*, reversely. The first case corresponds to the CPU copying the LUT table to the PU, and waiting for the PU to start the distance calculation. The second case occurs after the computation finishes, where the PU waits for the CPU to perform result aggregation.

For *WaitCPUCopy→Running*, in PIMANN, the CPU sends an ownership transfer message to the queue after completing the copy operation. Thus, the PU only needs to poll the message queue to receive the message, after which it can safely access the MRAM and perform computation.

For *Running→WaitCPUCopy*, since the PU itself cannot switch its MUX, it sends a message to the CPU via the queue. Upon receiving this message, the CPU switches the MUX and sends an ack message back to the PU. At this point, the PU can enter the WaitCPUCopy state safely.

To reduce the software stack overhead, PIMANN caches the MUX state for each PU in userspace. The interface for checking the MUX state directly reads from the cache. When switching a PU bus ownership, both the actual MUX state and its cached copy are updated simultaneously. Since only the CPU can modify the MUX state, cache inconsistency is not a concern.

Ownership switch in pairs. In our implementation, we discover a strange phenomenon: when the CPU directly accesses the MRAM of a certain PU, it sometimes causes the

tasks on the adjacent PU to malfunction. This lead us to a reasonable speculation: two adjacent PUs may share a physical MUX register. It may be due to the special hardware circuit design of UPMEM that there is only one control interface for the MUXes of two adjacent PUs. We later confirm this speculation through experiments. As a result, we determine that the minimum switching granularity of PIMANN is two adjacent PUs. Thus, even if the current query only involves one PU, it is necessary to simultaneously switch the ownership of two adjacent PUs, resulting in a resource waste of the other adjacent PU.

To address this limitation and minimize resource waste, we propose the optimization of pairwise cluster slicing to ensure that adjacent PUs are always invoked simultaneously. Specifically, instead of assigning a single IVF cluster to a PU, each cluster is sliced into two smaller *cluster slices* and adjacent PUs are allocated segments from the same cluster. This ensures that when one PU in a pair is active, its neighboring PU is also active, thus maximizing hardware resource efficiency. We will refine this slicing strategy in the section of load balancing design (§4.3).

Hiding the switching latency. The per-PU bus ownership switching may experience significant latency if the required MUX ownership is not immediately available. The reasons are twofold. First, PU's computing capability is weak. The clock frequency of PU is only O(100) MHz, so the task execution time of a single PU may be relatively long. If the PU is still executing a task, the switching time also includes the wait for the ongoing computation to complete, which can be substantial, reaching up to several milliseconds.

Second, the CPU-PIM communication latency is high. Even if the PU already releases the ownership, the CPU side cannot obtain it immediately. Instead, it has to poll the message queue on WRAM through a bandwidth-limited interface [2]; it takes 0.9 ms to poll the message queue of one rank (64 PUs). Thus, naively blocking the CPU thread is inefficient.

To mitigate this, we employ a coroutine-based optimization that allows the CPU to switch execution contexts to other tasks while waiting for ownership switching, thus hiding the switching latency. Specifically, our coroutine scheduling prioritizes tasks based on the two following principles. 1) Immediate ownership availability: coroutines associated with PUs whose bus ownership is already set to CPU side are prioritized for execution. 2) Predictive scheduling: for PUs whose MUX state is on the PU side, our scheduler estimates when the MUX state will switch based on the fixed execution time of PU tasks. Coroutines likely to have their ownership status become available soon are given higher priority. This estimation leverages the deterministic nature [28] of UPMEM's in-order RISC cores, which have no caching or speculative execution.

4.3 Per-PU query dispatching

The design of the persistent PIM kernel enables us to dispatch queries to individual PUs based on their real-time load, thereby addressing intra-batch underutilization. However, since each PU follows a share-nothing architecture, different PUs cannot access each other's MRAM. As a result, it is crucial to design an appropriate data (i.e., cluster) placement strategy. A naive approach, such as random assignment of clusters to PUs, often leads to significant load imbalances due to variations in cluster popularity.

To this end, PIMANN employs a *selective replication data placement* scheme. This scheme selectively replicates hot clusters across multiple PUs. In this section, we discuss: 1) how to choose clusters to replicate in §4.3.1, 2) how to live adjust the data placement when hotness shifts in §4.3.2, and 3) how to dispatch queries between replicas to achieve well load balance in §4.3.3.

4.3.1 Selective replication data placement

The selective replication data placement includes the following three steps.

Replicating high-popularity clusters. Clusters are replicated based on their popularity, which is quantified as the product of cluster size (s_i) and access frequency (f_i), because popularity is determined by the query time of a cluster and the query time is proportional to its size and its access frequency. The number of replicas for a cluster is determined by dividing its popularity ($p_i = s_i * f_i$) by the average PU load. Formally, for a high-popularity cluster i, its number of replicas $replica_count_i = p_i / p_{avg}$. where $p_{avg} = \frac{\sum_i p_i}{n}$, n denotes the cluster count.

Slicing clusters for less memory fragmentation. As the size of MRAM is not necessarily an exact multiple of the cluster size, the previous strategy may lead to significant memory fragmentation. To address this problem, we divide clusters into smaller slices of uniform size. Specifically, MRAM is partitioned into fixed-length slots, each capable of holding one slice. This approach simplifies data allocation and minimizes memory fragmentation, especially given the variety in cluster sizes and dynamic insertion or deletion of clusters in online adjustment (§4.3.2).

Placement of cluster slices. Cluster slices are placed to ensure balanced utilization across PUs. Considering the constraint of pairwise bus ownership switching, we group adjacent PUs and distribute slices evenly among PU pairs, ensuring that the two adjacent PUs are always active simultaneously. We define a heat upper limit for each PU pair. The heat upper limit is a hyperparameter tradeoff the time spent on generating the placement with the degree of load balancing. Within this upper limit, all cluster slices are greedily assigned to maintain balanced workload distribution. In our practice, this simple greedy strategy can already achieve a well load balance; please see the evaluation section.

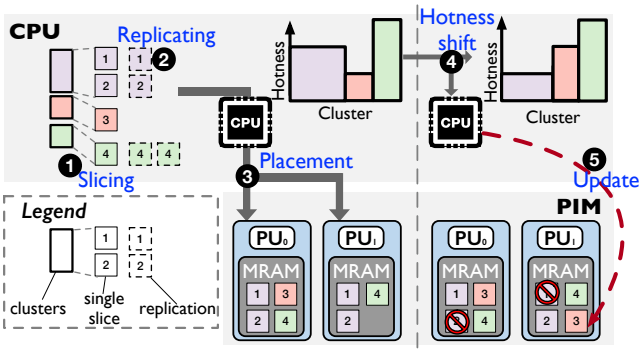


Figure 8: Design of per-PU query dispatching.

4.3.2 Live adjustment of data placement

Over time, cluster hotness may shift, necessitating real-time adjustments to the data placement. Fortunately, the per-PU scheduling design of the persistent kernel enables us to redistribute clusters without shutting down the PIM.

Hotness shifting detection. We use a lightweight monitoring mechanism that tracks cluster access frequency in real time. Specifically, each cluster is assigned a frequency counter that increments when a query involves the cluster. We define a time window and threshold used to detect significant changes in cluster popularity. For example, within the past 1,000 queries (pre-defined time window), if the ratio of a cluster's access frequency compared to the previous time window exceeds a factor of 2 (pre-defined threshold), whether the change is an increase or a decrease, it is considered that the cluster's heat has changed significantly.

Adjustment strategy based on updated hotness. 1) For increased cluster hotness, additional replicas are created on underutilized PUs based on the updated hotness. Queries are then dynamically redirected to these new replicas. 2) For decreased cluster hotness, we remove redundant replicas to free up MRAM space; we prioritize the removal of replicas on heavily loaded PUs. Freed memory slots can be reallocated to clusters with increasing hotness.

4.3.3 Online request dispatching

We maintain a mapping table that records the cluster ID to PU IDs (i.e., all replicas) on the host side. The dispatcher always selects the replica with the lightest current load to send the request. The load of PUs are denoted by the queue depth of message queues.

5 Evaluation

5.1 Experimental Setup

The experiments are conducted on a server equipped with UPMEM modules. The server features two Intel Xeon Silver 4210 processors, providing a total of 20 physical cores running at 2.4 GHz. It is configured with 128 GB of DDR4 memory distributed across 4 DIMM slots. The UPMEM setup consists of 20 DIMM modules, containing a total of 2,560 PUs running at 400 MHz. Each PU supports 16 threads. The server runs Ubuntu 22.04 with kernel version 5.15. The host program is developed using the GCC 11.4.0 compiler, while the PIM program is built with UPMEM SDK 2024.2.0, based on Clang 12.0.0. Additionally, we use an NVIDIA RTX A6000 for evaluating GPU-based ANNS systems.

Competitors:We compare PIMANN with the following ANNS systems.

- Faiss-CPU [22]: It is a popular approximate nearest neighbor search library developed by Meta.
- PIMANN-Batch: There exists one UPMEM baseline system named DRIM-ANN [12], which leverages the batching paradigm, but this work does not provide open-source code. Therefore, we rigorously implemented it according to its paper and named it PIMANN-Batch in our paper.
- Faiss-GPU [22]: It is the GPU-capable version of Faiss, which stores all vectors in GPU memory and performs their computations directly on the GPU.

As for FPGA-based systems [34, 70], because they are not open source, and work [34] requires a customized IP core, for which we do not have the necessary experimental setup, we are unable to include a comparison with these systems.

Datasets: We use the following two large-scale datasets for evaluation:

- SIFT-1B [62]: It consists of 1 billion 128-dimensional vectors encoded into 32 segments.
- SPACE-1B [3]: It consists of 1 billion 100-dimensional vectors encoded into 20 dimensions.

Metrics: We compare different solutions based on: throughput (QPS), latency, and tail latency under the same recall@10. Here, recall@10 refers to the recall calculated for the top 10 items returned by ANNS systems. For Faiss-GPU, we also compare the QPS/Watt to evaluate the energy efficiency of different systems. And we compare the QPS/$ to evaluate the cost efficiency.

Unless otherwise specified, we use the following default experimental configuration: selective replication defaults to using all MRAM memory; the number of clusters in IVF is set to 4096; the number of coroutine is 4; the slice size is 100 K; the default dataset is SPACE-1B; the default achieved recall@10 is 0.9.

5.2 Performance vs. Recall

We evaluate the throughput and latency of PIMANN, Faiss-CPU and PIMANN-Batch. PIMANN significantly outperforms the other two in all cases.

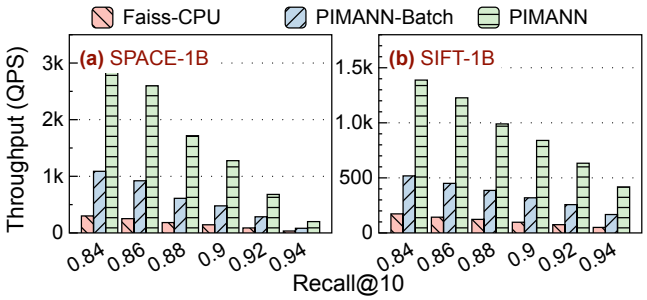


Figure 9: (Exp #1) Throughput under different recalls.

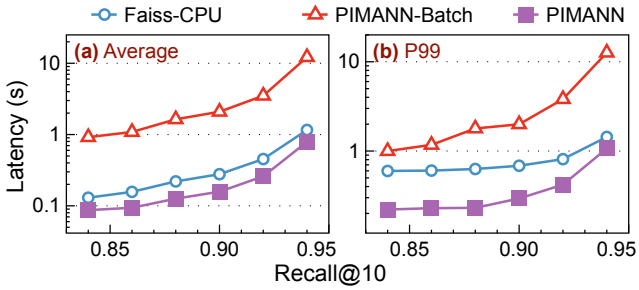


Figure 10: (Exp #2) Latency under different recalls. *(a) Average latency, (b) Tail latency*

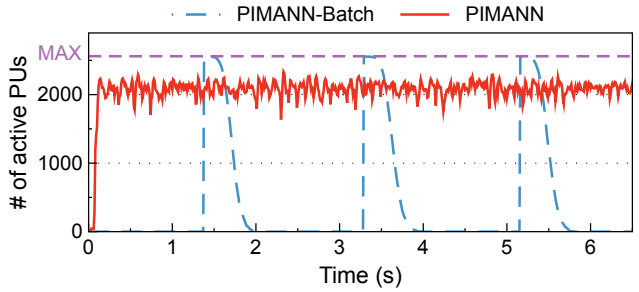


Figure 11: (Exp #3) The number of active PUs over time.

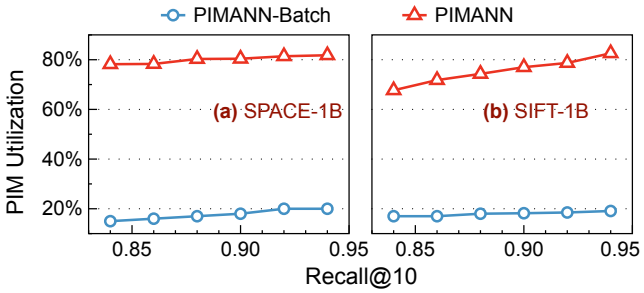


Figure 12: (Exp #3) PIM utilization of different systems. *Utilization is defined as the integral of the number of active PUs over time.*

Exp #1: Overall throughput. Figure 9 shows the throughput of PIMANN, Faiss-CPU, and PIMANN-Batch on SPACE-1B and SIFT-1B. We can make the following observations:

1) The throughput of PIMANN-Batch is 2.4-3.7× higher than that of Faiss-CPU. This is because batch takes advantage of the High Memory Bandwidth of UPMEM.

2) Compared with PIMANN-Batch, introducing per-PU scheduling (PIMANN) can further boost throughput by 2.4-2.9×. This is because PIMANN eliminates inter-batch under-utilization caused by batch gang scheduling, and alleviates intra-batch under-utilization caused by PU load imbalance within a batch. As a result, the QPS of PIMANN improves by 5.9-10.4× compared with that of Faiss-CPU.

3) For cases with low recall (≤ 0.88), the QPS of all three systems on SPACE-1B is higher than on SIFT-1B; for cases with high recall, the opposite is true. This is because, in low-recall cases, fewer clusters need to be queried on SPACE-1B compared to SIFT-1B to achieve the same recall rate, whereas in high-recall cases, the reverse holds true.

Exp #2: End-to-end latency. Figure 10 shows the average latency and the P99 tail latency of Faiss-CPU, PIMANN-Batch, and PIMANN under different levels of search recall requirements, ranging from 0.84 to 0.94. We can find that PIMANN exhibited significantly superior performance compared to CPU and PIMANN-Batch.

Compared to Faiss-CPU, although PIMANN-Batch is equipped with high-parallel UPMEM and enjoy a 2.4-3.7× throughput boost (Exp #1), its average latency is approximately 7.1 to 10.5× higher and its tail latency is about 1.7-8.8× higher, mainly due to the additional inter-batch blockage latency. In contrast, PIMANN eliminates this additional latency by enabling fine-grained bus sharing, reducing average latency by 32-43% and tail latency by 26-63%. The independent per-PU scheduling enables the independent scheduling of tasks, reducing the end-to-end per-query latency.

5.3 PIM utilization

Exp #3: PIM utilization. In this experiment, we revisit our motivation to investigate the PIM utilization of different systems. Figure 11 shows the number of active PUs over time. An active PU refers to a PU that is currently executing computing tasks. Here the used dataset is SPACE-1B; the recall rate is 0.9; other cases share a similar conclusion.

As shown in Figure 11, the number of active PUs in PIMANN (the red line) remains basically stable over time, achieving a utilization around 80%. This stability is achieved because PIMANN overcomes the limitations of batch scheduling. When the CPU receives a query task and completes the corresponding stage, it can immediately send the task to the PU. After the PU finishes the task, it can promptly return the result to the CPU and wait for new tasks. The PU remains active from the time it receives a task until it completes the task. In contrast, in PIMANN-Batch, the number of active PUs exhibits periodic fluctuations (the blue dashed

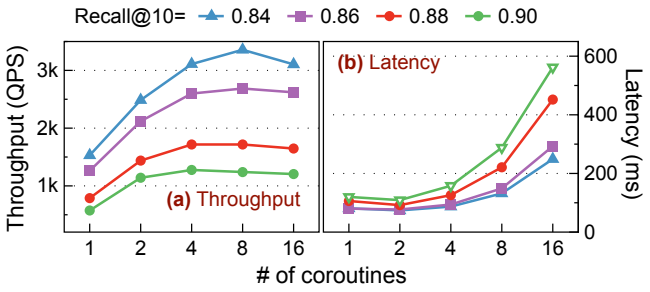


Figure 13: (Exp #4): Effectiveness of coroutine-based bus ownership switching.

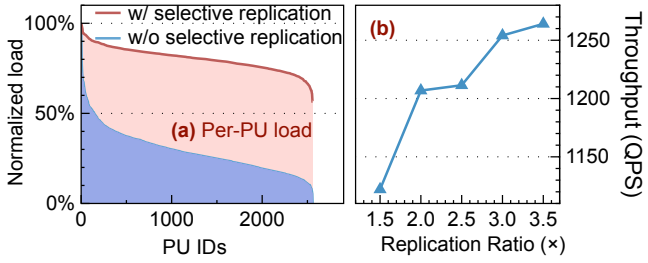


Figure 14: (Exp #5): Effect of per-PU dispatching. *(a) Per-PU load w/ or w/o selective replication, (b) Throughput with different capacity budgets.*

line). At the start of each batch, when all PUs are initially active, the number of active PUs peaks. Later, during the data copying phase between batches, all PUs yield the bus ownership and become inactive. Additionally, within a batch, due to the imbalanced task load among PUs, some PUs complete their tasks early and enter a waiting state, further reducing the number of active PUs.

Furthermore, we integrate the number of active PUs over time, and based on this, calculate the PIM utilization rates of the two systems under different datasets and recall conditions; please see Figure 12. Under all datasets and all recall rates, PIMANN can achieve 65-83% of the PIM utilization, far surpassing the batch paradigm (∼20%). This shows the effectiveness and efficiency of PIMANN's designs. The persistent PIM kernel technique eliminates the idle state between batches. On the other hand, the per-PU dispatching technique makes the load of each PU sufficiently balanced.

5.4 Techniques

Exp #4: Coroutine-based bus ownership switching. Figure 13 shows the performance with different coroutine counts. We can see that: 1) Compared to PIMANN without coroutine optimization (i.e., coroutine count=1), PIMANN with this optimization can yield around 3× better throughput. A moderate coroutine count can slightly reduce request latency, as coroutines help hide the latency associated with shared bus ownership switching. 2) However, an excessive number of coroutines increases the system's scheduling overhead, which negatively impacts both throughput and latency.

Exp #5: Effect of selective replication. Figure 14a shows the impact of selective replication technique in per-PU dispatching of PIMANN. Compared to the system without replication, which causes a single PU to become a bottleneck, PIMANN can use the replication technique to prevent hotspot clusters from concentrating on a few PUs, achieving better load balancing. Figure 14b shows the throughput under different memory capacity budgets, and throughput increases with the budget.

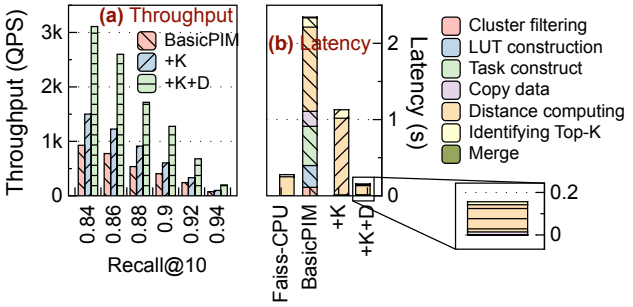


Figure 15: (Exp #6): Contributions of techniques. *(a) Throughput, (b) Latency breakdown of a query. Design techniques are cumulative. BasicPIM: a version of PIMANN-Batch w/o selective replication; K: persistent PIM kernel; D: per-PU query dispatching.*

Exp #6: Contributions of individual techniques. Figure 15 shows the contribution of each individual technique to the end-to-end performance. We only show the result of SPACE-1B, while SIFT-1B shares a similar conclusion.

We evaluate the throughput (Figure 15a) and latency breakdown (Figure 15b) of the system by gradually adding the persistent PIM kernel (abbr. as K) and the per-PU query dispatching technique (abbr. as D) to a baseline version of PIMANN-batch (called BasicPIM, which forbids the selective data replication optimization).

From Figure 15, we can observe that: 1) in all cases, both techniques have achieved positive performance gains. Specifically, the persistent PIM kernel technique has increased the throughput by 30% to 70%; by introducing the per-PU query dispatching, it further increases the throughput by 88% to 112%. 2) The persistent PIM kernel technique reduces the time in all stages except for distance computation, since it overcomes the inter-batch wait time of batch scheduling. 3) By introducing per PU query scheduling, distance computation time is also greatly reduced. The main reason is that this technique leads to a more balanced load distribution between PUs.

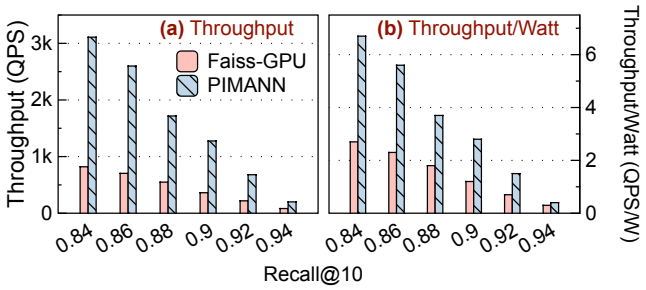


Figure 16: (Exp #7) Comparision with GPU-based system. *(a) Throughput, (b) Power Efficiency.*

Table 1: (Exp #8): Cost efficiency comparison of different solutions.

Solution	Price ($)	QPS	QPS/$
Faiss-CPU	1,500	144	0.096
Faiss-GPU	9,685	478	0.049
PIMANN	5,473	1,276	0.233

5.5 Comparison with GPU-based systems

Exp #7: Comparison with Faiss-GPU. We compare PIMANN and Faiss-GPU in terms of performance and power efficiency on the SPACE-1B dataset; see Figure 16. It is worth noting that there is a huge gap in peak computing resources between UPMEM and GPU. The GPU is equipped with 10,752 cores and has a computing power of up to 38.7 TFLOPS, while UPMEM can only achieve an integer processing capacity of 1 TOPS [65]. In terms of power consumption, the total power consumption of UPMEM is 462W [24]. In contrast, the power consumption of the RTX A6000 GPU is approximately 300W. The current high power consumption of UPMEM is mainly influenced by hardware fabrication, but it can be significantly reduced with future PIM designs. However, even with such a large power gap, as shown in Figure 16, the throughput of PIMANN is still 2.4 to 3.7× compared with that of the GPU, and the power efficiency is increased by 1.6 to 2.5× compared with that of the GPU. It is worth noting that Faiss-GPU requires storing all vectors in the GPU memory, so PIMANN has a much better capacity scalability.

5.6 Cost efficiency

Exp #8: Cost efficiency. As shown in Table 1, we evaluate the cost-effectiveness of three solutions using QPS/$. PIMANN outperforms both Faiss-CPU and Faiss-GPU. Compared to Faiss-CPU, PIMANN's price is higher, but its throughput is significantly improved, leading to a 2.4× increase in cost-effectiveness. Compared to Faiss-GPU, PIMANN achieves both the lower price and the higher throughput, resulting in a 4.8× improvement in cost-effectiveness.

6 Discussion

Available for the next generation of UPMEM: Our approach is fundamentally decoupled from specific architectural implementation details of the UPMEM platform. Instead, it leverages intrinsic hardware capabilities that are inherent to the underlying PIM infrastructure. Consequently, potential architectural modifications to UMPEM would not necessitate revisions to our proposed design.

The core of our methodology relies exclusively on the standardized control interface exposed by each PU. As it serves as the fundamental mechanism through which the host CPU issues critical control commands like launching PIM kernels, it is guaranteed to persist across future iterations of the UMPEM platform.

Other PIM architectures: We evaluate our framework on the UPMEM platform, which is the first commercially available real-world PIM hardware. And we observe that other PIM architectures [41,45,52,56,58] also exhibit similar characteristics to UPMEM. In future work, we will evaluate our framework on these platforms as conditions permit.

7 Related Work

We organize the related work into two types: 1) with similar goal and different mechanism, which uses other hardware to optimize ANNS systems, and 2) with different goal and similar mechanism, which uses PIM for other applications.

ANNS systems built on CPU, GPU, or FPGA. Faiss [22] and Milvus [67] are two widely used open-source vector databases. Both support CPU and GPU, but they are also restricted by the memory capacity.

DiskANN [35] and HM-ANN [60] all expand the memory of a single machine to support vector storage with a larger capacity. DiskANN extends DRAM with SSDs, forming a two-layer memory structure consisting of DRAM and SSD. It modifies the neighbor pruning algorithm of nodes, enabling free control over the scale of edge pruning, and adapts to the poor random access performance of SSDs through techniques such as cache prefetching. HM-ANN expands memory using heterogeneous memory and maps the hierarchical design of HNSW onto the memory hierarchy. The upper layer is placed on DRAM, and the bottom layer is placed on heterogeneous memory.

CAGRA [55], BANG [40] and RUMMY [71] use GPU to build ANN systems. CAGRA has designed a new indexing algorithm to make full use of the parallel processing capabilities of the GPU: it first calculates the kNN graph [21], and then prunes and reorders based on the kNN graph to build the index. However, the expensive GPU memory limits the scale of vectors. BANG focuses on graph-based ANNS, which requires much more memory than cluster-based systems. To extend GPU memory, BANG uses a storage structure similar to that of DiskANN. It stores compressed vectors in the

GPU video memory and the full set of vectors in DRAM for full distance computation. RUMMY solves the problem of redundant and inefficient data transmission between video memory and memory through techniques such as pipeline rearrangement.

Apart from CPUs and GPUs, there are also works that use FPGAs and SmartSSDs to build ANN systems. FANNS [37] achieves lower costs and power consumption than GPU through multi-stage automatic tuning, but it is still limited to a single card. DF-GAS [70] extends vector retrieval to multiple FPGAs, improving memory access efficiency through data prefetching and latency optimization. It also enhances query performance by enabling parallel searches that integrate both the entire graph and its subgraphs. Vstore [50] utilizes computable SSDs to complete graph-based approximate retrieval. SmartSSDs [63] are extended to multiple computable SSD devices. It has designed a multi-disk task scheduling strategy to ensure load balancing and a search pruning strategy to reduce unnecessary computations. CXL-ANNS [34] uses the CXL link protocol to connect memory pools and adopts caching and prefetching techniques to reduce the impact of the high latency of the CXL pooled memory.

Other systems using emerging PIM hardware. Numerous explorations have been made to build applications in different fields on PIM. 1) In the field of databases, PID-Join [51] utilizes PIM to implement the join operator and proposes a method for rapid communication among PIM nodes. Work [8] utilized PIM to address large table scan. Work [9] used PIM for query compilation. PIM-tree [38] designed an ordered index for PIM systems. 2) In the field of programming library, TransPimLib [27] provided a library of transcendental functions, such as trigonometric functions, logarithms, powers, etc. PID-Comm [54] proposes a framework for collective inter-PE communication designed for PIM-enabled DIMMs. 3) In the field of machine learning, work [16] implements two CNN algorithms, eBNN and YOLO3, on UPMEM. 4) In the field of scientific computing, UPMEM BLAST [43] deploys a molecular biology software on UPMEM. SparseP [25] provided a sparse matrix-vector multiplication library.

8 Conclusion

This paper presents PIMANN, an ANNS system designed to fully exploit the potential of UPMEM. By introducing the persistent PIM kernel and per-PU dispatching techniques, PIMANN effectively addresses the inefficiencies of traditional batching scheduling in PIM-based ANNS systems. The experimental results demonstrate its superiority in terms of throughput, latency, and resource utilization compared to existing CPU-based, GPU-based, and batching-paradigm-based solutions. PIMANN not only provides a promising approach for accelerating ANNS on PIM hardware but also offers valuable insights for future research in the intersection of memory-centric computing applications.

Acknowledgement

We appreciate our shepherd and anonymous reviewers' constructive comments for improving the quality of this paper. This work is supported by Beijing Natural Science Foundation (Grant No. 4254083), National Natural Science Foundation of China (Grant No. U23A20299), PCL-CMCC Foundation for Science and Innovation (Grant No. 2024ZY1C0030) and Kuaishou.

A Artifact Appendix

Abstract

The artifact consists of the C++ code of PIMANN and baseline systems used in evaluation. For ease of experiment reproduction and result visualization, we also provide runner scripts and plotter scripts accordingly. It is intended for validating the claims made in the paper and facilitating further research on Processing-in-Memory (PIM) hardware.

Scope

At a high level, the artifact allows its users to validate the following major claims in the paper:

- *Major Claim 1:* PIMANN outperforms baselines in terms of query throughput and latency. (Exp #1,#2,#7,#8)
- *Major Claim 2:* PIMANN overcomes batch scheduling limitations via fine-grained per-PU scheduling. (Exp #3,#6a)
- *Major Claim 3:* PIMANN's techniques synergistically improve performance.(Exp #4,#5,#6b)

Contents

The artifact contains source code, scripts, and README files. Below, we explain the contents aside from the README.

Source codes. The major part of source codes are well documented, facilitating further research.

```
common/              # shared files for both DPU and host programs
dpu/                 # DPU kernel functions
host/                # host programs interacting with DPU kernels
third-party/
   ├── upmem-2024.2.0-Linux-x86_64/  # modified UPMEM-SDK
   └── faiss_upmem/                  # modified FAISS
AE/                  # test scripts and plotting scripts
main.cpp             # main program entry point
```

Figure 17: Directory structure of our project.

Scripts. The scripts directory contains runner scripts for running experiments and plotter scripts for visualizing the results. Scripts directly reproducing the experiments (i.e., AE scripts) in the paper are in the ae subdirectory. Scripts not in the ae subdirectory serve as building blocks of the AE scripts, such as running a single experiment.

Hosting

PIMANN's artifact repository is hosted on GitHub at PIM-ANNS on the main branch. The first usable commit version is the initial commit (af44d9f). However, due to possible future bugfixes and updates, please always use the latest commit.

Requirements

Hardware Requirements: To run this project, a server equipped with UPMEM hardware (`https://www.upmem.com/`) is required.
Software Requirements: Additionally, we also need setup the following software environment.

- UPMEM-SDK: This project uses a modified version of UPMEM-SDK based on the 2024.2 release.

```
cd third-party/upmem-2024.2.0-Linux-x86_64
cd src/backends
bash ./install.sh
```

- FAISS: The IVFPQ index algorithm reuses portions of the FAISS codebase. For better UPMEM compatibility, we provide a modified version of FAISS based on: `https://github.com/facebookresearch/faiss`. Install like FAISS.

```
cd third-party/faiss_upmem
cmake -B build .
make -C build -j faiss
make -C build install
```

- Boost library: This project utilizes Boost's coroutine library. Thus, please install the libboost with the following commands.

```
sudo apt-get update
sudo apt install libboost-all-dev
```

Experiment workflow

Building PIMANN from source

Please simply use the CMake building system.

```
cmake -B build .
cd build
make -j
```

Hello-world example

To verify that everything is prepared, you can run a hello-world example that verifies PIMANN's functionality, please run the following command:

```
bash AE/hello_world.sh
```

It will run for approximately 1 minute and, on success, output something like below:

```
yyyy-mm-dd hh:mm:ss
json_path: PIMANN/config.json
query path is dataset/space/query10K.i8bin
query_num: 10000, dim: 100
searching SPACE1M, nprobe = 11
The command ./main 11 completed successfully
```

If you can see this output, then everything is OK, and you can start running the artifact.

Run all experiments

We provide convenient scripts for running either all experiments collectively (#1) or individual experiments selectively (#2).

#1: Running the all-in-one script. We provide an all-in-one AE script for running all experiments end-to-endly:

```
AE/run_all.sh
```

This script will run for approximately 8 hours and store all results in the "AE" directory.

#2: Running specific experiments. If you wish to replicate only specific experiments, we provide nine separate scripts corresponding to different experimental settings. These scripts, located in the `AE/exps/expX.sh` files (where X = 1, 2, ..., 8), can be used to reproduce all the figures presented in our paper. The name of all scripts (i.e., `expX.sh`) are aligned with those presented in our main paper.

If you want to run individual experiments, please refer to these script files and the comments in them (which describes the relationship between experiments and figures/tables).

Estimated running hours of experiments are shown in Table 2.

Table 2: Estimated time of all experiments

Experiment Description	Time(hours)
Exp #1: Overall throughput	1.5
Exp #2: End-to-end latency	1.5
Exp #3: PIM utilization	1.5
Exp #4: Coroutine	1.0
Exp #5: Selective replication	0.5
Exp #6: Individual techniques	1.0
Exp #7: Comparison with Faiss-GPU	0.5
Exp #8: Cost efficiency	0.2

Plot all figures & tables

We provide two alternative ways to visualize the experimental results.

#1: (Recommended) All-in-one jupyter notebook for Visual Studio Code users. Please install the Jupyter extension in VSCode. Then, please open AE/figures/plot.ipynb. Please activate the virtual environment (.venv/bin/python) by running the following commands.

```
source .venv/bin/activate
```

Then, you can run each cell from top to bottom. Each cell will plot a figure or table (see Figure 18). Titles of these figures and tables are consistent with those in the paper.

#2: Traditional python plotting scripts.

We provide a traditional plotter script. Please run it in the AE directory:

```
cd AE/figures
python3 plot.py
```

The command above will plot all figures and tables by default, and the results will be stored in the AE/figures directory (Figure 19). So, please ensure that you have finished running the all-in-one AE script before running the plotter.

The plotter further allows users to specify particular figures or tables to generate by providing supplementary command-line arguments. For example:

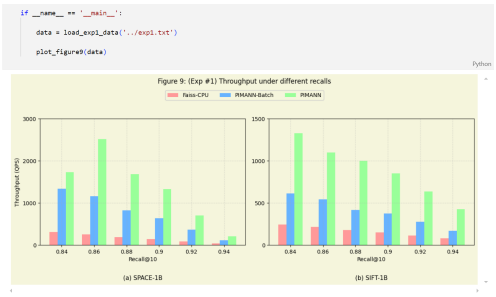


Figure 18: Visualizing results with the all-in-one Jupyter notebook.

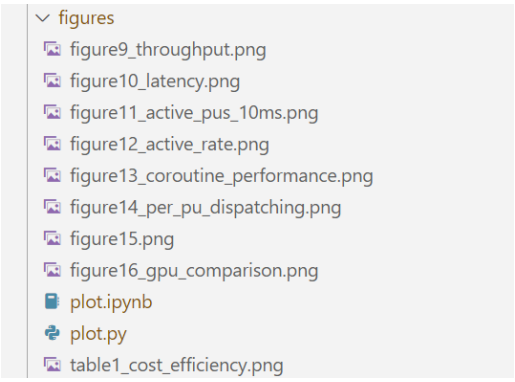


Figure 19: Visualizing results with the conventional script.

```
python3 plot.py exp1 exp2
```

Please refer to plot.py for accepted arguments.

```
python3 plot.py help
```

References

[1] Microsoft Research talk: Approximate nearest neighbor search systems at scale. [EB/OL].

[2] The Software Development Kit for programming and using the DPU provided by the UPMEM Acceleration platform.

[3] SPACEV1B: A billion-Scale vector dataset for text descriptors. https://github.com/microsoft/SPTAG/tree/main/datasets/SPACEV1B, 2021.

[4] Ameer MS Abdelhadi, Christos-Savvas Bouganis, and George A Constantinides. Accelerated approximate nearest neighbors search through hierarchical product quantization. In *2019 International Conference on Field-Programmable Technology (ICFPT)*, pages 90–98. IEEE, 2019.

[5] Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. A scalable processing-in-memory accelerator for parallel graph processing. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, pages 105–117, 2015.

[6] Junwhan Ahn, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. Pim-enabled instructions: A low-overhead, locality-aware processing-in-memory architecture. *ACM SIGARCH Computer Architecture News*, 43(3S):336–348, 2015.

[7] Dimitrios Androutsos, Konstantinos N Plataniotis, and Anastasios N Venetsanopoulos. A novel vector-based approach to color image retrieval using a vector angular-based distance measure. *Computer Vision and Image Understanding*, 75(1-2):46–58, 1999.

[8] Alexander Baumstark, Muhammad Attahir Jibril, and Kai-Uwe Sattler. Accelerating large table scan using processing-in-memory technology. *Datenbank-Spektrum*, 23(3):199–209, 2023.

[9] Alexander Baumstark, Muhammad Attahir Jibril, and Kai-Uwe Sattler. Adaptive query compilation with processing-in-memory. In *2023 IEEE 39th International Conference on Data Engineering Workshops (ICDEW)*, pages 191–197. IEEE, 2023.

[10] Amirali Boroumand, Saugata Ghose, Minesh Patel, Hasan Hassan, Brandon Lucia, Rachata Ausavarungnirun, Kevin Hsieh, Nastaran Hajinazar, Krishna T Malladi, Hongzhong Zheng, et al. Conda: Efficient cache coherence support for near-data accelerators. In *Proceedings of the 46th International Symposium on Computer Architecture*, pages 629–642, 2019.

[11] Amirali Boroumand, Saugata Ghose, Minesh Patel, Hasan Hassan, Brandon Lucia, Kevin Hsieh, Krishna T Malladi, Hongzhong Zheng, and Onur Mutlu. Lazypim: An efficient cache coherence mechanism for processing-in-memory. *IEEE Computer Architecture Letters*, 16(1):46–50, 2016.

[12] Mingkai Chen, Tianhua Han, Cheng Liu, Shengwen Liang, Kuai Yu, Lei Dai, Ziming Yuan, Ying Wang, Lei Zhang, Huawei Li, and Xiaowei Li. Drim-ann: An approximate nearest neighbor search engine based on commercial dram-pims, 2024.

[13] Sitian Chen, Amelie Chi Zhou, Yucheng Shi, Yusen Li, and Xin Yao. Memanns: Enhancing billion-scale anns efficiency with practical pim hardware, 2024.

[14] Rongxin Cheng, Yifan Peng, Xingda Wei, Hongrui Xie, Rong Chen, Sijie Shen, and Haibo Chen. Characterizing the dilemma of performance and index size in billion-scale vector search and breaking it with second-tier memory, 2024.

[15] Benjamin Y Cho, Jeageun Jung, and Mattan Erez. Accelerating bandwidth-bound deep learning inference with main-memory accelerators. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14, 2021.

[16] Prangon Das, Purab Ranjan Sutradhar, Mark Indovina, Sai Manoj Pudukotai Dinakarrao, and Amlan Ganguly. Implementation and evaluation of deep neural networks in commercially available processing in memory hardware. In *2022 IEEE 35th International System-on-Chip Conference (SOCC)*, pages 1–6. IEEE, 2022.

[17] Stephen Deering, Deborah L Estrin, Dino Farinacci, Van Jacobson, Ching-Gung Liu, and Liming Wei. The pim architecture for wide-area multicast routing. *IEEE/ACM transactions on networking*, 4(2):153–162, 1996.

[18] Alexandar Devic, Siddhartha Balakrishna Rai, Anand Sivasubramaniam, Ameen Akel, Sean Eilert, and Justin Eno. To pim or not for emerging general purpose processing in ddr memory systems. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, pages 231–244, 2022.

[19] T Dharani and I Laurence Aroquiaraj. Content based image retrieval system using feature classification with modified knn algorithm. *arXiv preprint arXiv:1307.4717*, 2013.

[20] Safaa Diab, Amir Nassereldine, Mohammed Alser, Juan Gómez Luna, Onur Mutlu, and Izzat El Hajj. A framework for high-throughput sequence alignment using real processing-in-memory systems. *Bioinformatics*, 39(5):btad155, 2023.

[21] Wei Dong, Charikar Moses, and Kai Li. Efficient k-nearest neighbor graph construction for generic similarity measures. In *Proceedings of the 20th international conference on World wide web*, pages 577–586, 2011.

[22] Matthijs Douze, Alexandr Guzhva, Chengqi Deng, Jeff Johnson, Gergely Szilvasy, Pierre-Emmanuel Mazaré, Maria Lomeli, Lucas Hosseini, and Hervé Jégou. The faiss library. *arXiv preprint arXiv:2401.08281*, 2024.

[23] Duncan G Elliott, Michael Stumm, W Martin Snelgrove, Christian Cojocaru, and Robert McKenzie. Computational ram: Implementing processors in memory. *IEEE Design & Test of Computers*, 16(1):32–41, 1999.

[24] Yann Falevoz and Julien Legriel. Energy efficiency impact of processing in memory: A comprehensive review of workloads on the upmem architecture. In *European Conference on Parallel Processing*, pages 155–166. Springer, 2023.

[25] Christina Giannoula, Ivan Fernandez, Juan Gómez Luna, Nectarios Koziris, Georgios Goumas, and Onur Mutlu. Sparsep: Towards efficient sparse matrix vector multiplication on real processing-in-memory architectures. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 6(1):1–49, 2022.

[26] Christina Giannoula, Nandita Vijaykumar, Nikela Papadopoulou, Vasileios Karakostas, Ivan Fernandez, Juan

Gómez-Luna, Lois Orosa, Nectarios Koziris, Georgios Goumas, and Onur Mutlu. Syncron: Efficient synchronization support for near-data-processing architectures. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 263–276. IEEE, 2021.

[27] Juan Gómez-Luna, Yuxin Guo, Geraldo F Oliveira, Mohammad Sadrosadati, and Onur Mutlu. Transpimlib: A library for efficient transcendental functions on processing-in-memory systems. *arXiv preprint arXiv:2304.01951*, 2023.

[28] Juan Gómez-Luna, Izzat El Hajj, Ivan Fernandez, Christina Giannoula, Geraldo F Oliveira, and Onur Mutlu. Benchmarking a new paradigm: An experimental analysis of a real processing-in-memory architecture. *arXiv preprint arXiv:2105.03814*, 2021.

[29] Nastaran Hajinazar, Geraldo F Oliveira, Sven Gregorio, João Dinis Ferreira, Nika Mansouri Ghiasi, Minesh Patel, Mohammed Alser, Saugata Ghose, Juan Gómez-Luna, and Onur Mutlu. Simdram: A framework for bit-serial simd processing using dram. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 329–345, 2021.

[30] Guseul Heo, Sangyeop Lee, Jaehong Cho, Hyunmin Choi, Sanghyeon Lee, Hyungkyu Ham, Gwangsun Kim, Divya Mahajan, and Jongse Park. Neupims: Npu-pim heterogeneous acceleration for batched llm inferencing. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pages 722–737, 2024.

[31] Kevin Hsieh, Eiman Ebrahimi, Gwangsun Kim, Niladrish Chatterjee, Mike O'Connor, Nandita Vijaykumar, Onur Mutlu, and Stephen W Keckler. Transparent offloading and mapping (tom) enabling programmer-transparent near-data processing in gpu systems. *ACM SIGARCH Computer Architecture News*, 44(3):204–216, 2016.

[32] Kevin Hsieh, Samira Khan, Nandita Vijaykumar, Kevin K Chang, Amirali Boroumand, Saugata Ghose, and Onur Mutlu. Accelerating pointer chasing in 3d-stacked memory: Challenges, mechanisms, evaluation. In *2016 IEEE 34th International Conference on Computer Design (ICCD)*, pages 25–32. IEEE, 2016.

[33] Huawei. Data Storage 2030. Technical report, 2024.

[34] Junhyeok Jang, Hanjin Choi, Hanyeoreum Bae, Seungjun Lee, Miryeong Kwon, and Myoungsoo Jung. {CXL-ANNS}:{Software-Hardware} collaborative memory disaggregation and computation for {Billion-Scale} approximate nearest neighbor search. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 585–600, 2023.

[35] Suhas Jayaram Subramanya, Fnu Devvrit, Harsha Vardhan Simhadri, Ravishankar Krishnawamy, and Rohan Kadekodi. Diskann: Fast accurate billion-point nearest neighbor search on a single node. *Advances in Neural Information Processing Systems*, 32, 2019.

[36] Hervé Jégou, Romain Tavenard, Matthijs Douze, and Laurent Amsaleg. Searching in one billion vectors: re-rank with source coding. In *2011 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 861–864. IEEE, 2011.

[37] Wenqi Jiang, Shigang Li, Yu Zhu, Johannes de Fine Licht, Zhenhao He, Runbin Shi, Cedric Renggli, Shuai Zhang, Theodoros Rekatsinas, Torsten Hoefler, et al. Co-design hardware and algorithm for vector search. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15, 2023.

[38] Hongbo Kang, Yiwei Zhao, Guy E. Blelloch, Laxman Dhulipala, Yan Gu, Charles McGuffey, and Phillip B. Gibbons. Pim-tree: A skew-resistant index for processing-in-memory. *Proc. VLDB Endow.*, 16(4):946–958, December 2022.

[39] Hongbo Kang, Yiwei Zhao, Guy E Blelloch, Laxman Dhulipala, Yan Gu, Charles McGuffey, and Phillip B Gibbons. Pim-trie: A skew-resistant trie for processing-in-memory. In *Proceedings of the 35th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 1–14, 2023.

[40] Saim Khan, Somesh Singh, Harsha Vardhan Simhadri, Jyothi Vedurada, et al. Bang: Billion-scale approximate nearest neighbor search using a single gpu. *arXiv preprint arXiv:2401.11324*, 2024.

[41] Seongguk Kim, Subin Kim, Kyungjun Cho, Taein Shin, Hyunwook Park, Daehwan Lho, Shinyoung Park, Kyungjune Son, Gapyeol Park, and Joungho Kim. Processing-in-memory in high bandwidth memory (pim-hbm) architecture with energy-efficient and low latency channels for high bandwidth system. In *2019 IEEE 28th Conference on Electrical Performance of Electronic Packaging and Systems (EPEPS)*, pages 1–3, 2019.

[42] Yongkee Kwon, Guhyun Kim, Nahsung Kim, Woojae Shin, Jongsoon Won, Hyunha Joo, Haerang Choi, Byeongju An, Gyeongcheol Shin, Dayeon Yun, et al. Memory-centric computing with sk hynix's domain-specific memory. In *2023 IEEE Hot Chips 35 Symposium (HCS)*, pages 1–26. IEEE Computer Society, 2023.

[43] Dominique Lavenier, Charles Deltel, David Furodet, and Jean-François Roy. *BLAST on UPMEM*. PhD thesis, INRIA Rennes-Bretagne Atlantique, 2016.

[44] Dongjae Lee, Bongjoon Hyun, Taehun Kim, and Minsoo Rhu. Pim-mmu: A memory management unit for accelerating data transfers in commercial pim systems. In *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 627–642. IEEE, 2024.

[45] Seongju Lee, Kyuyoung Kim, Sanghoon Oh, Joonhong Park, Gimoon Hong, Dongyoon Ka, Kyudong Hwang, Jeongje Park, Kyeongpil Kang, Jungyeon Kim, Junyeol Jeon, Nahsung Kim, Yongkee Kwon, Kornijcuk Vladimir, Woojae Shin, Jongsoon Won, Minkyu Lee, Hyunha Joo, Haerang Choi, Jaewook Lee, Donguc Ko, Younggun Jun, Keewon Cho, Ilwoong Kim, Choungki Song, Chunseok Jeong, Daehan Kwon, Jieun Jang, Il Park, Junhyun Chun, and Joohwan Cho. A 1ynm 1.25v 8gb, 16gb/s/pin gddr6-based accelerator-in-memory supporting 1tflops mac operation and various activation functions for deep-learning applications. In *2022 IEEE International Solid-State Circuits Conference (ISSCC)*, volume 65, pages 1–3, 2022.

[46] Suhyun Lee, Chaemin Lim, Jinwoo Choi, Heelim Choi, Chan Lee, Yongjun Park, Kwanghyun Park, Hanjun Kim, and Youngsok Kim. Spid-join: A skew-resistant processing-in-dimm join

algorithm exploiting the bank- and rank-level parallelisms of dimms. *Proc. ACM Manag. Data*, 2(6), December 2024.

[47] Sukhan Lee, Shin-haeng Kang, Jaehoon Lee, Hyeonsu Kim, Eojin Lee, Seungwoo Seo, Hosang Yoon, Seungwon Lee, Kyounghwan Lim, Hyunsung Shin, et al. Hardware architecture and software stack for pim based on commercial dram technology: Industrial product. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 43–56. IEEE, 2021.

[48] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in Neural Information Processing Systems*, 33:9459–9474, 2020.

[49] Cong Li, Zhe Zhou, Yang Wang, Fan Yang, Ting Cao, Mao Yang, Yun Liang, and Guangyu Sun. Pim-dl: Expanding the applicability of commodity dram-pims for deep learning via algorithm-system co-optimization. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS '24, page 879–896, New York, NY, USA, 2024. Association for Computing Machinery.

[50] Shengwen Liang, Ying Wang, Ziming Yuan, Cheng Liu, Huawei Li, and Xiaowei Li. Vstore: in-storage graph based vector search accelerator. In *Proceedings of the 59th ACM/IEEE Design Automation Conference*, pages 997–1002, 2022.

[51] Chaemin Lim, Suhyun Lee, Jinwoo Choi, Jounghoo Lee, Seongyeon Park, Hanjun Kim, Jinho Lee, and Youngsok Kim. Design and analysis of a processing-in-dimm join algorithm: A case study with upmem dimms. *Proceedings of the ACM on Management of Data*, 1(2):1–27, 2023.

[52] Haifeng Liu, Long Zheng, Yu Huang, Jingyi Zhou, Chaoqiang Liu, Runze Wang, Xiaofei Liao, Hai Jin, and Jingling Xue. Enabling efficient large recommendation model training with near cxl memory processing. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, pages 382–395, 2024.

[53] Ravi Nair. Evolution of memory architecture. *Proceedings of the IEEE*, 103(8):1331–1345, 2015.

[54] Si Ung Noh, Junguk Hong, Chaemin Lim, Seongyeon Park, Jeehyun Kim, Hanjun Kim, Youngsok Kim, and Jinho Lee. Pidcomm: A fast and flexible collective communication framework for commodity processing-in-dimm devices. *arXiv preprint arXiv:2404.08871*, 2024.

[55] Hiroyuki Ootomo, Akira Naruse, Corey Nolet, Ray Wang, Tamas Feher, and Yong Wang. Cagra: Highly parallel graph construction and approximate nearest neighbor search for gpus. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*, pages 4236–4247. IEEE, 2024.

[56] Jaehyun Park, Jaewan Choi, Kwanhee Kyung, Michael Jaemin Kim, Yongsuk Kwon, Nam Sung Kim, and Jung Ho Ahn. Attacc! unleashing the power of pim for batched transformer-based generative model inference. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS '24, page 103–119, New York, NY, USA, 2024. Association for Computing Machinery.

[57] Sang-Soo Park, KyungSoo Kim, Jinin So, Jin Jung, Jonggeon Lee, Kyoungwan Woo, Nayeon Kim, Younghyun Lee, Hyungyo Kim, Yongsuk Kwon, et al. An lpddr-based cxl-pnm platform for tco-efficient inference of transformer-based large language models. In *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 970–982. IEEE, 2024.

[58] Sang-Soo Park, KyungSoo Kim, Jinin So, Jin Jung, Jonggeon Lee, Kyoungwan Woo, Nayeon Kim, Younghyun Lee, Hyungyo Kim, Yongsuk Kwon, Jinhyun Kim, Jieun Lee, Yeon-Gon Cho, Yongmin Tai, Jeonghyeon Cho, Hoyoung Song, Jung Ho Ahn, and Nam Sung Kim. An lpddr-based cxl-pnm platform for tco-efficient inference of transformer-based large language models. In *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 970–982, 2024.

[59] Christian Platzer and Schahram Dustdar. A vector space search engine for web services. In *Third European Conference on Web Services (ECOWS'05)*, pages 9–pp. IEEE, 2005.

[60] Jie Ren, Minjia Zhang, and Dong Li. Hm-ann: Efficient billion-point nearest neighbor search on heterogeneous memory. *Advances in Neural Information Processing Systems*, 33:10672–10684, 2020.

[61] Samsung. "8Gb C-die DDR4 SDRAM x16," 2017. https://download.semiconductor.samsung.com/resources/user-manual/x16%20only_8G_C_DDR4_Samsung_Spec_Rev1.5_Apr.17.pdf. [Accessed 09-01-2025].

[62] Harsha Vardhan Simhadri, George Williams, Martin Aumüller, Matthijs Douze, Artem Babenko, Dmitry Baranchuk, Qi Chen, Lucas Hosseini, Ravishankar Krishnaswamny, Gopal Srinivasa, et al. Results of the neurips'21 challenge on billion-scale approximate nearest neighbor search. In *NeurIPS 2021 Competitions and Demonstrations Track*, pages 177–189. PMLR, 2022.

[63] Bing Tian, Haikun Liu, Zhuohui Duan, Xiaofei Liao, Hai Jin, and Yu Zhang. Scalable billion-point approximate nearest neighbor search using {SmartSSDs}. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*, pages 1135–1150, 2024.

[64] SM Tonmoy, SM Zaman, Vinija Jain, Anku Rani, Vipula Rawte, Aman Chadha, and Amitava Das. A comprehensive survey of hallucination mitigation techniques in large language models. *arXiv preprint arXiv:2401.01313*, 2024.

[65] UPMEM. Upmem website. https://www.upmem.com/, 2025.

[66] Vaishali S Vairale and Samiksha Shukla. Recommendation of food items for thyroid patients using content-based knn method. In *Data Science and Security: Proceedings of IDSCS 2020*, pages 71–77. Springer, 2021.

[67] Jianguo Wang, Xiaomeng Yi, Rentong Guo, Hai Jin, Peng Xu, Shengjun Li, Xiangyu Wang, Xiangzhou Guo, Chengming Li, Xiaohai Xu, et al. Milvus: A purpose-built vector data

management system. In *Proceedings of the 2021 International Conference on Management of Data*, pages 2614–2627, 2021.

[68] Mengzhao Wang, Xiaoliang Xu, Qiang Yue, and Yuxiang Wang. A comprehensive survey and experimental comparison of graph-based approximate nearest neighbor search. *arXiv preprint arXiv:2101.12631*, 2021.

[69] Chuangxian Wei, Bin Wu, Sheng Wang, Renjie Lou, Chaoqun Zhan, Feifei Li, and Yuanzhe Cai. Analyticdb-v: a hybrid analytical engine towards query fusion for structured and unstructured data. *Proceedings of the VLDB Endowment*, 13(12):3152–3165, 2020.

[70] Shulin Zeng, Zhenhua Zhu, Jun Liu, Haoyu Zhang, Guohao Dai, Zixuan Zhou, Shuangchen Li, Xuefei Ning, Yuan Xie, Huazhong Yang, et al. Df-gas: a distributed fpga-as-a-service architecture towards billion-scale graph-based approximate nearest neighbor search. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 283–296, 2023.

[71] Zili Zhang, Fangyue Liu, Gang Huang, Xuanzhe Liu, and Xin Jin. Fast vector query processing for large datasets beyond {GPU} memory with reordered pipelining. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 23–40, 2024.

SwCC: Software-Programmable and Per-Packet Congestion Control in RDMA Engine

Hongjing Huang[†], Jie Zhang[†], Xuzheng Chen, Ziyu Song, Jiajun Qin, Zeke Wang[*]

Department of Computer Science, Zhejiang University, China

Abstract

Many data centers adopt Remote Direct Memory Access (RDMA) to allow data center applications to achieve low latency and high throughput, while keeping minimal CPU overhead. The upper-layer applications keep evolving rapidly, and thus need congestion control algorithms (CCAs) that exist in the NIC hardware also to react correctly and timely, especially for a burstier ML workload. Even worse, the data center network will increase the line rate to 400 Gbps, even 800 Gbps soon. Therefore, how to reduce control loop delay for various CCAs becomes crucial to the performance of various applications. However, RDMA's hardwired CCA is not able to satisfy such a requirement.

To this end, we design and implement SwCC, an RDMA engine with on-NIC RISC-V cores that allows software-programmable and per-packet congestion control. To avoid the performance degradation caused by introducing the programmable RISC-V cores, SwCC carefully designs the 1) RISC-V core memory subsystem, 2) engine architecture, and 3) interaction between the RISC-V core and other NIC resources. Besides, SwCC provides a set of rich software APIs, allowing developers to deploy new CCAs with minimum engineering efforts. We prototype SwCC using the Xilinx U280 FPGA. Experimental results demonstrate that SwCC achieves performance comparable to current commercial RDMA NICs (Mellanox ConnextX-5). Both SwCC and ConnectX-5 reach 3.1 µs control loop RTT and need 512B packet size to reach line-rate traffic (100 Gbps). In terms of flexibility, SwCC allows to use the C language to implement nearly all kinds of existing CCAs, e.g., rate-based CCAs, window-based CCAs, and credit-based CCAs. The potential ASIC design of SwCC can easily scale to higher network bandwidth.

1 Introduction

Modern data centers urge their networks with low latency and high bandwidth utilization while keeping low CPU utilization. In recent years, many mainstream data centers have adopted RDMA to reduce CPU overhead and network latency. Among all RDMA technologies, RDMA over Converged Ethernet (RoCE [33]) is the most widely deployed in modern data centers such as Microsoft Azure [8, 24, 28], Alibaba Cloud [20], and Google Cloud [40]. RoCE relies on Priority Flow Control (PFC) [32] to ensure the network lossless [20, 87, 88]. However, PFC introduces issues such as head-of-line blocking, congestion spreading, and PFC deadlocks [25, 28, 29, 56, 73]. To essentially prevent the activation of PFC, developers propose various CCAs to optimize network performance by effectively managing traffic and reducing congestion, thereby inherently preventing the activation of PFC. According to the location of RDMA congestion control (CC) logic, we mainly identify four types of CC solutions.

ASIC-based NIC CC solutions, such as DCQCN [88], IRN [56], implement the CC logic on the ASIC-based NIC. These solutions adjust network transmission within tens of nanoseconds after congestion notifications reach the NIC, significantly reducing control loop latency. However, CCAs are tightly coupled with the NICs and can not be optimized or upgraded after manufacturing. When a better CCA is introduced, it takes years to deploy it in real commodity NICs. Therefore, these solutions cannot keep up with the rapid iterations of data center CCAs. For example, at least 11 CCAs [1, 2, 6, 21, 22, 44, 54, 76, 81, 82, 86] have been proposed in 3 years, while only 7 Mellanox NICs [47–53] are released in recent 10 years.

CPU-based CC solutions. To facilitate the rapid deployment of new CCAs, people place the CC logic in CPUs, leveraging software flexibility and programmability to enable fast deployment of various CCAs. CPU-based CC solutions, such as Soft-RoCE [62], implement the CC logic on the CPU to enable the flexible deployment of various CCAs, allowing dynamic adjustments to transmission bandwidth based on diverse congestion signals. However, compared with deploying CC logic in the NIC, these solutions usually have a high control loop delay due to the long latency from PCIe and layered software (tens of microseconds higher). Considering RDMA's micro-second latency, such a high delay could

[†]Contributes equally
[*]Corresponding author

greatly degrade the network performance when congestion occurs. In our experiment that runs two flows with DCQCN [88] on a dumbbell topology, we observe that a high control loop delay would significantly increase the time needed to drain the switch queue. Compared with NIC-based CC solutions, running CCAs in the host CPU can increase the switch queue draining time by 7.8× (Subsection 3.2).

Naive FPGA-based SmartNIC CC Solutions. To enable programmability while keeping low control loop delay, developers place the CC logic in SmartNICs to avoid PCIe latency and complicated host software layers. Tonic [4] and NanoTransport [5] place the CC logic on FPGA-based SmartNICs. These approaches offload transport logic in the SmartNIC and allow users to update the NIC transport logic using hardware description language (HDL). However, FPGA programming requires much more engineering effort and a much longer development cycle, making it hard to keep pace with the rapid iteration of CCAs.

SoC-based SmartNIC CC Solutions. To address the programmability issue, NVIDIA BlueField-3 [60] offers Programmable Congestion Control (PCC) [63] to run CCAs in a RISC-V processor named datapath accelerator (DPA), which allows users to deploy the CCAs using software programming. Though this provides high programmability, it is hard to ensure performance with wimpy DPA, because the CC controller is triggered in a long interval and can not be triggered each time when receiving a packet. Besides, PCC only allows developers to adjust per-flow rates and cannot support credit-based or window-based CCAs, and PCC only supports a small subset of CCAs (as shown in Figure 2). Therefore, PCC results in low flexibility.

In summary, none of the existing solutions can provide both high flexibility and high programmability while maintaining a low control loop delay. To this end, we design and implement SwCC, an RDMA engine that allows software-programmable and per-packet congestion control. SwCC integrates an on-NIC RISC-V processor to allow flexible and fast deployment of CCAs while keeping software programming. The main **challenge** is that naively integrating a programmable RISC-V processor into the RDMA engine will cause severe performance degradation (as observed in NVIDIA BlueField-3 PCC [63]). To this end, SwCC carefully design the 1) memory subsystem by exploiting the data access pattern of different CCAs, 2) engine architecture to allow the core to run at a high frequency, and 3) interaction between the RISC-V core and other NIC resources to reduce required CPU cycles when running CCAs in the RISC-V core.

We prototype SwCC on an Xilinx U280 FPGA. Experimental results demonstrate that SwCC achieves comparable performance to current commercial RDMA NICs (Mellanox ConnextX-5): 1) 3.1 µs/3.4 µs control loop RTT, and 2) 512-bytes/512-bytes packet size to achieve line-rate traffic (100 Gbps). SwCC supports rate-based, window-based, and credit-based mainstream CCAs to ensure flexibility. We implement representative CCAs such as TIMELY, DCQCN, and HPCC using only 102, 140, and 148 lines of code, respectively. We believe that the potential ASIC design of SwCC can easily scale to a higher network bandwidth.

2 Background

An end-to-end CC algorithm involves the collaboration between the sender and the receiver. The receiver is usually responsible for notifying the sender of the congestion with meta information such as credits [19, 57] and congestion notification packets (CNP) [86, 88]. The CC controller on the sender side reacts to the congestion and reallocates network resources such as sending rate, and congestion window, depending on the specific algorithm. Different CCAs 1) use different CC signals, 2) trigger the CC controller at different times, and 3) reallocate network resources with different strategies when the CC controller is triggered.

2.1 What CC Signals to Use

CC controller on the sender side relies on CC signals to be aware of the network congestion. Different CCAs use very different signals, which can be categorized into four classes.

ECN: CCAs such as DCQCN [88], TCD [86], and ACC [81] use explicit congestion notification (ECN). The receiver would send congestion notification packets (CNPs) to notify the senders of the network congestion.

RTT: CCAs, such as TIMELY [55], Swift [40], BBR [11], Copa [7], and Bolt [6], allows the senders to proactively measure the link RTT to be aware of the network congestion. The RTT value can indicate the degree of the queuing along the link, which is caused by network congestion.

INT: CCAs, such as HPCC [42], PowerTCP [1], and Poseidon [76], utilize INT (In-band Network Telemetry) to obtain precise link load. When a sender sends a packet to a receiver, the switches along the path insert metadata indicating the current loads of the packet's egress port. The receiver copies these metadata to the ACK packet and sends the ACK to the sender, which then adjusts its transmission accordingly.

Token: CCAs, such as pHost [19], NDP [27], Homa [57], and ExpressPass [14], use credits to avoid network congestion. The available sender credit of a flow is the total tokens received from the receiver minus the transmitted bytes. Usually, each transmitted packet piggybacks the credit in the ACK.

2.2 When to Trigger the CC Controller

CCAs use events to trigger the CC controller. Different CCAs generate events at different times. The events can be mainly categorized into sending packet events, receiving packet events, and timer/counter events. The sending/receiving packet events are generated when sending/receiving packets. Note that the above events may not generated for all kinds of packets. For example, some CCAs [40, 42, 55] only need to generate receiving packet events for ACK packets. Timer/-counter events are generated periodically or when the counter

Table 1: Comparison of CC Solution in Different Locations

Solution	Control Loop Delay	Flexibility	Programmability
CPU-based CC [62]	23 μs	High	High
ASIC-baesd NIC CC [33]	3.1 μs	Low	Low
Naive FPGA-based SmartNIC CC [4]	3.3 μs	High	Low
SoC-based SmartNIC CC [63]	3.5 μs	Low	High
SwCC	3.3 μs	High	High

reaches a threshold. One CCA may adopt one or multiple kinds of events to trigger the CC controller. For example, credit-based CCAs usually generate sending packet events to trigger the CC controller to decrease the credit and generate receiving packet events to trigger the CC controller to increase the credit.

2.3 How to Adjust Data Transmission

CC controller limits the data transmission in three main ways:
Window: Previous works [1, 6, 40, 42, 76, 82] adopt a per-flow congestion window to allow for 1) tracking each inflight packet/segment, and 2) reacting to network congestion in a fine-grained manner. Window-based CC is quite complicated, especially in the presence of selective acknowledgments.
Rate: Instead of tracking each packet/segment, previous works [54, 55, 75, 81, 82, 88] adopt a more coarse-grained approach to limit packet sending rate. They only adjust the sending rate of each flow. For example, TIMELY [55] measures the flow RTT and adjusts the sending rate according to the RTT gradient.
Credit: Previous works [14, 19, 27, 57] use credits as permissions to send data. The credit of a flow is the total number of received tokens minus the number of transmitted bytes and the credit calculation consists of simple addition/subtraction.

3 Motivation

An ideal RDMA NIC should offer 1) low control loop delay to let CCAs react to congestion timely; 2) high flexibility to enable various kinds of CCAs; and 3) high programmability to allow developers to program easily without knowing much hardware details. However, the existing solutions fail to meet all the requirements as shown in Table 1. In the following, we present the detailed design and the corresponding limitations of each kind of solution.

3.1 ASIC-based NIC Solution

RDMA NICs such as Mellanox Connect-X series [50] are widely deployed in nowadays datacenters. They offload the congestion control logic to the NIC ASIC area to save host CPU usage. Running CCAs in the NIC greatly brings low control loop delay as it avoids PCIe traversals for control signals. However, ASIC-based solutions are far behind the rapid evolution of the CCAs, because one special RDMA NIC usually only supports one or a few CCAs with configurable settings. For example, Mellanox CX series NICs only support DCQCN. However, CCAs are evolving rapidly. From 2021 to 2023 alone, 11 new CC algorithms were presented [1, 2, 6, 21, 22, 44, 54, 76, 81, 82, 86] to improve the CC performance for different scenarios. At the same time, commercial NIC hardware requires years of development. Mellanox releases only seven CX series NICs [47–53] since 2012. It's impossible for commercial NICs to always keep pace with the evolving CCAs.
Limitation: ASIC-based NIC solutions provide low control loop delay but have low flexibility and low programmability. Even worse, ASIC-based NIC usually only supports adjusting CCA parameters through firmware updates, and upgrading to a new CCA is nearly impossible for currently off-the-shelf ASIC NICs.

3.2 CPU-based CC Solution

To enable fast deployment of new CCAs, Soft-RoCE [62] runs CCA logic in host CPU software. It implements the RDMA transport on top of the Linux network stack. Users can use software programming languages to implement their CCAs, providing high flexibility and programmability. However, running CCA logic in the software significantly increases the control loop delay. As Table 1 shows, Soft-RoCE needs nearly an order of magnitude higher control loop delay than running CCAs in the hardware NIC. Compared with running CCAs in ASIC-based NIC, Soft-RoCE introduces severe overheads: PCIe traversal, software stack, and kernel switching.

The high control loop delay is harmful to the performance of CCAs. To illustrate the potential impact of control loop delay, we conduct an experiment in our testbed. We connect two sender servers and one receiver server to a P4 switch. Each server is equipped with an FPGA NIC running DCQCN [61]. We first let one sender establish a stable flow to the receiver and then let the other sender establish a new flow to the receiver. The control loop delay is measured from the generation of the CNP packet to when the sender receives it. We manually delay the processing of the CNP packet to obtain different control loop delays. Figure 1 shows the time to drain the switch queue when congestion occurs. We observe that a high control loop delay would significantly increase the time needed to drain the switch queue. Compared to NIC-based CC solutions, running CCAs in the host CPU can increase the draining time of the switch queue by 7.8×.
Limitation: CPU-based CC solution provides high flexibility and high programmability. However, the overheads from PCIe traversals and software context switching significantly increase control loop delay, thus degrading CC performance.

3.3 Naive FPGA SmartNIC-based CC Solution

To keep low control loop delay while enabling multiple kinds of CCAs, researchers offload transport protocols on FPGA SmartNICs. For example, Tonic [4] proposes a Verilog-programmable transport layer in hardware. NanoTransport [5] proposes a programmable hardware transport layer using the

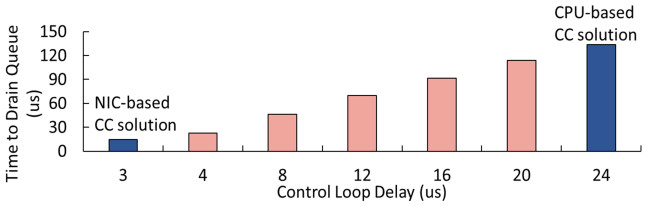


Figure 1: Effect of control loop latency on queue draining time.

P4 language. Thus flexibility is guaranteed because such a programmable transport allows developers to implement CCAs according to their needs. Low control loop delay is also guaranteed because the transport runs in the NIC and the CC control signals do not need to traverse the PCIe interconnect or kernel stack.

Limitation: However, offloading transport to FPGA exposes one severe limitation: hardware programming offers low programmability. Developers have to write hardware description languages (HDLs) such as Verilog and Chisel. Implementing a CCA using HDL requires much more engineering effort. Developers have to know hardware details well and fix timing problems. Besides, debugging is very difficult when using FPGAs to implement CCAs.

3.4 SoC SmartNIC-based CC solution

NVIDIA BlueField-3 SmartNIC allows the offload of CCAs to the RISC-V processor of the SoC SmartNIC, called programmable congestion control (PCC [63]). PCC allows developers to write a C-style program to run on the NIC RISC-V processor. NIC periodically generates events and notifies the user's CC program to adjust the data transmission rate. Conceptually, PCC keeps a low control loop delay while providing high flexibility and programmability.

However, the cache and memory system of the datapath accelerator (DPA, i.e., RISC-V cores) in BlueField-3 are inadequate to support running PCC efficiently. Because DPA has poor cache/memory performance and weak single-thread performance, DPA is suitable for simple workloads with high parallelism (using its 256 concurrent threads) [13]. However, CCAs need to fetch many packet-related data (i.e., fields in the packet header) and QP-related data (e.g., per-QP credit) from memory, so its poor cache (L1: 10 ns)/memory (300 ns) incurs high overhead to process each packet. For example, at a 100 Gbps line rate, a 1 KB packet arrives every 89 ns, well below the memory access latency (300 ns).

This cannot be easily solved by using DPA's multiple threads. CC is not a high-parallelism workload, as the packet processing of the same QP must be mapped to the same thread to avoid inter-thread communication/synchronization overhead. As such, letting DPA run CC for a small number of QPs cannot exploit the high parallelism of DPA and easily becomes a system bottleneck, especially for the CCAs triggered per packet. This is also why the **current PCC only supports rate-based CCAs**, because not every packet would trigger the CC in rate-based CCAs.

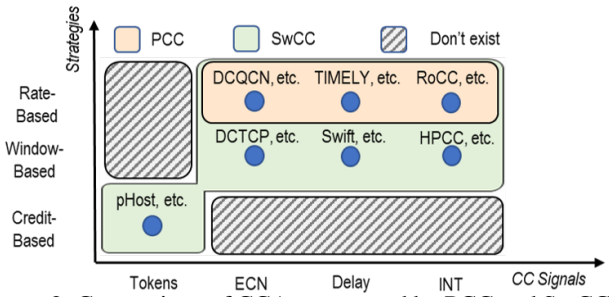


Figure 2: Comparison of CCAs supported by PCC and SwCC.

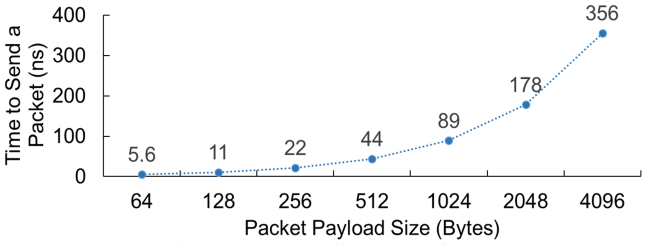


Figure 3: To achieve a network line rate of 100 Gbps, the NIC has to send a data packet within a specific duration depending on the packet size.

4 Design and Implementation

To address these limitations, we present SwCC, a NIC RDMA engine that allows flexible implementation of various CCAs through software programming without sacrificing performance. SwCC has three design goals:

Low Control Loop Delay and Short CC Controller Triggering Interval. As described in Section 3.2, control loop delay directly influences the efficiency of CCAs, SwCC aims to provide low control loop delay. Besides, SwCC must ensure a short CC controller triggering interval to support per-packet CCAs at a high network bandwidth. To be more specific, we have to allow triggering CC controller upon sending/receiving a packet. As shown in Figure 3, under the 100 Gbps line rate, the packet arrival interval is only 89 nanoseconds for 1 KB packets. Therefore, the frequency of the CC controller should be as high as possible, and the required cycles for processing each packet should be as low as possible.

High Flexibility. To support various kinds of CCAs, SwCC must allow developers to use different kinds of CC signals (§Section 4.3.1) and use different kinds of data transmission strategies (§Section 4.2.2). Besides, SwCC should allow triggering the CC controller flexibly (§Section 4.3.2).

High Programmability. To keep pace with the fast evolution of CCAs, SwCC must provide an easy-to-program interface to allow developers to focus only on the high-level CC algorithms, rather than on low-level hardware details.

To this end, we design and implement SwCC, an RDMA engine that allows software-programmable and per-packet congestion control. SwCC integrates an on-NIC RISC-V processor to allow flexible and fast deployment of CCAs while keeping software programming, thus providing flex-

ibility. However, this introduces significant processing overhead compared to hardwired solutions. Our key idea is to carefully design the interaction between the RDMA engine and the RISC-V core, so as to enable the user CC program running in the RISC-V processor to access and control the NIC resources at a very low cost. To do so, we rely on careful co-design from both hardware and software perspectives. As such, we have ensured 1) **low control loop delay** because the CCAs are running in the NIC RISC-V core; 2) **short CC controller triggering interval** due to the carefully designed hardware/software interaction mechanism; 3) **high flexibility** because we expose enough hardware resources to the RISC-V processor; and 4) **high programmability** because users can program the RISC-V core with a high-level software programming language.

4.1 SwCC Design Overview

Figure 4 provides SwCC system overview. The hardware part mainly consists of 1) RX path logic; 2) TX path logic; 3) "WQ Handler" module that processes WQEs, sending data packets to the TX CC and packets without data to the "Header Fabricator"; 4) "DMA" that performs address translation and reads/writes data to the host memory; 5) QP-aware memory subsystem that maintains the per-queue-pair QP context.

Along the RX path, SwCC consists of 1) "RX Parser" that parses incoming packets; 2) "Packet Retransmission" that handles go-back-N retransmission; 3) "Packet Filter" that generates events to trigger the RX CC core according to the user configuration; 4) "RX CC" that runs CCA logic related to dealing with RX events.

Along the TX path, SwCC consists of 1) "TX CC" that runs CCA logic related to whether to allow transmitting a packet; 2) "Header Fabricator" and "Packet Fabricator " that fabricate the header and the complete packet.

SwCC software consists of a rich set of APIs allowing easy programming. These APIs allow users to define 1) when to trigger the CC controller flexibly, 2) what extra header fields should be carried in each packet as needed, and 3) how to let the CC core interact with hardware efficiently. In the following, we present the detailed design of SwCC, organized in a manner that shows how it guarantees our design goals.

4.2 How to Keep Low Control Loop Delay and Short CC Controller Triggering Interval.

To keep a low control loop delay, we put a programmable CC controller (RISC-V core) that runs CCAs in the NIC engine. This greatly reduces CC controller response time to network congestion as it avoids the long PCIe wire latency, which could contribute up to 90% wire-to-wire response time for flows with small packets [59].

However, naively integrating a RISC-V processor into the NIC leads to low single-core performance due to the power/-form factor limitation of the NIC, as evidenced by the case of NVIDIA PCC. To keep a short CC controller triggering interval, we 1) strive to increase the CC controller hardware frequency as much as possible; 2) strive to reduce the required number of CPU cycles when the CC controller processes each event.

4.2.1 TX/RX Separated Multi-Core

Integrating more cores in the CC controller can linearly increase the CC processing capability, thus allowing a short CC triggering interval. Instead of naively adopting a multi-core design in the CC controller, we propose a TX/RX separated multi-core design, which runs the TX and RX logic of CCAs in different CC controller RISC-V cores.

Such a design is guided by our observation that **most CCAs can be clearly partitioned into a TX function and an RX function**, in both the programming perspective and logic perspective. In the RX function, when the CC controller is triggered by an event, the CC controller adjusts the context of the current queue pair, such as the congestion window and credits. In the TX function, the CC controller usually decides whether to send a packet or not, according to the context of the current queue pair.

Based on this observation, the proposed TX/RX separated multi-core lets each core only deal with either TX or RX function, so as to reduce processing latency. In case of enough hardware resources for $2N$ cores in the NIC, we let half of them process the TX function and the other half process the RX function. Each queue pair corresponds to a specific TX CC core and a specific RX CC core, according to the result of the queue pair number (QPN) mod N. This avoids consistency issues as the same queue pair would not be processed by different RX CC cores or TX CC cores.

The main benefit of TX/RX separated multi-core is that a single core would only deal with a subset of CC logic and thus only need to access a subset of the NIC resources. For example, the RX CC core only needs to be connected to the RX port, and the TX CC core only needs to be connected to the TX port. Compared to a fully connected data path, TX/RX separated multi-core can greatly reduce the required hardware resources to construct the data path. As such, the design can meet the timing constraints more easily. In our FPGA prototype, TX/RX separated multi-core can increase the design frequency by up to 16% (§Section 5.3). We believe that the conclusion still holds when applied to an ASIC-based design of SwCC.

4.2.2 QP-aware Memory Subsystem

A traditional memory subsystem relies on a large cache with a sophisticated pre-fetching mechanism to provide upper-layer applications with low-latency memory accesses. However, due to form factor and power constraints, the cache size in the NIC is usually limited, and the pre-fetching mechanism is much wimpier compared with traditional host server CPUs. And a NIC cache usually has much higher latency than the server CPU cache (an order of magnitude higher) as observed in NVIDIA BF3 [13].

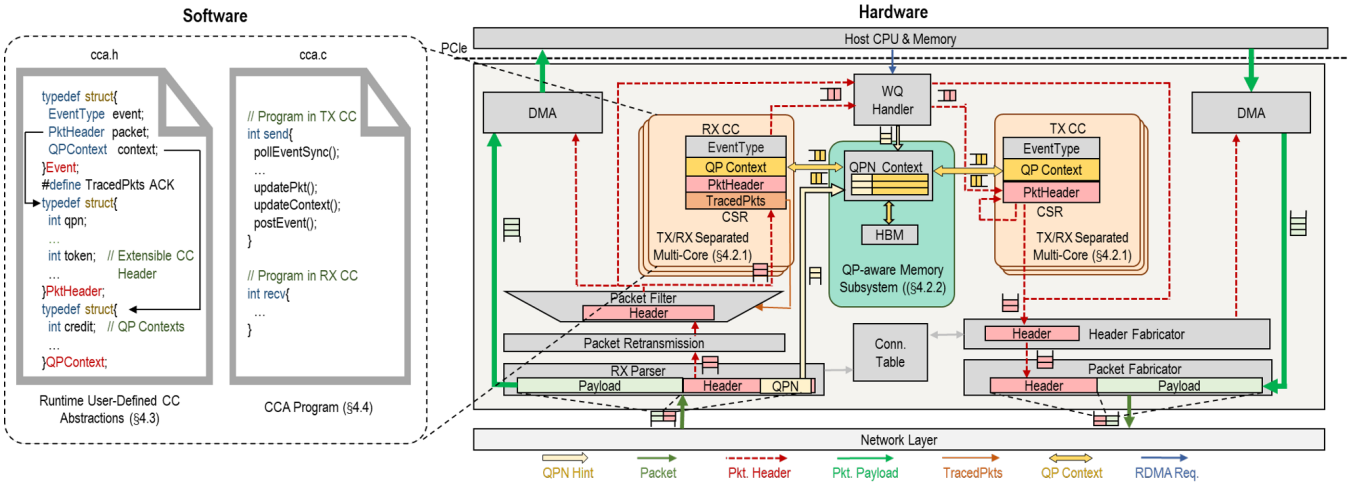


Figure 4: Overall architecture of SwCC.

Instead of trying to construct a powerful and general memory subsystem, we exploit the underlying data access pattern in the NIC CC core and construct our memory subsystem accordingly, based on the following observation. When receiving a packet, the packet is first processed by the RX-related logic, such as header parsing and retransmission logic. Then the CC controller is triggered to run CC logic. When running CC logic, the RX core needs to access QP-related context, such as per-QP credits, and packet-related information, such as returned tokens.

Inspired by this, we propose QP (Queue pair)-aware memory subsystem to forward the QP contexts and user-specified packet header to the CC core's control & status registers (CSR) before the CC core is triggered. What's more, before the packet is processed by the RX-related logic, we use the QPN as a hint to prefetch the QP context from off-chip DRAMs to the on-chip memory region. As such, it overlaps the DRAM access latency and RX-related logic processing, minimizing the time of forwarding QP context to the CC core's CSRs. To do so, we introduce four hardware-software co-designed mechanisms (described below) and revamp the data access process based on them.

QP Context Table. In SwCC, we do not design a traditional cache that organizes data in cache-line granularity, uses memory address to index each cache slot, and uses host DRAM to back on-chip SRAM. Instead, we use both on-chip SRAM and off-chip DRAM to store QP contexts. The on-chip SRAM is organized as a 4-way set-associative cache, with each cache line storing all the contexts of a QP. Each slot is 128 bytes[1] and the total size of the QP context table is 128 KB, which can store QP contexts of up to 1K QP. The swap-in/out process follows a simple LRU algorithm. Unlike traditional RNICs, which use host DRAM to back on-chip SRAM, SwCC uses on-NIC DRAM, which offers much lower access latency (1200 ns vs. 100 ns) and incurs significantly lower swapping overhead.

CSR-based Fast Path. Although the data organization differs from that of traditional caches, the QP context table still has a relatively high latency given the power/form factor constraints. Instead of letting CC cores access the QP context table at runtime, we let hardware write the corresponding QP contexts into the core's control & status registers (CSRs) in advance. The standard RISC-V ISA sets aside a 12-bit encoding space for up to 4,096 CSRs [66]. In each SwCC CC core, we leverage 64 CSRs[2] that have been reserved in the RISC-V ISA sets. Before the CC core is triggered to execute the RX or TX function, the NIC hardware would write the incoming event type, and QP contexts into the reserved CSRs using a user-configurable manner in advance. For the RX CC core, the header of the received packet that triggers the RX function would also be written into the CSR registers. During the execution of the TX/RX function, the CC cores directly operate on the CSR registers instead of the QP context table. Reading data from CSRs to general-purpose registers (GPRs) only takes one cycle since implementing 64 one-cycle CSRs (256 bytes in total) is easy even under power/form factor constraints. As such, the CC core execution time is greatly reduced as high NIC cache access latency is removed.

Prefetch with QPN Hint. When an event is scheduled to run on an RX CC core, the NIC hardware has to write the QP context into the CC core's CSRs. However, it's likely that the QP's slot is not in the QP context table, as the on-chip table size is limited and the number of concurrent QPs could be numerous. In this case, the QP context table has to fetch the QP contexts from the backed off-chip DRAM and the CC core has to be stalled for dozens of cycles due to the long off-chip DRAM access time. As such, we use QPN as a hint to prefetch the QP contexts from off-chip DRAM in advance.

In the RX path, when a packet enters "RX Parser", the packet QPN would be extracted and forwarded to the QP

[1]We choose 128 bytes because the per-QP contexts of most CCAs can fit in this size.

[2]The 64 CSRs include 32 CSRs for writing QP context, 16 CSRs for writing packet header, and 16 CSRs for writing event type and other configurations.

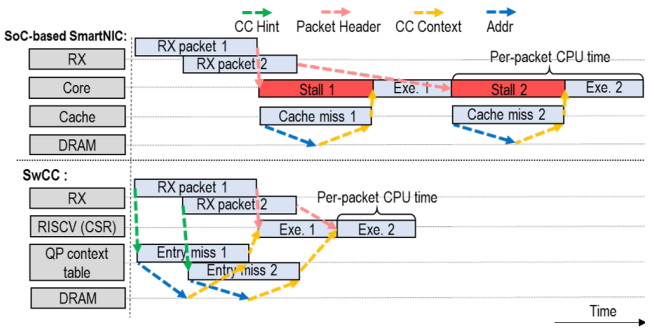


Figure 5: Comparison of Soc-based SmartNIC and SwCC when processing packets.

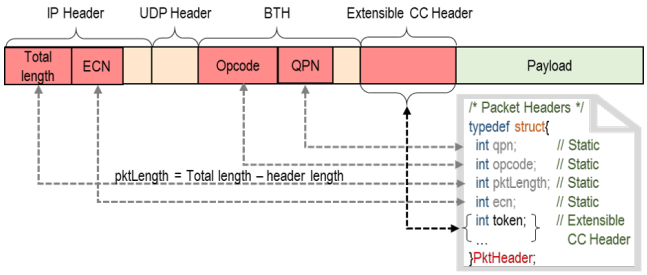


Figure 6: The packet format of SwCC.

context table to prefetch the QP contexts from the backed DRAM if the QPN's contexts are not in the table. Before the packet's corresponding event runs on the RX CC core, it has to go through "RX Parser", update the packet sequence number (PSN) in the "Connection table", and go through "Packet Filter". Processing the above logic takes around 60 nanoseconds. Considering that a DRAM access time is very close to this value, forwarding the QPN in advance allows the QP context table to hide the stall penalty caused by long-latency DRAM access.

In the TX path, when a sending packet request from the user/RX core enters the "WQ Handler", the request QPN would be forwarded to the QP context table for prefetching. This also helps to hide QP context table miss penalty as the "WQ Handler" has to slice large messages into small packets and the processing time could be up to dozens of nanoseconds.

Consistency Guarantee. Although different QPs are mapped to different CC cores, it is likely that a TX CC core and an RX CC core concurrently access the QP context of a particular flow. As such, the QP context table must provide a consistency guarantee. We implement locking purely in hardware, without requiring atomic instructions in the software. The core is triggered by hardware only when the QP context entry to be processed is not owned by other cores. Once triggered, the QP context entry is owned by this core, and the hardware manages the entry ownership. Each entry has a lock bit to indicate whether it is owned by a core. When triggering a core with an event, the hardware checks the lock bit; if occupied, it tries other events; otherwise, it triggers the core and sets the lock bit. When the core finishes execution, the hardware clears the lock bit. This mechanism works for two reasons. First, each event follows a run-to-completion model, so a core does not require ownership after processing. Second, the core processes events one by one, with each event having only one corresponding QP context entry, ensuring no complicated lock dependencies.

Summary. With the above four hardware-software co-designed mechanisms, SwCC is able to reduce as many as possible CC core stalls due to the long off-chip memory access time. Figure 5 illustrates how the packet events are processed when there is a miss in the on-chip cache/QP context table. In SoC-based SmartNIC, the RX packet has to be pre-processed by the RX hardware pipeline before triggering CC core execution. The cache miss would incur the stall of the core. In SwCC, we prefetch with a QPN hint to hide on-chip QP context table misses within the RX packet pre-processing. As such, the real RISC-V core cycles used for each packet event are minimized, thus a short CC controller triggering interval is guaranteed.

4.3 How to Keep High Flexibility

SwCC should provide developers with the ability to implement various kinds of CCAs, i.e., developers can decide 1) what signals to use, 2) when to trigger the CC controller, and 3) how to adjust data transmission in the CC controller.

As described in § 4.2.1, we place a multi-core RISC-V processor as the CC controller to allow developers to implement their custom logic on how to adjust data transmission. In the following, we present how developers can decide what signals to use and when to trigger the CC controller in SwCC.

4.3.1 Using Different CC Signals in SwCC

SwCC allows developers to leverage different CC signals to provide as much flexibility as possible. As discussed in §Section 2, CC signals can be mainly categorized into 1) ECN, 2) delay, 3) token, and 4) INT signals.

Extensible CC Header. To support the above CC signals, we propose the extensible CC header, a customized header sitting after the RoCEv2 BTH header. To use the extensible CC header, developers can add their own fields in the definition of a structure `PktHeader`. The first four fields (QPN, RDMA opcode, pktLength, and ECN) in the `PktHeader` are common for all kinds of CCAs, and developers can add their own fields, such as tokens, after the fixed four fields. Each packet in SwCC would carry the user-defined extensible CC header. The size of the extensible CC ranges from 0 to 64 bytes, as 64 bytes are enough for all mainstream CCAs (for example, HPCC [42] has 42B). In the RX path, when receiving a packet, the QPN and opcode in the BTH header, the packet length and ECN signal in the IP header would be extracted to the first four bytes of the structure `PktHeader`. The user-defined extensible CC header would also be placed in the following position of the structure `PktHeader`. The RX CC core can read the `PktHeader` structure to get all the required information. In the TX path, the TX CC core can write data into the structure `PktHeader`, and the hardware would

be responsible for padding the data into the packet header. The position of the `PktHeader` in the packet header is shown in Figure 6.

Compatibility. Different CCAs have different deployment requirements. If the chosen CCA does not utilize the extensible CC header and thus does not require the remote side to provide custom metadata/signals, the remote side can be an existing commercial NIC. Otherwise, the remote side must be a SwCC-enhanced NIC. For non-INT CCAs, the switch simply treats the extensible CC header as part of the RDMA packet payload, and there is no switch deployment requirement. For INT-based CCAs, we currently need to use programmable switches to add INT metadata in the extensible header, as existing switches can only add INT metadata at the end of the packet.

The extensible CC header allows developers to implement ECN-based, token-based, and INT-based CCAs. To support delay-based CCAs, SwCC implements a hardware timer and exposes the timer as a CSR to all CC cores. CCA logic in the CC core can read this CSR to get the hardware time and use it to implement their delay-based CC logic. As such, SwCC can support all four kinds of CCAs and SwCC allows developers to decide what CC signals to use flexibly.

4.3.2 Selective Triggering

To allow developers to flexibly trigger the CC controller, we design a "Packet Filter" module that sits between the "RX Parser" and RX CC cores. Instead of letting each incoming packet generate an event and trigger the CC controller, the "Packet Filter" only generates events for the user-specified packets.When a packet arrives, the "Packet Filter" checks the opcode field in the packet header and determines whether to generate an event to trigger the RISC-V core. As such, developers can decide whether to trigger the CC controller at runtime.

The "Packet Filter" recognizes the RDMA opcode in the BTH header of the incoming packet. The opcode consists of 8 bits, where opcode[7:5] indicates the RDMA transport service type (e.g., Reliable Connection) and opcode[4:0] indicates the packet type (e.g., Send, Recv, Read, Write, etc.). In addition to the opcodes defined by the RoCEv2 standard, developers can adopt unused 0x18-0x1F opcodes to define their own packet types [34]. When implementing a CCA, developers must specify which RDMA opcodes will trigger an event by setting a 32-bit one-hot value. During system initialization (i.e., after the CCAs are programmed to the RISC-V core), the RISC-V core updates one of its CSRs using the one-hot value. The value of this CSR is connected as an input of the "Packet Filter" module, which would trigger the events according to this input. By blocking unnecessary packet events, selective triggering allows the CC controller to focus on processing user-specified events. As such, SwCC achieves its design goal of letting developers decide when to trigger the CC controller flexibly.

4.4 How to Keep High Programmability

SwCC provides a rich set of programming APIs (Table 2) to help developers to deploy their CCAs. We take a simple credit-based CCA as an example and show how to use these APIs to implement the CCA. The CCA has two functional parts: sender-side TX function and receiver-side RX function.

On the sender side, the TX function checks if the remaining credit is larger than the length of the to-be-sent packet. If the credit is enough, the TX function would notify the NIC hardware to construct and send this packet out. Otherwise, the request would be deferred.

On the receiver side, the RX function increments the credit if it has received an ACK packet. If the received packet is a data packet, the RX function would generate a send packet event to notify the NIC hardware to send an ACK packet to the remote sender. The CCA defines four bytes in the extensible CC header to store the returned credits in the ACK packet.

```
/* Selective processing */
#define TracedPkts (ACK | WRITE_REQ | READ_RESP)
/* Extensible packet headers */
typedef struct{
  int qpn;          /*Static*/
  int opcode;       /*Static*/
  int pktLength;    /*Static*/
  int ecn;          /*Static*/
  int returned_credit;
}PktHeader;
/* Extensible QP context */
typedef struct{
  int total_credit;
}QPContext;
/* Program running in the TX CC cores*/
int send(){
  int credit;
  Event e;
  while(true){
    /* Poll for new packet arrivals and read events from CSR */
    pollEventSync(&e);
    /* Update the credit when sending a packet */
    if(e.context.total_credit > e.packet.pktLength){
      credit = e.context.total_credit - e.packet.pktLength;
      updateContext(e.context.total_credit, credit);
      e.eventType = Send;
    }else{
      e.eventType = LoopBack;
    }
    postEvent(&e);  /* Write the event to CSR */
  }
}
/* Program running in the RX CC cores*/
int recv(){
  Event e;
  int credit;
  while(true){
    pollEventSync(&e);
    /* Update the credit when receiving an ACK */
    if(e.packet.opcode == ACK){
      credit = e.context.total_credit + e.packet.returned_credit;
      updateContext(e.context.total_credit, credit);
      e.eventType = Done;
    /* Send a ACK when receiving a WRITE_REQ or READ_RESP */
    }else{
      updatePkt(e.packet.returned_credit, e.packet.pktLength);
      updatePkt(e.packet.opcode, ACK);
      e.eventType = GenTxEvent;
    }
    postEvent(&e);
  }
}
```

Listing 1: An example of programming with SwCC.

Next, we show how to implement the CCA using the provided programming interfaces. Before writing CC controller logic, the developer has to complete three definitions. First, the developer has to define the variable `TracedPkts`

Function	Parameters	Description
pollEventSync (`&Event`)	The `Event` containing the `PKTHeader`, `QPContext` and `EventType`	Busy polling for the arrival of new packet, then read the `PKTHeader` and `QPContext` from the CSR into the `Event`.
updatePkt (`packet.item`, `item`)	An item in the `PKTHeader` (`packet.item`), A variable (`item`) to update the item in the `PKTHeader`	Update the `packet.item` in the `PKTHeader` using `item`
updateContext (`context.item`, `item`)	An item in the `QPContext` (`context.item`), A variable (`item`) to update the item in the `QPContext`	Update the `context.item` in the `QPContext` using `item`
postEvent (`&Event`)	The `Event` containing the `PKTHeader`, `QPContext` and `EventType`	Write the `Event` into the CSR, then notify the hardware engine that packet processing is complete

Table 2: SwCC programming APIs for implementing CCAs.

(Line 2), and this value can tell the "Packet Filter" hardware to only generate RX events when receiving ACK-/RDMA_WRITE_REQ/RDMA_READ_RESP packets. Second, the developer has to define the `returned_credits` field as an extensible CC header in the structure `PktHeader` (Line 9), and this field is used to store the returned credits in the ACK packets. Third, the developer has to define the field `total_credit` field in the QP context table (Line 13), and this field is used to store the remaining credits of each QP. After that, the developer can write the RX function and TX function.

In the TX path, the TX CC core first polls send packet requests in the send request FIFO (Line 21). After polling a request, the TX CC core checks whether the remaining credit is larger than the requested packet size (Line 23). If so, the TX CC core decreases the total credit accordingly (Lines 24-25) and marks the request with `Send`, indicating that this send packet request is granted (Line 26). Otherwise, the TX CC core marks the request with `LOOPBACK` which indicates that the request should be pushed back into the send request FIFO (Lines 28). At last, the TX CC core posts the event and notifies the NIC hardware (Line 30).

In the RX path, the RX CC core polls the event FIFO (Line 38). After polling an event, the RX CC core checks whether the received packet is an ACK packet (Line 40). If so, the RX CC core increases the total credits by the value of the `returned_credit` field in the extensible CC header (Lines 41-42), and then marks the event with `Done` (Line 43), indicating that NIC hardware does not need to do extra work. Otherwise, we should generate an ACK for the received packet (`RDMA WRITE_REQ` or `RDMA READ_RESP`). In this case, the RX CC core marks the event with `GenTxEvent` (Line 48) and sets the `returned_credit` field in the extensible QP context as the received packet length (Line 46). At last, the RX CC core posts the event and notifies the NIC hardware.

In summary, SwCC provides programmers with easy-to-use programming abstractions. As such, SwCC allows developers to focus on the CCA logic instead of dealing with low-level NIC hardware interactions.

5 Evaluations

Our evaluations aim to answer the following questions:

- How does the control loop delay of SwCC compare to that of RoCE and Soft-RoCE (§5.2)?
- How much frequency can the TX/RX separated multi-core increase (§5.3)?
- How effective is the QP-aware memory subsystem (§5.4)?
- How are the flexibility and programmability of SwCC (§5.5)?
- How many resources do the SwCC use (§5.6)?
- How does the end-to-end performance of SwCC compare to that of ASIC-based NIC CC solutions (§5.7.1) and CPU-based CC solutions (§5.7.2)?

5.1 Experimental Setup

Experimental platform. The experimental platform consists of three servers, connected with a Wedge100BF-32X P4 reconfigurable switch. Each server is equipped with two Intel Xeon Silver 4214 CPUs @2.20GHz, 256GiB 2400MHz DDR4 memory, a Xilinx U280 FPGA, and a Mellanox ConnectX-5 NIC.

SwCC implementations. We prototype "SwCC" in Xilinx Alveo U280 FPGA (6K LoCs of Chisel3). SwCC mainly reuses the PCIe and CMAC modules of FpgaNIC [77]. There are mainly three extensions. First, we replace FpgaNIC's TCP/IP transport with our RoCEv2 implementation. Second, we place several RISC-V cores in the NIC, which are positioned between the transport layer and the PCIe module. The RISC-V cores use the open-source *riscvmini* core [15], features a simple 3-stage pipeline. We mainly modify the cache/memory subsystem and the interface to the hardware. Third, we propose a QP-aware memory subsystem and related hardware logic that serves the RISC-V cores to minimize data/packet access time during CCA processing.

The SwCC logic runs at 250 MHz. We implement five CCAs (DCQCN [88], TIMELY [55], HPCC [42], Swift [40], Homa [57]) in the SwCC.

RoCE Baseline. The ASIC-based NIC CC solution baseline, labeled "RoCE", runs traffic on the Mellanox ConnectX-5 NIC. We use MLNX_OFED 23.1 driver and enable DCQCN in the server.

Soft-RoCE Baseline. We implement CPU-based CC solution baseline, labeled "Soft-RoCE", based on Soft-RoCE [62]. We implement DCQCN (100 LoCs of C++), TIMELY (70 LoCs of C++) and HPCC (110 LoCs of C++) by referring to [42, 55, 88].

RDMA-HLS Baseline. The FPGA-based 100Gb RoCEv2 stack [74] baseline used in [68–70], labeled "RDMA-HLS", runs traffic on the Xilinx Alveo U280 FPGA. We implement DCQCN in the stack using HDL.

BF3 Baseline. We implement baseline "BF3" that uses PCC

in SoC SmartNIC Bluefield-3 [60] to implement DCQCN.

Parameter Setting. We implement DCQCN, TIMELY, and HPCC using SwCC and Soft-RoCE. In the two implementations, we use the same parameters except those related to RTT, which we configure according to the actual measurements from SwCC and Soft-RoCE. For DCQCN, we set K_{min} = $5KB$, K_{max} = $200KB$, P_{max} = 1% and $g = 1/256$ following the default parameter setting in [88]. For TIMELY, we set α = 0.875 and β = 0.8, as suggested in [55]. For HPCC, we set η = 0.95 and *maxStage* = 5, as suggested in [42].

5.2 Comparison of Control Loop Delay

We first measure the RTT control loop delay of `RDMA WRITE` with a packet size of 1 KB. Figure 7 illustrates the results. We have two observations.

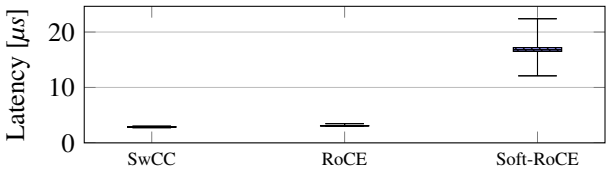


Figure 7: Control loop delay comparison. Whiskers show the 1st and 99th percentile.

First, both SwCC and RoCE achieve a similar average control loop delay of approximately 3.1 μs. This is because both approaches offload the CC controller to the NIC, avoiding additional PCIe traversals and software overhead. Second, Soft-RoCE exhibits significantly higher and more variable latency compared to SwCC and RoCE. Soft-RoCE runs the CC controller on the CPU software, which introduces PCIe traversals along with overheads from data copying, and kernel switching, resulting in a longer and less stable control loop delay. The SwCC design would not increase the control loop delay and can have nearly the same control loop delay as the ASIC-based solution.

5.3 Effect of TX/RX Separated Multi-Core

We then validate the effectiveness of TX/RX Separated Multi-Core by measuring the maximum frequency and the throughput of SwCC.

Figure 8 shows the FPGA routing with/without the TX/RX separated multi-core (two cores for both implementations). We observe that the TX/RX separated multi-core has much less (50%) routing wire from/to the CC cores. This is because the TX/RX separated multi-core allows a single core to connect to only a subset of NIC resources as a core only needs to process TX or RX logic.

The reduced routing wire can help to increase the achieved design frequency. Figure 9 shows the achieved frequency with/without TX/RX separated multi-core, under different numbers of RISC-V cores[3] with Vivado default implementation strategy. We observe that TX/RX separated multi-core can increase the achieved frequency by up to 16%. This is because TX/RX separated multi-core reduces the routing wire to/from the CC cores.

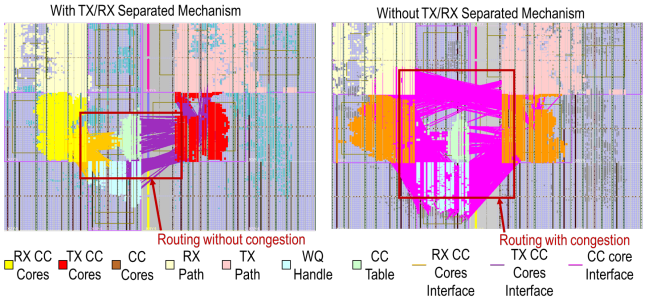


Figure 8: Comparison of routing with/without functional separation mechanism. "congestion" indicates the routing resource contention for circuits as the number of required routing wires increases.

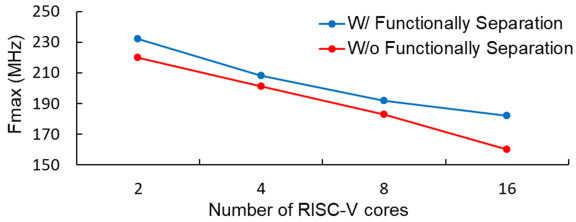


Figure 9: Comparison of the assessed F_{max} with/without TX/RX separation mechanism.

Effect of Core Number. We evaluate the effect of core number, by comparing the throughput of `RDMA WRITE`.

Figure 10 demonstrates the throughput of different RDMA implementations under different packet sizes.[4] The "SwCC-n" means that SwCC instantiates *n* "RX CC" cores and n "TX CC" cores. The "RoCE-n" means the utilization of n PUs in the CX-5 NIC. There are only 8 PUs in the CX-5 NIC, so "RoCE-8" has reached the maximum throughput of the CX-5 NIC [65]. The "Soft-RoCE-n" means the use of n threads in the CPU to run Soft-RoCE. The CPU in our testbed supports up to 24 hardware threads and we use up to 24 threads in Soft-RoCE.

We have three observations. First, SwCC can achieve comparable throughput to that of commercial NICs. "SwCC-8" and "RoCE-8" both achieve line rates with packet sizes above 512B. The throughput of "SwCC-1" is 1.1-1.5 times that of "RoCE-1". This is because SwCC uses two RISC-V cores (a "TX CC" and a "RX CC") to handle a single flow, while the CX-5 NIC uses only one PU to process a flow. Second, the throughput of Soft-RoCE is much lower than the other two implementations. Even with 24 threads in "Soft-RoCE-24", Soft-RoCE only achieves around 24Gbps throughput. This

[3]The calculation method is: F_{max} (MHz) = max(1000/(T - WNS)), where T is the target clock period (ns) and WNS is the worst negative slack (ns) of the target clock [3].

[4]Except for "RoCE-1", which uses a single QP, all other approaches utilize 24 QPs. This is because the CX5 NIC can only control the number of PUs (processing units) used by adjusting the number of QPs. Additionally, all approaches, except RDMA-HLS, enable DCQCN for CC.

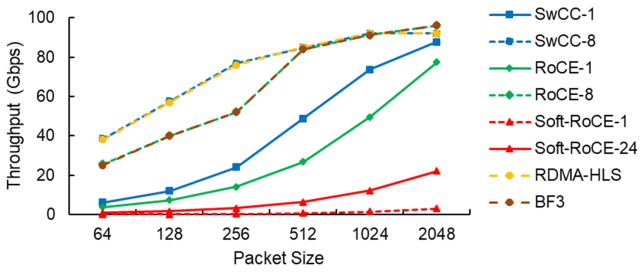
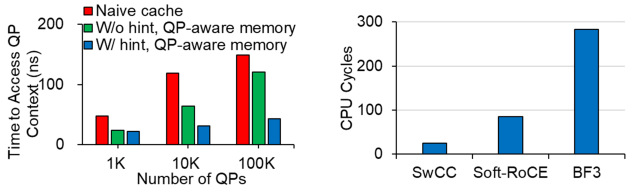
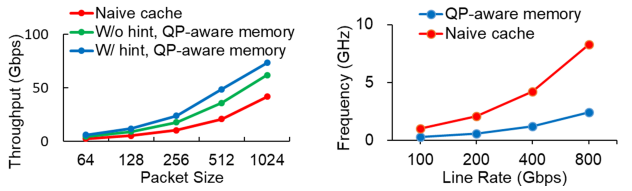


Figure 10: Comparison of throughput for various approaches using different numbers of cores/units/threads.



(a) Average time to access QP context.

(b) The cycles of CC controller.



(c) The `RDMA WRITE` throughput of SwCC.

(d) The frequency required to achieve various line rates.

Figure 11: The effect of QP-aware memory subsystem.

is because of the inefficiencies in the kernel network stack caused by data copying from the kernel to application buffers, kernel switching, etc [10]. Third, "SwCC" and "RDMA-HLS" achieve nearly identical performance as both run on FPGA-based SmartNICs. Likewise, "BF3" and "RoCE" deliver comparable performance because they use the same hardware network card.

5.4 Effect of QP-aware Memory Subsystem

We examine the effect of the QP-aware memory subsystem by comparing the CC core's access time to the QP context, CC controller execution cycles and throughput of `RDMA WRITE`. The size of QP context table is 128 KB, "w/o hint" and "w/ hint" indicate whether the QPN hint interface is used for the QP-aware memory subsystem. We also implement a naive memory subsystem, which includes a 4-way associative cache with a cache line size of 64B, and the size of cache is 128 KB, labeled "naive cache".

Time to Access QP Context. We establish 1K, 10K, and 100K QPs on two servers. The sender randomly selects a QP from these QPs with a uniform probability of initiating an `RDMA WRITE` request. We send 100M requests with each carrying a 1KB message and measure the time to access the QP context on the RISC-V core.

Figure 11a illustrates the average QP context access time for different approaches with different numbers of network flows. We observe that, regardless of the number of network flows running, QP-aware memory reduces memory access time by at least 50% compared to naive cache. When the number of QPs is 1K, the QP context is fully cached, the naive cache requires multiple load/store instructions to move the QP context into the registers, leading to additional delays. When the QP context is not fully cached, QP-aware memory can leverage its hint interface to prefetch the QP context, further reducing access time. As the number of QPs increases, the performance of SwCC remains nearly constant.

Execution Cycles. Figure 11b shows the cycles of different baselines for the CC controller running DCQCN. We have two observations. First, "Soft-RoCE" runs on the CPU and uses a naive memory system, and thus requires approximately 3× the cycles of "SwCC". Second, "BF3", significantly increases the total execution cycles, reaching 11.4× that of "SwCC". It is mainly due to BlueField-3's poor DPA memory subsystem.

Throughput. We compare the network throughput with naive cache and QP-aware memory under different packet sizes. We instantiated 1 pair of CC cores in SwCC, and we set the number of network flows to 100K. Each flow sends 1K `RDMA WRITE` requests with each carrying a 1 KB message.

As shown in Figure 11c, when one pair of CC cores is instantiated in SwCC, the QP-aware memory significantly impacts throughput. The throughput of the naive cache is approximately 40% of that of the QP-aware memory with small packets. This is because, in this case, the throughput bottleneck of the SwCC core lies in the CC core, where excessive CPU cycles are required to process each event, resulting in low throughput. Therefore, reducing the number of CPU cycles required by the CC core to process events can effectively increase the throughput of SwCC.

Potential ASIC Design. SwCC currently targets an FPGA-based SmartNIC, however, because it comprises fixed components and RISC-V cores, it readily supports an ASIC design, which offers higher clock frequencies and more efficient hardware resource utilization. We use simulations to estimate the processor frequency required to achieve line rate under a single QP with different memory subsystems, setting the RDMA message size to 1 KB. As shown in Figure 11d, naive cache requires up to 8 GHz to reach an 800 Gbps line rate, whereas QP-aware memory needs only 2.4 GHz, which is more realistic to achieve. This outcome demonstrates SwCC's potential for ultra-high-speed networks.

Regarding the power consumption of the ASIC design, we can compare SwCC with RISC-V cores in commercial SmartNICs. The BlueField-3 SmartNIC features a 1.8 GHz, 16-core/256-thread RISC-V core, with total NIC power consumption at 66 W. Since SwCC employs only a simple in-order, 3-stage 2-core RISC-V design at 2.4 GHz, we believe its projected power consumption to be reasonable for a NIC.

Table 3: C code lines for implementing various CCAs in SwCC.

CCA	CC Signal	Triggering Event	Adjustment Strategy	C code (lines)
DCQCN	ECN	CNP, DATA, Timer	rate	140
TIMELY	timestamp	ACK	rate	102
HPCC	INT	ACK	wnd	148
Swift	timestamp	ACK	wnd	164
Homa	token	DATA, GRANT, RESEND, BUSY	credit	95

Table 4: FPGA resource consumption.

Name	LUTs (K)	REGs (K)	BRAMs	URAMs
RDMA-HLS	26 (2%)	65(2.5%)	130 (6.4%)	0 (0%)
SwCC-1	51 (3.9%)	62 (2.4%)	220 (11%)	4 (0.4%)
SwCC-8	112 (8.6%)	124 (4.8%)	220 (11%)	32 (3.3%)
CC Core	8 (0.6%)	4 (0.2%)	0 (0%)	2 (0.2%)
QP-aware Memory	2.5 (0.2%)	0.5 (0.1%)	114 (5.6%)	0 (0%)

5.5 Flexible User-Defined CC Framework

Based on SwCC, we implement five representative CCAs that 1) use different CC signals, 2) trigger the CC controller at different times, and 3) adjust data transmission differently. Table 3 shows 1) what CC signals these CCAs use, 2) what types of packets trigger the CC controller when received by these CCAs, 3) which kind of data transmission strategies these CCAs use, and 4) lines of C code to implement these CCAs. We have two observations. First, SwCC supports a broad range of CCAs. SwCC achieves its design goal of high flexibility. Second, regardless of the algorithm, each can be implemented in less than 200 lines of code. This is because SwCC allows developers to focus on the CCA logic without dealing with complex hardware details. SwCC achieves its design goal of high programmability.

5.6 Hardware Resource Usage

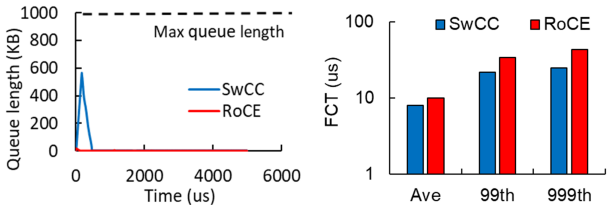
Table 4 shows the FPGA resource consumption of "RDMA-HLS" and "SwCC". "SwCC-1" requires the instantiation of RISC-V cores and supports extended CC headers of various lengths. Consequently, it utilizes slightly more resources compared to "RDMA-HLS". "SwCC-8", which instantiates 16 RISC-V cores, consumes approximately twice the hardware resources of "SwCC-1". Despite the increased resource usage, it only occupies under 10% resources, leaving ample capacity for other functionality.

5.7 End-to-End Performance

We compare "SwCC", "RoCE", and "SoftRoCE" to evaluate the end-to-end performance of "SwCC". We compare "SwCC" and "RoCE" on a 100Gbps network. The comparison between "SwCC" and "Soft-RoCE" is conducted on a 10Gbps network because "Soft-RoCE" cannot reach a line rate of 100Gbps.

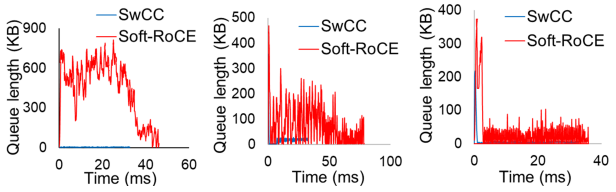
5.7.1 SwCC vs. RoCE

In the experiment, we use three servers, two as senders and one as a receiver. Each of two senders starts 5 flows to the receiver, where each flow sends 2K messages, each with a size of 2 KB. We implement the DCQCN algorithm in SwCC,



(a) Queue length (b) FCT

Figure 12: The end-to-end performance comparison of SwCC and RoCE.



(a) TIMELY (b) DCQCN (c) HPCC

Figure 13: Comparison of buffer occupancy at the congested port of the switch under various CCAs.

labeled "SwCC". We set the period of CNP generation in DCQCN as 50 μs, and the period of updating α as 55 μs, which follows the parameter setting in [88].

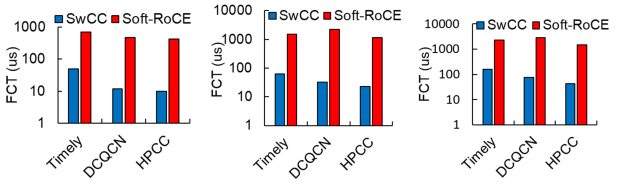
Figure 12 shows the buffer occupancy comparison and flow completion time. We observe that when the CX-5 NIC detects congestion in the "RoCE" baseline, it halts data transmission for approximately 90 μs. In contrast, SwCC's implementation of the DCQCN algorithm adjusts the transmission rate based on work [88]. Consequently, the buffer occupancy of "RoCE" drains more quickly than that of "SwCC", but it still remains significantly below the switch's maximum queue length. Additionally, the halt in transmission results in a slightly longer FCT for "RoCE" compared to "SwCC". In the absence of publicly available details of the NIC's CCAs, we can not devise an identical algorithm. In sum, "SwCC" delivers slightly better performance than "RoCE".

5.7.2 SwCC vs. Soft-RoCE

We implement the DCQCN, TIMELY, and HPCC in SwCC and Soft-RoCE, with the same settings as Section 5.7.1. In our testbed, the RTT is approximately 5 µs for SwCC and 30 µs for Soft-RoCE, respectively.

Figure 13 shows the switch buffer occupancy comparison under different algorithms for SwCC and Soft-RoCE during congestion. We observe that SwCC alleviates congestion more rapidly under different levels of network congestion. This is due to SwCC's shorter control loop, which increases its sensitivity to congestion, allowing it to respond more quickly and effectively mitigate network congestion.

Figure 14 shows the comparison of FCT under different algorithms for SwCC and Soft-RoCE during congestion. We observe that when implementing the TIMELY, DCQCN, and HPCC CCAs, the FCT of Soft-RoCE is 14×, 39×, and 42× higher than that of SwCC respectively. This is mainly because



(a) Average (b) 99th percentile (c) 999th percentile

Figure 14: Comparing of FCT statistics for SwCC and Soft-RoCE under various CCAs.

SwCC has a much lower control loop delay than Soft-RoCE.

6 Related Work

ASIC-based NIC CC solution. Some NICs offload the network stacks with hard-wired transport protocols [9, 35, 53]. However, they only implement at most two types of congestion control, including PFC [32] and DCQCN [61], and can only be modified by their vendor.

CPU-based CC solution. The "pluggable TCP" API [37] and QUIC [41] support user-space programming of congestion control. However, they do not bypass the kernel, resulting in poor performance. Kernel bypass libraries such as DPDK [16], netmap [67] and Snap [46] allow programs to send packets from userspace to the NIC. The mTCP [38] implements NewReno [18] on DPDK. Google deploys Swift [40] and 1RMA [71] on Snap. However, Snap's RTT latency for a single-sided operation is 8.8 µs, which is 2.8 × higher than the control loop latency of ASIC-based NICs.

FPGA-based SmartNIC CC solution. The emergence of FPGA-based SmartNICs [12, 30, 31, 70, 84, 85, 89] has made programmable CC on NICs feasible. Prior works [4, 5] enable programmers to implement a variety of transport protocols in hardware by Verilog or P4 language. However, the HDL offers low programmability. Some works [26, 64] implement rate-based CCAs using packet pacing, but they do not provide a programmable interface.

SoC-based SmartNIC system. Prior works [17, 23, 39, 43, 45, 72, 78, 79, 83] propose the frameworks or systems on SoC-based SmartNIC to alleviate the host CPU pressure. None of these works address the issue of a programmable CC controller. Many vendors [36, 58, 60, 80] have developed SoC-based SmartNICs that incorporate CPUs and network acceleration hardware. However, only a few of these SmartNICs [63] feature dedicated programmable congestion control interfaces, and their flexibility remains limited.

7 Conclusion

This paper presents SwCC, an RDMA engine integrated with a RISC-V processor that enables the execution of a wide range of CCAs directly on the RISC-V core, while keeping high performance. SwCC achieves an end-to-end control loop delay of 3 µs, which is comparable to that of leading commercial NICs while providing high flexibility and programmability for CCA development. The ASIC design of SwCC can easily scale to higher network bandwidth.

Acknowledgement. The work is supported by the following grants: the National Key R&D Program of China (Grant No. 2022ZD0119301), the National Natural Science Foundation of China under the grant numbers (62472384, 62441236, U24A20326). Zeke Wang is the corresponding author.

References

[1] Vamsi Addanki, Oliver Michel, and Stefan Schmid. {PowerTCP}: Pushing the performance limits of datacenter networks. In *NSDI*, 2022.

[2] Saksham Agarwal, Arvind Krishnamurthy, and Rachit Agarwal. Host congestion control. In *SIGCOMM*, 2023.

[3] AMD. UltraFast Design Methodology Guide for FPGAs and SoCs. https://docs.amd.com/r/en-US/ug949-vivado-design-methodology, 2024.

[4] Mina Tahmasbi Arashloo, Alexey Lavrov, Manya Ghobadi, Jennifer Rexford, David Walker, and David Wentzlaff. Enabling programmable transport protocols in {High-Speed}{NICs}. In *NSDI*, 2020.

[5] Serhat Arslan, Stephen Ibanez, Alex Mallery, Changhoon Kim, and Nick McKeown. Nanotransport: A low-latency, programmable transport layer for nics. In *SOSR*, 2021.

[6] Serhat Arslan, Yuliang Li, Gautam Kumar, and Nandita Dukkipati. Bolt:{Sub-RTT} congestion control for {Ultra-Low} latency. In *NSDI*, 2023.

[7] Venkat Arun and Hari Balakrishnan. Copa: Practical {Delay-Based} congestion control for the internet. In *NSDI*, 2018.

[8] Wei Bai, Shanim Sainul Abdeen, Ankit Agrawal, Krishan Kumar Attre, Paramvir Bahl, Ameya Bhagat, Gowri Bhaskara, Tanya Brokhman, Lei Cao, Ahmad Cheema, et al. Empowering azure storage with {RDMA}. In *NSDI*, 2023.

[9] Broadcom. BCM957508-P2100G Dual-Port 100 Gb/s QSFP56 Ethernet PCI Express 4.0 x16 Network Interface Card. https://docs.broadcom.com/doc/957508-P2100G-DS, 2019.

[10] Qizhe Cai, Shubham Chaudhary, Midhul Vuppalapati, Jaehyun Hwang, and Rachit Agarwal. Understanding host network stack overheads. In *SIGCOMM*, 2021.

[11] Neal Cardwell, Yuchung Cheng, C Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. Bbr: congestion-based congestion control. *CACM*, 2017.

[12] Adrian M Caulfield, Eric S Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, Joo-Young Kim, et al. A cloud-scale acceleration architecture. In *MICRO*, 2016.

[13] Xuzheng Chen, Jie Zhang, Ting Fu, Yifan Shen, Shu Ma, Kun Qian, Lingjun Zhu, Chao Shi, Yin Zhang, Ming Liu, et al. Demystifying datapath accelerator enhanced off-path smartnic. *arXiv preprint arXiv:2402.03041*, 2024.

[14] Inho Cho, Keon Jang, and Dongsu Han. Credit-scheduled delay-bounded congestion control for datacenters. In *SIGCOMM*, 2017.

[15] Donggyu Kim. riscv-mini. https://github.com/ucb-bar/riscv-mini, 2024.

[16] DPDK. https://dpdk.org/.

[17] Yixiao Feng, Sourav Panda, Sameer G Kulkarni, KK Ramakrishnan, and Nick Duffield. A smartnic-accelerated monitoring platform for in-band network telemetry. In *LANMAN*, 2020.

[18] Floyd, Sally and Henderson, Tom and Gurtov, Andrei. The NewReno modification to TCP's fast recovery algorithm. https://www.rfc-editor.org/rfc/rfc3782, 2004.

[19] Peter X Gao, Akshay Narayan, Gautam Kumar, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. phost: Distributed near-optimal datacenter transport over commodity network fabric. In *CoNEXT*, 2015.

[20] Yixiao Gao, Qiang Li, Lingbo Tang, Yongqing Xi, Pengcheng Zhang, Wenwen Peng, Bo Li, Yaohui Wu, Shaozong Liu, Lei Yan, et al. When cloud storage meets {RDMA}. In *NSDI*, 2021.

[21] Prateesh Goyal, Akshay Narayan, Frank Cangialosi, Srinivas Narayana, Mohammad Alizadeh, and Hari Balakrishnan. Elasticity detection: A building block for internet congestion control. In *SIGCOMM*, 2022.

[22] Prateesh Goyal, Preey Shah, Kevin Zhao, Georgios Nikolaidis, Mohammad Alizadeh, and Thomas E. Anderson. Backpressure flow control. In *NSDI*, 2022.

[23] Stewart Grant, Anil Yelam, Maxwell Bland, and Alex C Snoeren. Smartnic performance isolation with fairnic: Programmable networking for the cloud. In *SIGCOMM*, 2020.

[24] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. Rdma over commodity ethernet at scale. In *SIGCOMM*, 2016.

[25] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. Rdma over commodity ethernet at scale. In *SIGCOMM*, 2016.

[26] Y Sinan Hanay, Abhishek Dwaraki, and Tilman Wolf. High-performance implementation of in-network traffic pacing. In *HPSR*, 2011.

[27] Mark Handley, Costin Raiciu, Alexandru Agache, Andrei Voinescu, Andrew W Moore, Gianni Antichi, and Marcin Wójcik. Re-architecting datacenter networks and stacks for low latency and high performance. In *SIGCOMM*, 2017.

[28] Shuihai Hu, Yibo Zhu, Peng Cheng, Chuanxiong Guo, Kun Tan, Jitendra Padhye, and Kai Chen. Deadlocks in datacenter networks: Why do they form, and how to avoid them. In *HotNets*, 2016.

[29] Shuihai Hu, Yibo Zhu, Peng Cheng, Chuanxiong Guo, Kun Tan, Jitendra Padhye, and Kai Chen. Tagger: Practical pfc deadlock prevention in data center networks. In *CoNEXT*, 2017.

[30] Hongjing Huang, Yingtao Li, Jie Sun, Xueying Zhu, Jie Zhang, Liang Luo, Jialin Li, and Zeke Wang. P4sgd: Programmable switch enhanced model-parallel training on generalized linear models on distributed fpgas. *TPDS*, 2023.

[31] Hongjing Huang, Zeke Wang, Jie Zhang, Zhenhao He, Chao Wu, Jun Xiao, and Gustavo Alonso. Shuhai: A tool for benchmarking high bandwidth memory on fpgas. *IEEE Transactions on Computers*, 2021.

[32] IEEE. Priority-based Flow Control. https://1.ieee802.org/dcb/802-1qbb/, 2011.

[33] InfiniBand. RoCEv2. https://www.infinibandta.org/infiniband-trade-association-releases-updated-specification-for-remote-direct-memory-access-over-converged-ethernet-roce/, 2014.

[34] InfiniBand. InfiniBandTM Architecture Specification Volume 1. https://www.infinibandta.org/ibta-specification/, 2020.

[35] Intel. Intel® Ethernet Network Adapter E810-CQDA2T. https://cdrdv2.intel.com/v1/dl/getContent/726375, 2022.

[36] Intel. Intel® IPU Platform F2000X-PL. https://www.intel.com/content/www/us/en/products/details/network-io/ipu/f2000x-pl-platform.html#tab-blade-1-2, 2023.

[37] J. Corbet. Pluggable congestion avoidance modules. https://lwn.net/Articles/128681/, 2005.

[38] EunYoung Jeong, Shinae Wood, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. {mTCP}: a highly scalable user-level {TCP} stack for multicore systems. In *NSDI*, 2014.

[39] Mikhail Khalilov, Marcin Chrapek, Siyuan Shen, Alessandro Vezzu, Thomas Benz, Salvatore Di Girolamo, Timo Schneider, Daniele De Sensi, Luca Benini, and Torsten Hoefler. {OSMOSIS}: Enabling {Multi-Tenancy} in datacenter {SmartNICs}. In *ATC*, 2024.

[40] Gautam Kumar, Nandita Dukkipati, Keon Jang, Hassan MG Wassel, Xian Wu, Behnam Montazeri, Yaogong Wang, Kevin Springborn, Christopher Alfeld, Michael Ryan, et al. Swift: Delay is simple and effective for congestion control in the datacenter. In *SIGCOMM*, 2020.

[41] Adam Langley, Alistair Riddoch, Alyssa Wilk, Antonio Vicente, Charles Krasic, Dan Zhang, Fan Yang, Fedor Kouranov, Ian Swett, Janardhan Iyengar, et al. The quic transport protocol: Design and internet-scale deployment. In *SIGCOMM*, 2017.

[42] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, et al. Hpcc: High precision congestion control. In *SIGCOMM*. 2019.

[43] Ming Liu, Tianyi Cui, Henrik Schuh, Arvind Krishnamurthy, Simon Peter, and Karan Gupta. ipipe: A framework for building distributed applications on multicore soc smartnics. In *SIGCOMM*, 2019.

[44] Shiyu Liu, Ahmad Ghalayini, Mohammad Alizadeh, Balaji Prabhakar, Mendel Rosenblum, and Anirudh Sivaraman. Breaking the {Transience-Equilibrium} nexus: A new approach to datacenter packet transport. In *NSDI*, 2021.

[45] Yiqing Ma, Han Tian, Xudong Liao, Junxue Zhang, Weiyan Wang, Kai Chen, and Xin Jin. Multi-objective congestion control. In *Eurosys*, 2022.

[46] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C Evans, Steve Gribble, et al. Snap: A microkernel approach to host networking. In *SOSP*, 2019.

[47] Mellanox. ConnectX®-3 Ethernet Single and Dual SFP+ Port Adapter Card User Manual. https://network.nvidia.com/related-docs/user_manuals/ConnectX-3_Ethernet_Single_and_Dual_SFP+_Port_Adapter_Card_User_Manual.pdf, 2015.

[48] Mellanox. ConnectX-6. https://www.nvidia.com/en-sg/networking/ethernet/connectx-6/, 2018.

[49] Mellanox. ConnectX®-4 Lx EN Card. https://network.nvidia.com/files/doc-2020/pb-connectx-4-lx-en-card.pdf, 2020.

[50] Mellanox. ConnectX®-5 EN Card. https://network.nvidia.com/files/doc-2020/pb-connectx-5-en-card.pdf, 2020.

[51] Mellanox. ConnectX-6 Lx 25G/50G Ethernet NIC. https://nvdam.widen.net/s/nqbftvdlh7/connectx-6-lx-en-card-product-brief-1731006, 2021.

[52] Mellanox. NVIDIA CONNECTX-6 DX. https://www.nvidia.com/content/dam/en-zz/Solutions/networking/ethernet-adapters/connectX-6-dx-datasheet.pdf, 2021.

[53] Mellanox. NVIDIA CONNECTX-7 400G ETHERNET. https://www.nvidia.com/content/dam/en-zz/Solutions/networking/ethernet-adapters/connectx-7-datasheet-Final.pdf, 2021.

[54] Zili Meng, Yaning Guo, Chen Sun, Bo Wang, Justine Sherry, Hongqiang Harry Liu, and Mingwei Xu. Achieving consistent low latency for wireless real-time communications with the shortest control loop. In *SIGCOMM*, 2022.

[55] Radhika Mittal, Vinh The Lam, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. Timely: Rtt-based congestion control for the datacenter. *SIGCOMM*, 2015.

[56] Radhika Mittal, Alexander Shpiner, Aurojit Panda, Eitan Zahavi, Arvind Krishnamurthy, Sylvia Ratnasamy, and Scott Shenker. Revisiting network support for rdma. In *SIGCOMM*, 2018.

[57] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John Ousterhout. Homa: A receiver-driven low-latency transport protocol using network priorities. In *SIGCOMM*, 2018.

[58] Netronome. Agilio CX SmartNICs. https://netronome.com/agilio-smartnics/, 2024.

[59] Rolf Neugebauer, Gianni Antichi, José Fernando Zazo, Yury Audzevich, Sergio López-Buedo, and Andrew W Moore. Understanding pcie performance for end host networking. In *SIGCOMM*, 2018.

[60] NVIDIA. NVIDIA BLUEFIELD-3 DPU. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/documents/datasheet-nvidia-bluefield-3-dpu.pdf, 2021.

[61] NVIDIA. HOWTO CONFIGURE DCQCN (ROCE CC) VALUES FOR CONNECTX-4 (LINUX). https://enterprise-support.nvidia.com/s/article/howto-configure-dcqcn--roce-cc--values-for-connectx-4--linux-x, 2022.

[62] NVIDIA. HOWTO CONFIGURE SOFT-ROCE. https://enterprise-support.nvidia.com/s/article/howto-configure-soft-roce, 2022.

[63] NVIDIA. NVIDIA DOCA PCC Application Guide. https://docs.nvidia.com/doca/sdk/nvidia+doca+pcc+application+guide/index.html, 2023.

[64] Sivasankar Radhakrishnan, Yilong Geng, Vimalkumar Jeyakumar, Abdul Kabbani, George Porter, and Amin Vahdat. {SENIC}: Scalable {NIC} for {End-Host} rate limiting. In *NSDI*, 2014.

[65] Waleed Reda, Marco Canini, Dejan Kostić, and Simon Peter. {RDMA} is turing complete, we just did not know it yet! In *NSDI*, 2022.

[66] RISC-V. The RISC-V Instruction Set Manual: Volume II. https://github.com/riscv/riscv-isa-manual/releases/tag/20240411, 2024.

[67] Luigi Rizzo. netmap: a novel framework for fast packet i/o. In *USENIX Security*, 2012.

[68] David Sidler, Gustavo Alonso, Michaela Blott, Kimon Karras, Kees Vissers, and Raymond Carley. Scalable 10gbps tcp/ip stack architecture for reconfigurable hardware. In *FCCM*, 2015.

[69] David Sidler, Zsolt István, and Gustavo Alonso. Low-latency tcp/ip stack for data center applications. In *FPL*, 2016.

[70] David Sidler, Zeke Wang, Monica Chiosa, Amit Kulkarni, and Gustavo Alonso. Strom: smart remote memory. In *EuroSys*, pages 1–16, 2020.

[71] Arjun Singhvi, Aditya Akella, Dan Gibson, Thomas F Wenisch, Monica Wong-Chan, Sean Clark, Milo MK Martin, Moray McLaren, Prashant Chandra, Rob Cauble, et al. 1rma: Re-envisioning remote memory access for multi-tenant datacenters. In *SIGCOMM*, 2020.

[72] Chihun Song, Michael Jaemin Kim, Tianchen Wang, Houxiang Ji, Jinghan Huang, Ipoom Jeong, Jaehyun Park, Hwayong Nam, Minbok Wi, Jung Ho Ahn, and Nam Sung Kim. Tarot: A cxl smartnic-based defense against multi-bit errors by row-hammer attacks. In *ASPLOS*, 2024.

[73] Brent Stephens, Alan L Cox, Ankit Singla, John Carter, Colin Dixon, and Wesley Felter. Practical dcb for improved data center networks. In *INFOCOM*, 2014.

[74] Systems Group, ETH Zurich. Scalable Network Stack supporting TCP/IP, RoCEv2, UDP/IP at 10-100Gbit/s. https://github.com/fpgasystems/fpga-network-stack?tab=readme-ov-file.

[75] Parvin Taheri, Danushka Menikkumbura, Erico Vanini, Sonia Fahmy, Patrick Eugster, and Tom Edsall. Rocc: robust congestion control for rdma. In *CoNEXT*, 2020.

[76] Weitao Wang, Masoud Moshref, Yuliang Li, Gautam Kumar, TS Eugene Ng, Neal Cardwell, and Nandita Dukkipati. Poseidon: Efficient, robust, and practical datacenter {CC} via deployable {INT}. In *NSDI*, 2023.

[77] Zeke Wang, Hongjing Huang, Jie Zhang, Fei Wu, and Gustavo Alonso. {FpgaNIC}: An {FPGA-based} versatile 100gb {SmartNIC} for {GPUs}. In *ATC*, 2022.

[78] Xingda Wei, Rongxin Cheng, Yuhan Yang, Rong Chen, and Haibo Chen. Characterizing off-path SmartNIC for accelerating distributed systems. In *OSDI*, 2023.

[79] Shaoke Xi, Fuliang Li, Lingxiang Hu, Xingwei Wang, and Kui Ren. Flowvalve+: Multi-queue packet scheduling framework on soc-based smartnics. *TSC*, 2024.

[80] Xilinx. ALVEO™ SN1000 SmartNICs. https://www.xilinx.com/content/dam/xilinx/publications/product-briefs/xilinx-alveo-sn1000-product-brief.pdf, 2023.

[81] Siyu Yan, Xiaoliang Wang, Xiaolong Zheng, Yinben Xia, Derui Liu, and Weishan Deng. Acc: Automatic ecn tuning for high-speed datacenter networks. In *SIGCOMM*, 2021.

[82] Chen-Yu Yen, Soheil Abbasloo, and H Jonathan Chao. Computers can learn from the heuristic designs and master internet congestion control. In *SIGCOMM*, 2023.

[83] Jie Zhang, Xuzheng Chen, Yin Zhang, and Zeke Wang. Dmrpc: Disaggregated memory-aware datacenter rpc for data-intensive applications. In *ICDE*, 2024.

[84] Jie Zhang, Hongjing Huang, Xuzheng Chen, Xiang Li, Jieru Zhao, Ming Liu, and Zeke Wang. Rpcnic: Enabling efficient datacenter rpc offloading on pcie-attached smartnics. In *HPCA*, 2025.

[85] Jie Zhang, Hongjing Huang, Lingjun Zhu, Shu Ma, Dazhong Rong, Yijun Hou, Mo Sun, Chaojie Gu, Peng Cheng, Chao Shi, et al. Smartds: Middle-tier-centric smartnic enabling application-aware message split for disaggregated block storage. In *ISCA*, 2023.

[86] Yiran Zhang, Yifan Liu, Qingkai Meng, and Fengyuan Ren. Congestion detection in lossless networks. In *SIGCOMM*, 2021.

[87] Xiaolong Zhong, Jiao Zhang, Yali Zhang, Zixuan Guan, and Zirui Wan. Pacc: Proactive and accurate congestion feedback for rdma congestion control. In *INFOCOM*, 2022.

[88] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. Congestion control for large-scale rdma deployments. *SIGCOMM*, 2015.

[89] Noa Zilberman, Yury Audzevich, Georgina Kalogeridou, Neelakandan Manihatty-Bojan, Jingyun Zhang, and Andrew Moore. Netfpga: Rapid prototyping of networking devices in open source. *SIGCOMM*, 2015.

A Artifact Appendix

A.1 Abstract

This artifact provides the source code of SwCC and scripts to reproduce the main experimental results. The experiments are run on two 4U AMAX servers, each equipped with two Intel Xeon Silver 4214 CPUs@2.2GHz, 256GB DDR4 memory, a Xilinx U280 FPGA, a ConnectX-5 NIC. SwCC is implemented on Xilinx Alveo cards U280 with Vivado 2020.1.

A.2 Artifact check-list

- **Program:** C/C++
- **Compilation:** g++-11.3.0, gcc-11.3.30
- **Run-time environment:** QDMA driver installed
- **Hardware:** Xilinx Alveo U280
- **Execution:** Running commands as root with sudo
- **Metrics:** control loop delay and the throughput for various approaches using different numbers of cores/units/threads
- **Output:** Experiments produce outputs in the console or bin files
- **Experiments:** a) Control loop delay comparison between SwCC, RoCE and Soft-RoCE, b) Throughput comparisons between SwCC-1, SwCC-8, RoCE-1, RoCE-8, Soft-RoCE-1 and Soft-RoCE-24
- **Disk space required:** 1GB
- **Time needed to prepare workflow:** 1 hour
- **Time needed to complete the experiments:** 3 hours
- **Publicly available:** Yes
- **Code licenses:** MIT
- **Data licenses:** MIT

A.3 Description

A.3.1 How to access

The codebase can be accessed from GitHub https://github.com/RC4ML/SwCC.git.

A.4 Installation

Use the following commands to clone the SwCC repository, install the necessary tools and build binary programs.

```
git clone --recursive https://github.com/RC4ML/SwCC.git

## Install QDMA driver
cd qdma_driver
make
sudo insmod /path/to/qdma_driver/src/qdma-pf.ko
echo '1024' | sudo tee -a
     /sys/bus/pci/devices/0000:1a:00.0/qdma/qmax
sudo dma-ctl qdma1a000 q add idx 0 mode st dir bi
sudo dma-ctl qdma1a000 q start idx 0 dir bi desc_bypass_en
     pfetch_bypass_en

## Install MLNX_OFED
wget https://content.mellanox.com/ofed/MLNX_OFED
```

```
-23.04-1.1.3.0/MLNX_OFED_LINUX-23.04-1.1.3.0
-ubuntu18.04-x86_64.tgz -O mlnx.tgz
tar -zxvf ./mlnx.tgz
cd mlnx && sudo ./ofedinstall

## build SwCC programs
mkdir SwCC/build
cd SwCC/build
cmake ..
make

## Build baseline software
mkdir Baseline/libr/build
cd Baseline/libr/build
cmake ..
make

## After completing the RoCE experiments, follow these steps to
     prepare for the Soft-RoCE experiments
### Uninstall MLNX_OFED
cd ~/MLNX_OFED_LINUX-23.04-1.1.3.0-ubuntu18.04-x86_64/
sudo ./uninstall.sh
sudo reboot

### Load kernel modules and install required libs
sudo modprobe ib_core
sudo modprobe ip6_udp_tunnel
sudo modprobe udp_tunnel
sudo modprobe ib_uverbs
sudo depmod -a
cd ~/SwCC/Baseline/rxe/
make -j
sudo insmod ./rdma_rxe.ko
sudo apt install librdmacm-dev libibverbs-dev libibumad-dev
     libpci-dev

### Environment setup in atc25@r4:
sudo rdma link add sr4 type rxe netdev enp62s0np0
sudo ifconfig enp62s0np0 mtu 9000 up
sudo service irqbalance stop

### Environment setup in atc25@r3:
sudo rdma link add sr3 type rxe netdev enp28s0np0
sudo ifconfig enp28s0np0 mtu 9000 up
sudo service irqbalance stop
```

A.5 Experiment workflow

We provide three machines for artifact evaluation: a **Sender**(atc@r4), a **Receiver**(atc@r3), and a **Vivado machine**(atc@max). Both the **Sender** and the **Receiver** are equiped with a Xilinx U280 FPGA and a ConnectX-5 NIC machine. The **Sender** and the **Receiver** are used for the experiment, and the **Vivado machine** is used for deploying bitstream.

You can refer to our GitHub repo to see how to connect and deploy bitstream on FPGA. Please reboot the FPGA machine after programming the FPGA. And then you can run the SwCC experiment on the FPGA machine.

A.6 Evaluation and expected results

We use the control loop delay experiment of SwCC as an example, you can find other evaluations in our GitHub repo.

A.6.1 Run the control loop delay experiment

Program **bistream/SwCC-1.bit** to the FPGA on both the sender and receiver, and after reboot the machines, run the following command to start the experiment:

```
## On the Receiver:
sudo SwCC/build/example/latency_receiver
## On the Sender:
sudo SwCC/build/example/latency_sender
```

Then the Receiver and Sender display the messages.

```
## On the Receiver:
After both the sender and receiver programs are started, enter
     any value to launch the receiver:
## On the Sender:
After the receiver is launched, enter any value to begin the
     latency test.
```

Entering any value in the Receiver, then entering any value in the Sender. The output will be like this:

```
5th percentile: 3096 ns
25th percentile: 3168 ns
50th percentile (median): 3168 ns
75th percentile: 3168 ns
95th percentile: 3168 ns
```

A.7 Notes

Before running Soft-RoCE experiments, it is necessary to uninstall MLNX_OFED and configure the Soft-RoCE environment. Therefore, we strongly recommend completing all RoCE-related experiments before proceeding with Soft-RoCE experiments.

DRack: A CXL-Disaggregated Rack Architecture to Boost Inter-Rack Communication

Xu Zhang[1,2] , Ke Liu[1,2] , Yuan Hui[3] , Xiaolong Zheng[3] , Yisong Chang[1,2] , Yizhou Shan[4]
Guanghui Zhang[5] , Ke Zhang[1,2] , Yungang Bao[1,2] , Mingyu Chen[1,2] , Chenxi Wang[1,2]
[1]*SKLP, Institute of Computing Technology, CAS*
[2]*University of Chinese Academy of Sciences*
[3]*Huawei,* [4]*Huawei Cloud,* [5]*Shandong University*

Abstract

Data-intensive applications are scaling out across more and more racks, and boosted with advanced computing units with enhanced throughput, which necessitates increased NIC capacity and network bandwidth to transport inter-rack traffic. As a result, when running them over ToR-centric racks, inter-rack traffic can be bottlenecked at host NICs and core network due to oversubscription. However, we observe that, although a large volume of inter-rack traffic exists, the utilization of the host's NICs within a rack remains low. If those underutilized NICs within a rack can be utilized by any host, inter-rack communication can be accelerated. Therefore, we propose DRack. At its core, DRack disaggregates all NICs within a rack from their hosts, forming a shared NIC pool. As the local memory bandwidth or the PCIe link at a host is much smaller than the NIC pool capacity, the host cannot fully utilize the NIC pool. DRack also disaggregates memory devices within a rack from their hosts, so that data from the NIC pool can be written and read from multiple memory with full capacity, while host processors can directly access the memory pool with memory semantics. We realize DRack with CXL as it supports device pooling and memory semantics, which is well-suited to our designs. We have implemented DRack prototype and evaluated it with real applications, such as DNN training and graph processing. The result shows that DRack can reduce the communication stage by an average of 37.3% compared to ToR-centric rack.

1 Introduction

Today's production clouds organize hosts in ToR-centric racks. Each rack hosts tens of hosts connected via a Top-of-Rack (ToR) switch and is linked to the network core. Data-intensive applications running in those clouds, such as graph computing [1], data analytics [2], and deep neural network (DNN) training [3] often operate with Bulk Synchronous Parallel (BSP) [4] or MapReduce paradigms [2], generally involving computation stage and communication stages (*e.g.*, data shuffling and synchronization). They tend to scale the job out across more and more racks for acceleration, resulting in a large volume of across-rack traffic. As measured by Facebook during 24 hours [5], an average of 87.1% of network traffic in their datacenter needs to go out of the rack. Moreover, recent innovations on processors and domain-specific accelerators, such as GPUs [6, 7], and FPGAs [8], have significantly enhanced computing throughput, which necessitates increased NIC capacity and network bandwidth to accommodate. Thus, the two trends result in communication bottlenecks for inter-rack communication, including NIC egress/ingress port, and core network bottleneck due to oversubscription that is predominant in datacenter networks (DCN) (§2.1) .

We can address these bottlenecks intuitively with a brute-force approach, by overprovisioning NICs for every hosts [9–11], or switching equipments [10]. It is cost-prohibitive in terms of power consumption and hardware expenses. Another line [12–14] adds reconfigurable hardware to provide extra bandwidth between the intensively communicating racks, or to enable a reconfigurable network that changes its topology to adapt dynamic inter-rack traffic volume. However, they are limited by reconfiguration delay and complexity, and face challenges in predicting inter-rack traffic pattern that is bursty and highly dynamic [5]. Last, some job schedulers [15–20] reduce inter-rack traffic by scheduling jobs based on predictable features, *e.g.*, the data volume to send, which is hard to predict for existing data-intensive applications (see §2.3).

By studying the traffic trace from Facebook [5] and running real-world applications in our cluster, we find, *although a large volume of inter-rack traffic prevails [12], the hosts NIC utilization within a rack remains low (for both ingress and egress)*. The reason can be two-fold: 1) there is no network traffic during job computing stages and the traffic volume to send/receive during communication is highly skewed between hosts; 2) each job has its resource requirement and emphasis, resulting in resource fragmentation, *e.g.*, non-distributed DNN training jobs leave the NICs unused [21].

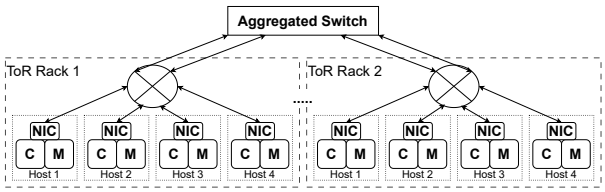
Inspired, a natural direction to explore is, *if those underutilized NICs from other hosts within a rack can be borrowed, could inter-rack communication be accelerated?* Therefore, we propose Disaggregated Rack (DRack), a new rack architec-

ture concept with the following three key designs. ① DRack disaggregates all the NICs of existing hosts to form a rack-level NIC pool which can be used by any host to accelerate its inter-rack traffic. However, maximizing the utilization of NIC pool capacity poses a challenge. This is because, when writing the data from the NIC pool to the memory of destined hosts, the NIC pool's capacity can surpass the bandwidth of the PCIe links at the destined hosts, or the total memory bandwidth of their local memory, given the number of communicating hosts is small. Thus, ② DRack also disaggregates local memory from hosts, forming a rack-level memory pool with an aggregated bandwidth exceeding the capacity of the NIC pool, so that data can be written/read to/from multiple memory devices from the NIC pool with the full capacity. ③ DRack allows host processors or accelerators (*e.g.*, CPU) to load and store the memory pool with memory semantics during computation, without moving data to their local memory first with DMA. This approach contrasts with prior works that relied on PCIe for device pooling [22–24].

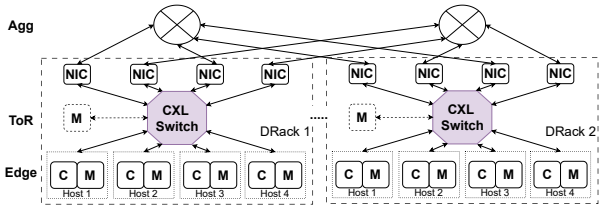
To realize the above three key designs, we find that CXL is a perfect fit due to the following facts: 1) CXL 2.0 [25] supports memory and IO device pooling with CXL switches which are responsible for routing and address remapping when the host is accessing any device attached to it. Consequently, CXL 3.0 enhances it by proposing CXL fabric, which is an interconnect that supports rack-level resource polling (designs ① and ②). 2) CXL supports memory semantics (CXL.mem), which enables hosts to load and store the memory pool directly throughout the computation stage (③).

By doing so, DRack alleviates the bottlenecks in the inter-rack communication when running over the ToR-centric architecture, and its new architecture complements existing job scheduling algorithms with a better communication efficiency [26, 27]. Compared to prior architectures, DRack is static without the complexity of reconfigurability and traffic prediction, and does not add more hardware for more bandwidth across racks, while DRack can be flexible and scalable by simply attaching more memory/NIC devices to the CXL interconnect.

We still face several challenges in porting real applications/jobs and ensuring DRack efficiency (see §2.4). For example, CXL.mem load/store to remote memory (in other hosts) exhibits higher latency than accessing the local memory, which could impact the computation efficiency. We solved them and implemented a full functional DRack prototype with 8 customized MPSoC FPGAs and one server connected by optical fibers, instead of using simulator and NUMA nodes in prior works [28–31], enabling running different applications transparently on DRack with various (CXL) bandwidth and latency configurations. We realized CXL 3.0-like memory protocols, such as CXL.mem and CXL.io, atop an academia lightweight conceptual hardware protocol stack [32]. We evaluated DRack with both microbenchmarks, real data-intensive applications, such as DNN training, graph processing, and KV store, and user cases like job scheduling, and show that DRack reduces the communication time by an average of 37.3% compared to the conventional ToR-based rack. DRack also achieves 62.2% lower p99 tail latency when running latency-sensitive applications, *e.g.*, Redis.



(a) Conventional Rack-centric architecture.



(b) DRack: a logical shared memory and NIC pool

Figure 1: The architectural comparison between two racks. CPU (c), Memory (M).

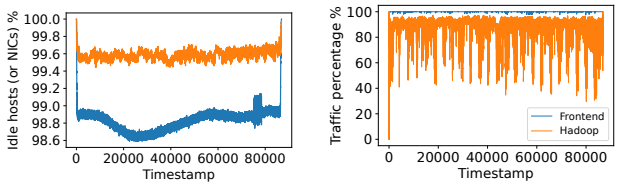
2 Background and Motivations

In this section, we first elucidate the reason why inter-rack communication is prevalent and inevitable in conventional rack-centric datacenter networks (DCN), and why it is critical to the performance of data-intensive applications. Second, we review prior approaches aimed at enhancing inter-rack communication efficiency. Third, we reveal a unique insight resulting from workload execution paradigms (*e.g.*, BSP) and job placements – the utilization of existing NICs' full-duplex bandwidth is low, *e.g.*, 20%-45% reported in [21]. This insight motivates the DRack strawman design, which decouples NICs from their physical hosts and pools them to form a rack-level NIC pool. Last, we discuss the use cases that can benefit from DRack and the challenges that must be addressed to realize DRack in practical.

2.1 Inter-rack Communication in DCNs

Although conventional DCNs, such as public clouds in Amazon [33] and Tencent [34], can vary from one deployment to another. They have one property in common – they arrange hosts in racks as the basic unit. As shown in Figure 1(a), a ToR switch interconnects tens of hosts via Ethernet links, all of which are stacked within a physical rack. Racks connect to the aggregation layer with oversubscribed uplinks. This *ToR-based rack architecture* streamlines the management process, while aiming to offer high throughput and low latency for application traffic with a high degree of rack locality.

Inter-rack traffic is rising and inevitable. We conducted an analysis of a public dataset from Facebook [5], which en-



(a) The percentage of hosts (or NICs) are not communicating.

(b) The percentage of the volume of inter-rack traffic.

Figure 2: The network traffic patterns motivating DRack.

compasses packet-level traces gathered from both Hadoop and Frontend production clusters over one day, sampled at a rate of 1:30k. Each packet sample includes source and destination hosts. As depicted in Figure 2(b), an average of 86.7% and 97.3% of Hadoop and Frontend traffic, respectively, is directed to hosts located in different racks, thereby underscoring the significant demand for inter-rack communication. This observation aligns with prior works [5, 12] and can be attributed to the following factors:

❶ Service-based rack organization: for operation convenience, a rack is typically dedicated to hosts serving a specific (Frontend) service. *e.g.*, cache servers primarily send responses across racks to web servers in other racks, which issued cache read requests. ❷ Resource fragmentation: a Hadoop (Map-Reduce) job may span multiple racks if rack resources are fragmented, thereby resulting in heavy inter-rack traffic for that job. The fragmentation occurs as the scheduler endeavors to localize jobs within a rack [35]; second, dynamic job churns ensure that rack resource is not always neatly packed [36, 37]. ❸ Large scale applications: Data-intensive applications like data analytics [2], iterative graph processing [1] and machine learning [3] (ML), is scaled out across 100 to 1K of hosts [38, 39], *i.e.*, tens or hundreds of racks, to accelerate computation over large input dataset. Their executions follow BSP and Map-Reduce paradigms that require extensive data communication between racks (hosts) after computation, *e.g.*, gradient synchronization.

Inter-rack communication efficiency is crucial for applications. Data-intensive applications are increasingly leveraging advanced computing units such as CPU, GPU and FPGA to execute compute-intensive tasks [6–8]. This trend significantly reduces computation time and generates data at rates exceeding hundreds of GBps, which is several orders of magnitude higher than NIC capacity (*e.g.*, 100 Gbps). Consequently, after the computation phase, each host accumulates a substantial volume of intermediate data, necessitating a bursty transmission and a subsequent reception between racks (or hosts) during the communication phase. When running over ToR-based racks, this communication process traverses the host NIC twice (for egress and ingress) and the potentially oversubscribed uplink of the ToR twice. Therefore, enhancing both the NIC capacity and the capacity of the oversubscribed uplinks of ToRs is crucial for communication efficiency and application performance.

2.2 Existing Approaches

Topology Reconfigurability. One line of prior works [12–14] adds extra bandwidth between the most intensively communicating racks with extra cables, lasers, or antennas to relieve the bottleneck at the uplinks of ToRs, which is usually oversubscribed. Another line does not add bandwidth [12], but builds a flexible network that changes network topology to adapt dynamic workload communication patterns. The efficiency of both approaches highly depends on the predictability of traffic patterns, while it is challenging to identify them due to the *bursty* nature of host- and rack-level outbound traffic (see §2.1). The measurements in production clusters also reveal the presence of a large number of short-lived heavy hitters [5], that is, there is no more than a 50% chance that host- and rack-level heavy hitters will persist in the next 100*ms*.

Brute-force hardware upgrade. To relieve the potential bottleneck at the host's egress bandwidth, GRIN [40] leverages multi-port NICs to expand the host egress capacity, but the improvement is constrained by the fact that the number of additional paths is usually limited. Production GPU clusters like EFLOPS [11], vClos [9], and HPN [10] bound each GPU to a single NIC (400 Gbps) to catch the pace with the hardware throughput. They also leverage multi-homed topology to provide a full bisectional network core. However, they exacerbate the link (and NIC) underutilization, given their utilization is already low [5, 41–43]: as measured, 99% of all links are less than 10% loaded. Moreover, this imposes an escalated "scale tax" – power consumption, hardware and operational expenses [44, 45].

Job placement and scheduling. To reduce inter-rack traffic contention, existing cluster schedulers optimize the placement of jobs close to the input data [15, 16] that is spread over the cluster randomly in a distributed file system (*e.g.*, HDFS [46]). Others schedule communication based on the predictable features of tasks [17–20], *e.g.*, the volume of data, start/end time, and source and destination of the communication. However, due to the unpredictability of job traffic as explained in §2.1, job schedulers cannot completely avoid cross-rack traffic contention.

Takeaway. Existing reconfigurable topologies and job schedulers cannot predict the pattern and volume accurately due to the bursty nature of across-rack traffic of distributed workloads, which persists with any frequency [5]. Upgrading switching hardware to meet the peak bandwidth degrades the network utilization. The goal of this work is to seek a cost-effective rack-level architectural design that efficiently handles the inter-rack traffic without extra hardware for more bandwidth or reconfiguration complexity.

2.3 Insights on NIC utilization

By analyzing the same packet-level dataset from [5] and conducting real experiments with data-intensive applications, we observed that *the bandwidth utilization of existing NICs within*

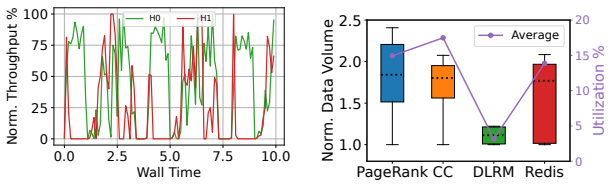
a rack remains low – the insight motivates and carries important implications for the design of DRack. Specifically, we count the number of hosts sending and receiving packet samples within a rack every 1s, respectively, and averaged it over all racks. Figure 2(a) shows over 90% of hosts (NICs) are not sending or receiving in 1s, *i.e.*, remain underutilized, for both Hadoop and Frontend clusters. This observation aligns with prior studies [5,21,41,43], The imbalanced NIC utilization is common and can be attributed to the two factors.

Application semantics. Although all hosts schedule their NICs during the communication stage, the data volume processed by these NICs vary largely due to these factors:

❶ *Compute Irregularity:* Computation tasks are often unevenly distributed across hosts, leading to different computation times [47]. This irregularity results in hosts communicating at different times and NICs handling disparate intermediate data volumes, which are unpredictable and cause imbalanced utilization of NICs. For example, when running graph processing in parallel, each host processes subgraphs with different properties on vertices and edges. This requires varying computation times and produces different amounts of data for synchronization. Moreover, every subgraph evolves iteratively, increasing the unpredictability of communication and computation times. As shown in Figure 3(a) and Figure 3(b), we run a PageRank and CC job over 8 servers and validate data volume and the start time of communication varies significantly. ❷ *Data skewness*: The distribution of data between hosts can be uneven, causing certain NICs to process a disproportionate amount of data compared to others. For example, in DLRM training, input samples of a batch for hosts exhibit heterogeneous access locality on requested embeddings [48–50]. This leads to hosts fetching varying numbers of distinct embeddings from other hosts' embedding tables, and resulting in the uneven generation of gradient data for the embedding layer. In Figure 3(b), the data volume varies substantially for all applications running over 8 hosts, leading to an average NIC utilization less than 20%. We summarize more examples in the Appendix (§A.2), highlighting the generalizability of low NIC utilization across various data-intensive applications.

Resource fragmentation. In datacenters, multiple jobs run concurrently. To schedule them, resource fragmentation is inevitable, as each job has its own resource requirements. If a job can be accommodated by a single host, *i.e.*, a non-distributed job, the NIC of that host remains underutilized. Additionally, BSP paradigm mandates that the NICs can be idle during computation. As observed in production clusters, 20%-45% of machines run non-distributed jobs [21], resulting in underutilized NICs. These NICs could potentially be leveraged by throughput-intensive distributed applications to enhance NIC utilization and performance.

Implications. Given that inter-rack traffic is predominant (§2.1) and yet NIC utilization remains low and imbalanced, if the underutilized NICs equipped at the hosts were controllable and accessible by a different host within the same rack, inter-rack communication can be accelerated, while the burstiness of the traffic can be alleviated by spreading the traffic between more NICs and paths. Moreover, resource fragmentation due to job scheduler can be reduced if NICs can be scheduled independently, without the need to schedule additional computing resources on the same host.



(a) Sampling 2 host's egress throughput running PageRank.

(b) Sent data volume difference between hosts in communication stage.

Figure 3: The imbalanced NIC utilization. (testbed see § 7.1)

2.4 The Benefits of DRack

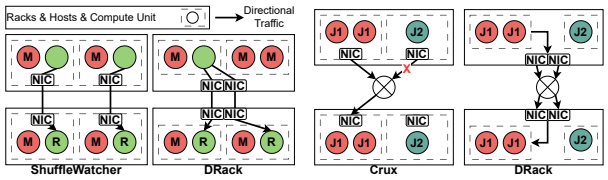
DRack design principle. Driven by the implications of large inter-rack traffic demand and the underutilization of rack NIC ports, we propose a new rack architectural concept – Disaggregated Rack (DRack).

At its core, DRack disaggregates NICs from host boundaries, consolidating them into a rack-level NIC pool accessible by any host to transfer its inter-rack traffic. Maximizing the utilization of the NIC pool's capacity is challenging, as its capacity can exceed the bandwidth of the PCIe links at the destined hosts or the total memory bandwidth of their local memory. For example, in many-to-one across-rack traffic, a NIC pool of 16x200 Gbps NICs (with 16 hosts) can achieve a receiving throughput of 3200 Gbps, which far exceeds the 500 Gbps capacity of a PCIe5x16 link on the destination host. When performing direct memory access (DMA) from/to all NICs, both sending and receiving operations can ultimately be limited by the host's PCIe link or local memory bandwidth.

Thus, the second principle of DRack is to disaggregate system-integrated memories from all host boundaries, forming a rack-level memory pool with an aggregated bandwidth exceeding the capacity of the NIC pool. Hosts can allocate descriptors with referenced address locations spanning all hosts' memory devices, facilitating distributive DMA writing of inter-rack traffic, received from the NIC pool, across all memory devices, not only the local memory of the destined host. Similarly, the NIC pool performs DMA reads from multiple memory devices to achieve full bandwidth transmission.

Last, DRack allows hosts' processors (*e.g.*, CPU) to access the memory pool using memory semantics, This enables them to directly *load* and *store* of data during computation stage, eliminating the need to initially move data to their local memory via DMA.

User cases and benefits. DRack addresses the bottlenecks in inter-rack communication, without the need of additional bandwidth hardware and reconfigurability. Second, DRack is



(a) DRack saves more cores. (b) DRack reduce comm. time.

Figure 4: Workflows of job schedulers on DRack.

not only orthogonal to existing job scheduling and placement strategies, but also unleashes their performance gains.

❶ *Mitigating inter-rack communication bottlenecks.* **1) NIC egress bottleneck**: Recall, during the computation stage, a substantial volume of intermediate data is produced, resulting in a throughput higher than the NIC capacity by several orders. To minimize the idle time of valuable computing resources, DRack allow a host to use underutilized NICs within the same rack to boost its egress capacity like in Figure 1(b), the host then can control multiple NICs to send packets concurrently by passing descriptors to them, thereby mitigating the gap between the compute hardware throughput and the local NIC's egress capacity. **2) Oversubscribed network core**: The NIC pool completes its transmission, as shown in Figure 1(b), acting as the uplinks of tier 2 (ToR), in contrast to Figure 1(a). This is because DRack shifts rack NICs and the above topology components, like ToR switches and aggregated switches, up one tier compared to original ToR-based architecture. The ToR switches effectively take on the role of aggregated switches, while the original aggregated switches can either be eliminated or repurposed to serve as core switches. By appropriately wiring between the NICs and ToR switches in Figure 1(b), DRack achieves a fully bi-sectional bandwidth between any pair of racks, unlike the oversubscribed network core found in ToR-based rack architecture, and does so without the need for additional switching hardware. **3) NIC ingress bottleneck.** Data is received by the NIC pool of the destined rack, as shown in Figure 1(b), and DMA to all rack host' memory devices to fully exploit the NIC pool capacity. In ToR-based racks, incast loss at the ToR switch's egress port can occur due to possible many-to-one traffic and limited capacity of the host's NIC, as the data is only written to a single host.

❷ *User cases.* We show why DRack is orthogonal to existing job scheduler, and can improve them with the following examples: **1) MapReduce jobs schedulers**: We validate how DRack improves ShuffleWatcher [27]. Figure 4(a) shows, upon the mappers of a job running at 4 hosts are done, ShuffleWatcher schedules cross-rack network shuffling with the reducers, This requires each host to schedule the NIC and a compute unit, P, for fetching mappers' output from HDFS and managing data I/O, totally 4P, if little cross-rack traffic exists (assume accurate traffic monitor). Otherwise, ShuffleWatcher schedules mappers of another job to replaces finished mappers. With DRack, it only needs 1P from a host for all shuffling tasks with reducers, as any host can concurrently access all NICs, while leaving the remaining 3P for mappers, thereby enhancing resource efficiency. **2) BSP-based job schedulers**: We validate how DRack improves Crux [26]. Figure 4(b) shows that Crux prioritizes job 1 to use network core bandwidth without contending with other jobs on the same uplink, as it allocates more compute units (4P) than the others (2P). With DRack, job 1 can use 2 NICs (1 from the other host) to send cross-rack traffic, further reducing communication time and minimizing idle time of computing resources.

We validate the DRack benefit in a large scale by evaluating the above two user cases with real experiments (§7.3).

2.5 Realizing DRack

To build DRack, we explore the following techniques and show their suitability for DRack's design.

CXL. The Compute Express Link (CXL) standard [25] enables memory and device pooling, which is a great fit for realizing DRack's memory pool and NIC pool. DRack leverage the fabric topology in CXL 3.0 that supports data-sharing and switch-based pooling: a physical CXL switch can be virtualized into multiple virtual CXL switches (VCS). Each host sees a tree-based virtual hierarchy built with a separate VCS, which handles address remapping for accessing the devices attached in the switch. This maps devices, including memory devices (*e.g.*, local memory and newly-added remote memory in CXL switches) and IO devices (e.g., NICs), along with hosts, into a unified logical address space. CXL memory semantics (CXL.mem) enables hosts to load and store the memory pool directly during the computation stage, unlike prior works using PCIe interconnects for device pooling, which requires moving data to the host's local memory via DMA prior to computation [22–24]. Clearly, CXL is well-suited for DRack design in §2.4. Importantly, DRack only uses CXL hardware (*e.g.*, a CXL switch) without modifications, simplifying future deployment.

SR-IOV. A PCIe device in a PCIe slot (*e.g.*, PCIe NIC) can function as one or multiple physical functions (PFs). SR-IOV (Single Root I/O Virtualization) [51] is a technology that enables a PF to be virtualized into multiple virtual functions (VFs). Thus, each virtual NIC (vNIC) has a copy of dataplane and a portion of the configuration space (IO space), *i.e.*, Tx/Rx queue heads and VLAN QoS settings for a vNIC. This allows different hosts to use the NIC via a vNIC. By accessing the vNIC's IO space, we can finely control the traffic to each vNIC, *e.g.*, setting QoS to support priority scheduling. As shown in CXL specification [25, 52], CXL.io, one of CXL protocols, maintains backward compatibility with PCIe standard. This ensures connectivity with conventional PCIe devices including IO devices such as NICs, thereby inherently supporting SR-IOV. DRack uses CXL.io for its NICs to DMA read/write the memory pool, as well as other functions such as device discovery, configuration, initialization, *etc.*.

Memory Interleaving. Memory interleaving distributes

memory requests by partitioning the physical address space among multiple memory devices at uniform intervals, forming an interleave set (IS). A host can possess multiple ISs. For example, the Intel Skylake platform with two CPU sockets, configures a unique IS for each socket by default, containing the local memory devices of the respective socket. CXL defines the smallest continuous address partition (interleave granularity, IG) ranging from 256B to 16 KB.

Challenges and Solutions. Although the above techniques pave a promising substrate for DRack, there are challenges to overcome to ensure DRack's efficiency and practicality.

C1: CXL accesses exhibits a large latency. Hosts access to remote memory devices within the pool (*e.g.*, the local memory of other hosts) with CXL.mem load and store, which exhibit at least 2.7x latency than accessing local memory of the same NUMA node [53]. Second, CXL.mem load and store are synchronous, cacheline-based with limited concurrency (*e.g.*, 64 at most [54]), resulting in bandwidth inefficiency compared to DMA. We introduce a cache device installed at each host's CXL port – *DRAM cache*, so that remote data can be cached and even prefetched to hide the latency effectively by exploiting the data locality during the computing and communication stages. (see §5.2).

C2: Compatibility. DCN applications are built on the socket and RDMA verbs programming models, which rely on the TCP/IP and RDMA stack for communication, respectively. When hosts and devices within a rack are interconnected with CXL, existing applications must use CXL memory semantics for intra-rack communication. To maintain compatibility with existing host stacks and applications, we introduce a kernel module that seamlessly translates socket system calls (`SEND` and `RECV`), into pass-by-reference semantics (CXL.mem-based load and store) (see §5.1).

3 DRack Overview

DRack hosts support CXL, as shown in Figure 5, there is a CXL port connected to the root complex of the CPU, which is used to access the NIC pool's IO space and memory pool via CXL interconnect. The NIC pool comprises conventional PCIe NICs, compatible with CXL. Besides existing NICs the NIC pool can flexibly be added more NICs to enhance capacity. Besides hosts' local memory, the memory pool can also be expanded with additional remote memory devices attached in the CXL interconnects. A segment of every host's local memory and possible remote memory devices constitute an interleave set of the memory pool (IS0), saving user data. The remaining local memory becomes another interleave set, storing the data structures (*e.g.*, reference queues) frequently accessed by the host (§4.2). The memory pool and NIC pool's IO space are mapped into a unified address space so that any rack host can allocate, deallocate, and access them. The CXL port of a host is connected to a cache device, which caches hot remote data in the on-die DRAM components [55–57].

Workflow. In BSP-based job, each host has generated intermediate data with its data buffers spanning all memory in IS0, which is being synchronized with other hosts during the communication stage. For intra-rack communication (§4.3), where both sending and receiving hosts are within the same rack, the sender's kernel runtime (§5.1) translates the original `SEND` into CXL.mem store, writing references (packet pointers) into a shared queue (in IS1) accessible by the receiver. The receiver's runtime translates the original `RECV` into a CXL.mem load, loading the references and validating them through the TCP stack. The TCP stack validation involves sequential loads of the header of the referenced packets, potentially accessing IS0 remotely. To accelerate it, the receiver leverages the spatial locality by reading a large block into the DRAM cache (§5.2), and subsequent access to packets will hit the DRAM cache. Once a packet is validated, its payload is ready for application computation. Note that, DRAM cache also caches hot data to accelerate computation by exploring memory access locality.

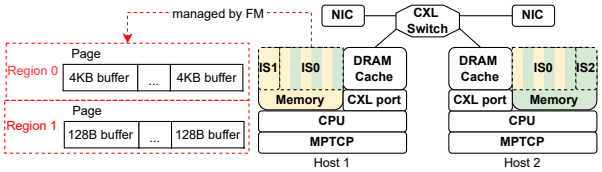


Figure 5: DRack overview and its components

For inter-rack communication, by leveraging SR-IOV, every host in a rack is assigned a virtual NIC (vNIC) for every physical NIC. The sending host splits a flow into multiple subflows with MPTCP. Each subflow is sent by a vNIC. Note that MPTCP is responsible for distributing the data buffers to subflows. The sender notifies every vNIC to trigger DMA by storing the references of the data buffers to the vNIC's descriptor queue via MMIO. Every vNIC DMA reads the data buffer and sends them across racks. At the receiving rack, each vNIC DMA writes the data into its preallocated receive buffers and stores the references of the received packets in the completion queue in IS2. The receive buffers are uniformly distributed across the memory devices of IS0 to utilize the NIC pool's capacity. The host owning the vNIC is interrupted, which triggers its kernel (or CPU) to load those references for TCP validation. Similarly, the DRAM cache prefetches and caches large blocks per remote load issued by the kernel, minimizing remote accesses during both TCP validation and computation.

4 DRack Dataplane

This section presents the architecture of DRack and its integration with the existing DCN, then the communication protocols for intra-rack and inter-rack data, respectively.

4.1 Architecture

DRack changes the traditional multi-tier Clos topology [58] by inserting a CXL interconnect (*e.g.*, CXL fabric) at the edge tier between the hosts and NIC pools, so that every host accesses the NIC pool with its CXL port through the CXL

interconnect. As a result, CXL interconnect becomes ToR tier, and the NIC pool become the uplinks of the ToR tier, while the original ToR switches are repurposed as the aggregation tier. Compared to Clos topology (Figure 1(a)), it enables more paths to transfer inter-rack traffic, and the original aggregation tier can be simply eliminated, and the core of the network remains the same.

Figure 1(b) shows an example of the wiring between 2 racks with each hosting 4 hosts using DRack, and the resultant DCN topology. The wiring between racks also applies to a general case. In a pod comprising m racks and n hosts per rack, Each DRack's NIC pool has n NICs, now there are n uplinks for every pair of racks without uplink oversubscription. Thus, rack 0 has NIC $ID = 0 \sim (n-1)$, rack 1 has NIC $ID = n \sim (2n-1)$, and rack $m-1$ has NIC $ID = (m-1)n \sim (mn-1)$. As explained, the original ToR switches become aggregated switches in DRack, there are m aggregated switches with $ID = 0 \sim (m-1)$. NIC i is linked with the aggregated switch (i mod m), where $0 \leq i < mn$. For example, when $n = 20, m = 10$, NICs 0, 10,..., connect to switch 0, NICs 1, 11,..., connect to switch 1, *etc.*. Similarly, when $n = 20, m = 8$, NICs 0, 8, 16,..., connect to switch 0, NICs 1, 9, 17,..., connect to switch 1, *etc.*. We plot the two examples in Appendix (§A.1).

4.2 Components

CXL interconnect. DRack assumes the existence of the CXL capabilities below based on standards [25] and existing prototypes [59–61]. The CXL Fabric Manager (FM), a software firmware, is responsible for managing the capabilities: 1) setting the mapping between physical addresses and memory devices, 2) configuring the interleave granularity (*e.g.*, 256B to 16KB). These capabilities are the key to efficiently utilizing the NIC pool's capacity by leveraging the total bandwidth of the memory devices. FM can run on a host, a CXL switch, or a device. We run FM on a server in the prototype (§6).

Memory Pool. The memory pool is organized as follows. As shown in Figure 5, the memory pool is mapped to a single Fabric physical Address Space (FAS), which includes multiple ISs. FAS is organized as a series of coarse-grained `Section`s, each with a hugepage size (2 MB). DRack groups a segment of every host's local memory and possible remote memory devices into memory pool IS0, with the smallest IG (256B) to efficiently balance the user data and memory requests across IS0. Each host groups its remaining local memory segment into a separate IS (ISi, where i denotes the host ID). ISi is used to store latency-critical data structures frequently accessed by host i. For example, the descriptor queues of vNICs are accessed by the host that performs multiple load-/store operations to update descriptors, whereas NICs only perform a single DMA read of them in batches. In our prototype, 12GB of local memory is in IS0, while the remaining 4GB is in ISi.

Each IS is organized as a series of 2 MB huge pages. K consecutive `Section`s are grouped into a `region`, K is configurable, and K=512 by default. Note that `region` is used to co-design with DRAM cache (§5.2) to hide the latency of the memory pool. FM is responsible for page allocation/deallocation. The workflow is as follows.

① Every host allocates a `Section` from FM before accessing it to ensure data isolation. A background kernel daemon applies and manages these `Section`s using buddy system: `Section`s from the same `region` are divides into `Buffer`s of same size, and these `Buffer`s are linked as a list.

② Applications at a host apply `Buffer`s from the daemon, which can be used to store application data or as receive buffers for DMA write by NICs. Then, the daemon commits the allocation by removing `Buffer`s from the corresponding list, and creating page table entries (PTE) that map applications' virtual addresses to `Buffer`s' fabric addresses.

③ When an application frees `Buffer`s, it notifies the daemon, which in turn frees the PTEs. To avoid some hosts holding too many free `Section`s, The daemon periodically deallocates excess free `Section`s to FM.

NIC pool. Figure 6 shows the structure and metadata of NIC pool. To share the NIC pool between rack hosts, every SRIOV-capable NIC is virtualized into N virtual NICs (vNIC), with each assigned to a host. For example, Intel 82599 NIC can support up to 64 vNICs [62]. Thus, a host can be assigned M vNICs and interacts with each vNIC using standard PCIe I/O primitives: host processors configure each vNIC using memory-mapped I/O (MMIO), vNICs access the host's data in the memory pool using DMA. However, the vNIC configuration is different by incorporating the memory pool: to interface with a vNIC, host i allocates a dedicated queue pair (`virt_queue`) in its ISi, the local memory segment for storing transmission and reception descriptors. For the receive (RX) path, host i allocates RX buffers uniformly across IS0, minimizing contention at a single memory device and posts RX descriptors to the RX `virt_queue` prior to actual DMA writes by vNIC. Similarly, for the transmit (TX) path, the host allocates TX buffers uniformly across the IS0, populates the TX descriptors with references to consecutive buffers' addresses, and stores them in the TX `virt_queue`. This maximizes the NIC pool throughput when the vNIC performs DMA reads during TX. Each host always uses all NICs in the pool for TX and RX, and balances loading between NICs with a multi-path control (see §4.3).

4.3 Communication

Intra-rack Communication. DRack's intra-rack communication is based on CXL.mem load and store, a pass-by-reference semantics, which reduces the communication time by eliminating data copies in ToR-centric racks. We assume each host has a unique ID (*e.g.*, IP). The communication involves transferring a reference to data in the memory pool from host ID_{src} to host ID_{dst}, so that host ID_{dst} can load/store it. To transfer a reference, ID_{src}'s processor stores a descriptor in ID_{dst}'s `ref_queue`, with each entry containing ID_{dst}, ID_{src},

a reference, and the length it references. A ID_{dst} allocates a separate `ref_queue` for every ID_{src}, denoted by `ref_queue` [ID_{src}], to separate descriptors from different hosts, thus each host needs M-1 `ref_queues` for a rack size of M. Assuming the application has already updated data `Buffers` using load/-store during computation. Figure 6 summarizes the workflow of an intra-rack communication between ID_{src} and ID_{dst}.

① ID_{src} stores the descriptor into the tail entry of ID_{dst}'s `ref_queue` [ID_{src}], where each queue is implemented as a circular buffer. Each queue is associated with context registers mapped to the FAS, such as tail pointer, and ID_{src} stores to these registers will trigger a ID_{dst}'s interrupt.

② After receiving an interrupt, ID_{dst}'s runtime loads the head entry of `ref_queue` [ID_{src}], and invokes TCP stack and application to validate flow reliability and compute with the references sequentially. Note that DRAM cache fetches and cache data from IS0 with a large block size (*i.e.*128B to 4KB) to reduce future remote access (§5.2). The pass-by-reference transfer is realized in Linux and transparent to applications with socket programming model (§5.1).

Inter-rack communication. DRack relies on PCIe I/O primitives (MMIO and DMA) to interact with the NIC pool for inter-rack communication. This design is simple and efficient, as it does not require new hardware support compared to previous works [63–65]. To schedule a data stream between NICs, every host scatters it between all vNICs on a packet basis, so that the NIC pool is utilized fairly for all hosts. However, it may lead to out-of-order packet arrivals, thus underutilizing the NIC pool due to the core network path asymmetry [66–68]. To address it, we exploit MPTCP in existing kernel [69, 70], which can open multiple subflows with each bounding to a vNIC. The receiving host resequences subflows and provides network feedback to the sender, which rebalances the load based on path conditions [69,71]. Figure 6 shows the inter-rack communication workflow.

③ When a ID_{src} has the TX `Buffers` and TX `virt_queues` ready, ID_{src}'s processor rings the doorbells of all vNICs via CXL stores to the MMIO registers. All DMA engines of the vNIC then fetch data from the TX `Buffers` based on descriptors in their corresponding TX `virt_queue`, and perform packetizing. Data `Buffers` are interleaved in the memory pool to use the NIC pool at full capacity, and the descriptors distributed to those TX `virt_queues` are decided by MPTCP.

④ The packet stream intended for ID_{dst} should be received by all vNICs of ID_{dst} uniformly. Each vNIC DMA engine directly writes the payload to the RX `Buffers` specified by the descriptors in the RX `virt_queue`. The ID_{dst} is interrupted by the vNIC via the message signaled interrupts mechanism of PCIe as soon as the DMA engine finishes a batch of the DMA writes. The ID_{dst}'s TCP stack loads the references from all RX `virt_queues` with CXL.mem, and validates the reliability of each subflow by loading the referenced packets' headers. After validation, the application loads and stores `Buffers` during computations. The DRAM cache caches previously accessed blocks and is checked before every load to IS0.

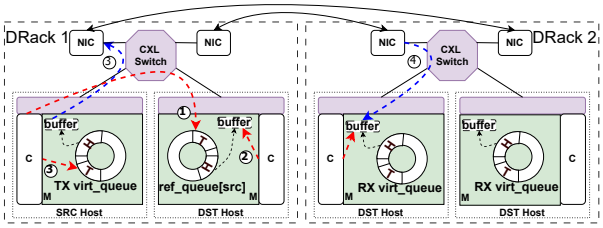


Figure 6: The inter-rack(3,4) and intra-rack(1,2) communication from the SRC host to the DST host. Red and blue lines are memory load/store and DMA read/write, respectively.

Discussion. DRack employs MPTCP to evenly distribute subflows between vNICs. However, this can incur performance overhead when the number of NICs increases. This is because the number of subflows increases with the number of pooled NICs. Each subflow requires separate interrupts, MMIO operations, and kernel processing of subflows merging. However, CPU cores (typically exceeding pooled NIC counts) remain underutilized during bulk data synchronization/shuffling, enabling sufficient cores for NIC pool operations.

5 System Support

5.1 Software Runtime

To enable socket-based applications to use pass-by-reference transparently, a driver below TCP/IP stack acts as an indirection to orchestrate inter-/intra-rack communication with pass-by-reference. The TCP/IP stack operates on the socket buffer object (`sk_buff`), which includes a `data` field that contains the reference to the actual data. The *data* can be headers or payloads of packets, located in the memory pool. For performance consideration, the TCP/IP stack uses the write-around cache policy for `Buffers` so that data is directly written/updated in memory without bringing them to the cache first.

We show a runtime workflow with an example. ① A socket-based application of a host ID_{src} allocates the `Buffers` in memory pool and populates them with application data, before calling the `send` to initiate communication. ② The TCP stack passes a scatter-gather list of `sk_buffs` wrapped with those `Buffers`' references to the driver by calling `ndo_tx`, where `ndo` is the network device operators exposed to the TCP stack by the driver. The first element in the list is a pointer to a kernel `Buffer` storing the packet's header (also in memory pool), and the following elements are pointers to application `Buffers`. ③ The driver stores the references of `sk_buffs` to the intended `virt_queue` of a vNIC (mainstream NICs support scatter-gather I/O) or a `ref_queue` for inter- or intra-rack communication, respectively. For inter-rack communication, the vNIC traverses the list and DMAs data without local-to-pool copies.

ID_{dst} is interrupted to read `virt_queues` or `ref_queue` whenever a batch of new references is available, which is handled by its driver. To enable TCP stack to load the referenced data `Buffers` in the memory pool, ID_{dst} wraps the references

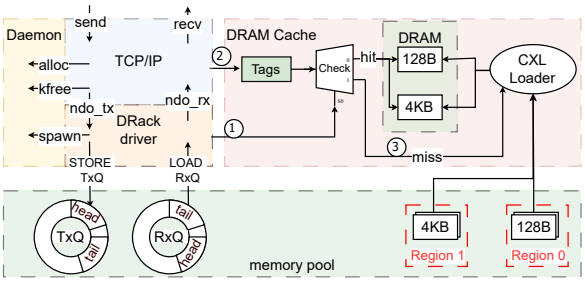


Figure 7: DRAM cache's architecture and working flow.

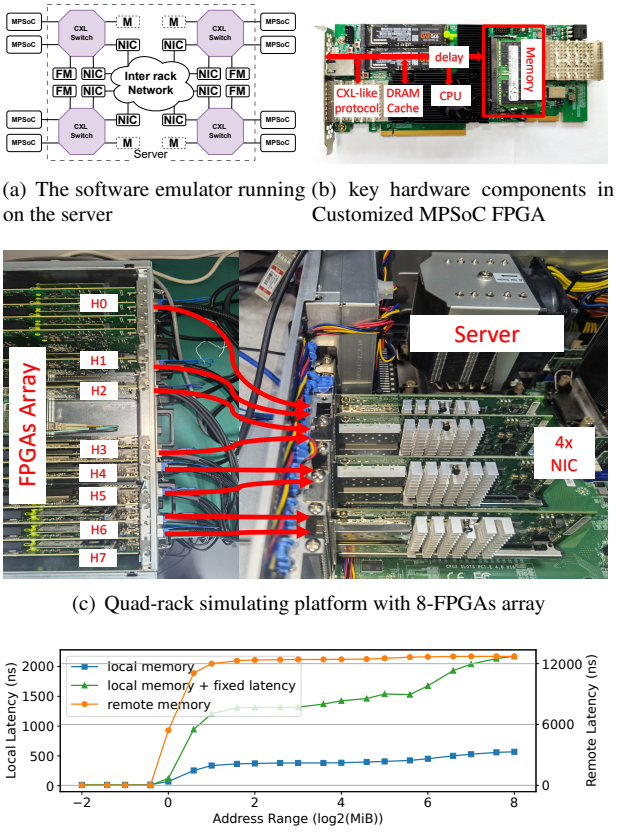
into `sk_buffs` before passing it up to the TCP stack, still no data copy involved. When the TCP stack no longer uses those `Buffers`, it hands the `Buffer` to the daemon for deallocation.

Discussion. Cache coherence is needed if ID_{src} and ID_{dst} use memory sharing for intra-rack communication. However, DRack focuses on the case where applications use message-passing for large data synchronization, simultaneous data accessing between ID_{src} and ID_{dst} is unnecessary, thus no sharing issues. If ID_{src} stores new data to the location of the previously sent `Buffer` after calling the `send` API, ID_{src}'s TCP stack releases ownership of the sending `Buffer`, allocates new `Buffers` from the memory pool, and updates page tables for subsequent stores.

5.2 CXL-attached DRAM Cache.

By analyzing the TCP stack, we summarize the three key points to guide the DRAM cache design. **1) Variable time interval between accessing headers and payloads.** The runtime processes packet headers and data during the bottom half of the interrupt, and the `recv` syscall, respectively. Therefore, there will be a time interval between the two, where the interval length is determined by the utilization rate of the CPU, and the data can be evicted out of the cache. **2) Buffer release function is explicitly called.** the runtime will flush the `Buffer` out of the cache once it calls release function explicitly, **3) Variable packet granularity.** The packet size may be thousands of or dozens of bytes, like TCP SYN and ACK. Caching data with coarse granularity leads to a low cache utilization rate and prolongs the latency, thus different caching granularities should be adopted.

As shown in Figure 7, we add a multi-way DRAM cache to the system. It has a larger capacity compared to CPU caches, which is sufficient to tackle the potential false evictions caused by conflicts during variable intervals. The metadata (`Tags`) is stored separately in the on-chip memory, so that one read can get the whole set's metadata. This cache space can be divided into n parts, each of which uses a different cache granularity (*e.g.*, 128B and 4KB in Figure 7), and the capacity of each part is different, but the total number of lines is the same. The usage of DRAM cache can be summarized as follows. ① The driver configures the FAS of a `region` – a consecutive number of `Sections`, whose `Buffers` will be cached in one part of DRAM cache, and the caching granularity, *i.e.*, `Buffer` size. For the first access to a `Buffer` in the `Region`②, a miss is triggered and the `Buffer` is fetched from the memory pool via CXL.io ③. Then, the DRAM cache fills valid metadata and evicts the victim `Buffer`. The following accesses to the `Buffer`② will hit the local DRAM cache. When the runtime decides to free `Buffers`, it will explicitly called the release function that flushes the DRAM cache.



(a) The software emulator running on the server

(b) key hardware components in Customized MPSoC FPGA

(c) Quad-rack simulating platform with 8-FPGAs array

(d) The memory access latency with/without added delay

Figure 8: The proof-of-concept prototype to evaluate DRack.

6 Implementation

Given the lack of commercial products supporting CXL 3.0, we have developed a proof-of-concept quad-rack system prototype, as shown in Figure 8(c). The prototype is used to simulate both a quad-rack DRack architecture and a quad-rack ToR-centric architecture as the baseline (§7.1). Each rack is equipped with two customized MPSoC FPGAs [72] serving as hosts. Each FPGA has a quad-core CPU, two memory channels, and four optical fiber ports. The MPSoC exports the CPU memory bus to FPGA logic via HP/HPC ports, which can be used to implement a CXL-like load/store. We connect two FPGAs (in a rack) to a dual-port NIC [62] on a server. We emulate the NIC pool, CXL interconnect in DRack, and ToR switches in the baseline using a DPDK-based software network emulator on the server [73].

CXL-like protocol layer. The key to emulating CXL fabric in CXL 3.0 is to externalize loads and stores. This allows applications to access remote memory devices within the

memory pool transparently, enabling the seamless execution of legacy applications. To this end, we leveraged DoCE [32], a hardware protocol stack as the basis of our CXL-like protocol layer. DoCE directly encapsulates all AMBA AXI on-chip interconnect signals within an Ethernet frame which can be delivered via standard Ethernet infrastructure. Instead of packing AXI signals into Ethernet packets with DoCE, we augmented DoCE with a CXL-like protocol layer and implemented them as a hardware module in the MPSoC FPGA (*i.e.*, CXL-DoCE). As shown in Figure 8(b), it transforms AXI signals into relevant CXL transactions, such as MemRd, MemWr, Cmp, and MemData defined in CXL 3.0 specification [25], before encapsulating CXL transactions into Ethernet packets and sending them via an optical fiber port. Similarly, any received Ethernet packet from the port would go through the module, which transforms it into AXI signals to access the local address space if that packet contains CXL-like transactions. Note that an Ethernet packet can encapsulate transactions not limited to CXL-like as long as the module can recognize them, *e.g.*, the module can interrupt the CPU if the Ethernet packet carries an interrupt transaction.

Memory and NIC pool. We use the local memory in all MPSoCs (16 GB) to form a shared memory pool. Any MPSoC can access it with the above CXL-like protocols. Following the design, we divide the local memory of every MPSoC into two parts, 12GB joins an interleave set with 256B granularity, the other 4GB does not interleave with others, so that `virt_queues` and *sk_buf* used can be stored locally. To emulate NIC pool in the server (Figure 8(a)), we use one core to manage a software queue to emulate a vNIC. The server's core can load/store a vNIC's `virt_queue` or DMA read/write data by issuing CXL-DoCE packets via the real NIC. We use inter-processor interrupt (IPI) to forward the packet "across rack" to another CPU core (a vNIC in the other NIC pool).

Example. Here is an example showing the memory pool accessing in the prototype with CXL-like protocol layer. ① In a rack with MPSoC0 and MPSoC1, suppose MPSoC0 HP port exports CPU load/store instructions as AXI signals, an address decoder implemented in the FPGA uses the 8th bit of the physical address (IG=256B) to decide the target memory device. If bit 8 is 0, memory requests target MPSoC0's memory; otherwise, they target MPSoC1's one. ② Recall, to issue requests to the MPSoC1, MPSoC0 leverages CXL-DoCE to transform AXI signals into CXL transactions, *e.g.*, MemRd, and prepend them with an Ethernet header (containing MAC addresses of both endpoints). Then, it sends them via on optical fiber port. ③ As shown in Figure 8(b), the server (network emulator) receives these Ethernet packets via optical fiber links at its physical NICs, and routes them directly to MPSoC1 based on MACs. ④ MPSoC1 receives Ethernet packets from the optical fiber port, and transforms them into AXI signals with CXL-DoCE, to access the local address space. Appendix §A.3 shows an example of NIC pool accesses.

Compared to existing emulation approaches. Due to scarcity of CXL hardware, prior works have employed 3 emulation approaches. ① NUMA nodes. CPUs load/store remote node's memory is emulated as accessing CXL memory [28, 31, 74]. However, this cannot adjust CXL memory latency, and its scalability is limited by the node count. ② Software simulator [75, 76], like Gem5 in full-system mode [77], is thousands of times slower than real prototypes, unable to simulate full application execution. ③ FPGA-based implementation [53, 60, 78], has the flexibility to implement CXL protocols [60, 72], and run real applications transparently. However, FPGA has a lower operating frequency compared to ASICs [53]. ④ Our approach. We utilize the flexibility of MPSoC-FPGAs to realize CXL-like protocols for resource pooling and running real applications. To run distributed applications at scale and test them in various settings, we use software-based emulation for flexible configurations of NIC pool, CXL interconnects, and inter-rack network topology. To rigorously validate DRack performance gain with low frequency in FPGA, we focus on accurately replicating local/remote memory access latency ratios rather than absolute metrics, which aligns with prior FPGA-based emulation [60]. The ratio configurations are explained in §7.1.

Table 1: Key parameters of the two prototypes.

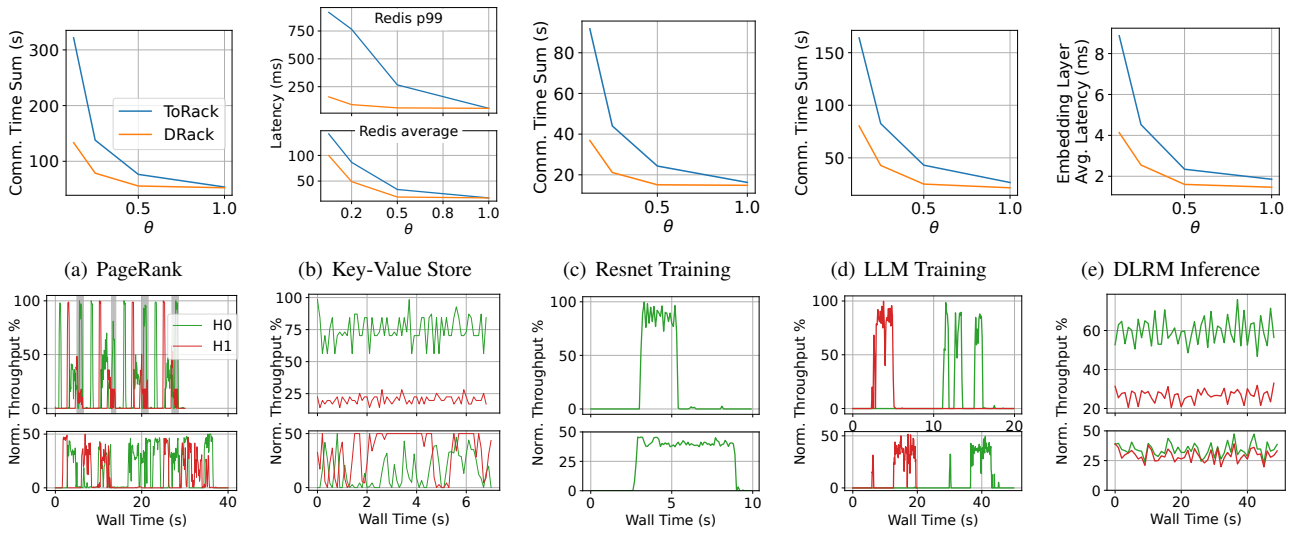
Parameter	Value
`DRack`	
Memory Pool	2×16 GB; remote 13 μs; local 2.2 μs
NIC pool	$2 \times B$ *Gbps* (2 NICs)
Host	1 GB DRAM Cache; 1 CXL port;
`ToRack`	
ToR Switch	2B Gbps Uplink (no oversubscription) 256 KB headroom per port [79–81]
Host	B Gbps NIC; 16GB memory
Common	
Inter-rack Network RTT	60 μs [82]
CPU	4-core ARM A53 CPU; 1.2 GHz
TCP/IP	MTU 4 KB; XPS, RPS enabled

7 Evaluations

We first evaluated DRack's performance gain over the conventional ToR-based rack using the proof-of-concept prototype with both real-world applications (§7.2), and microbenchmarks (see Appendix §A.4). Second, we show DRack can improve existing job schedulers (§7.3). Last, we breakdown the performance contribution of every key components (§7.4).

7.1 Experiments Setup

We evaluate the quad rack DRack proof-of-concept prototype (§6), and use the same implementation to build baseline systems for comparison. By default, each rack has two hosts for both. We use `ToRack` to denote the baseline system—quad ToR-based racks architecture. Table 1 summarizes the parameters and configuration in experiments. The memory pool of DRack is composed of the local memory of MPSoCs. For



(a) PageRank (b) Key-Value Store (c) Resnet Training (d) LLM Training (e) DLRM Inference

(f) PageRank (g) Key-Value Store (h) Resnet Training (i) LLM Training (j) DLRM Inference

Figure 9: (a)-(e) DRack's performance gain comapred to ToRack with the increase on every NIC capacity (θ from 0 to 1). (f)-(j) Compared to ToRack (bottom), hosts in DRack (top) can utilize the NIC pool capacity to reduce the communication time.

Table 2: Real-world Applications used for DRack evaluation.

Application	Dataset
GeminiGraph [83]	LiveJournal [84]
Redis Cluster [85] and Memtier_benchmark [86]	Uniform Distribution
ResNet18 [87] Training	CIFAR-10 dataset [88]
TinyStories-33M [89] Training	TinyStories
MapReduce [90] WordCount	Uniform Distribution
DLRM [91] Inference	Criteo_dataset [92]

ToRack, we ported an example project [93] to our MPSoC FPGA, where the CPU uses one optical fiber port as a NIC. We set Ethernet link latency in network emulator 60μs [82].

Latency alignments. To align the latency ratio (R) between accessing local (D_{local}) and remote memory (D_{cxl}) with CXL, as shown in Figure 8(b), we add a hardware module on MPSoC FPGAs to add configurable cycles (λ) to local memory accessing from CPUs, such that $R = \frac{D_{cxl}}{D_{local}+\lambda/cpu_freq}$. According to the theoretical latency specified in CXL-related standards [25, 94], accessing the remote memory through a CXL switch is expected to add 170 to 270 ns to the local memory latency. Assuming that the local memory latency is between 50 and 100 ns, the resulting ratio should fall within 2.7 to 6.4. The real measurements of the two latencies align with this range [53]. To illustrate the minimal performance gain achieved by DRack, we configured the latency ratio $R = 6.4$, thus adopting the maximum remote latency. In Figure 8(d), we increase the size of the data array running pointer chasing with Lmbench [95], until accessing it triggers a cache miss, *i.e.*, when the data array size is 256 MB and use the resulted D_{local} and D_{cxl} to compute $\lambda = 460$.

Throughput alignments. The capacity of the emulated NIC pool depends on the maximum achievable throughput of the hosts in the rack, governed by the capabilities of the CPU in the MPSoC. This is because the FPGA NIC (*i.e.*, optical fiber port) is not the bottleneck [93], *i.e.*, the maximum achievable throughput of an MPSoC CPU, denoted by C, cannot saturate the NIC. Thus, the emulated NIC capacity of a host can be set to C at most, as the MPSoC CPU cannot utilize the capacity exceeding C. To emulate the NIC egress/ingress bottleneck in the communication stage, we introduce a reducing factor θ, thus the NIC capacity is adjusted as $\theta \times C$. As the NIC pool is built from existing NICs of the hosts in a rack, to have a fair comparison, the equivalent DRack's NIC capacity is $B = \theta \times 2C$ for 2-host rack. We measure C by sending raw packets to the optical fiber port with DPDK.

7.2 Real-world Applications

We validate the performance gain of DRack compared to ToRack with real applications with various communication properties, as shown in Table 2.

7.2.1 Throughput-sensitive Applications

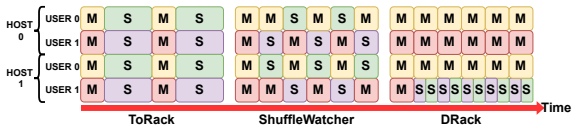
Graph Processing. We run several graph workloads in a distributed graph engine, Gemini [83], over DRack and ToRack. To execute it, we divided the input datasets among the hosts of 4 racks (8 hosts), each subgraph dataset is incrementally fed to a host's graph engine in batches. For every batch or iteration, hosts run different analytics over the input subgraph and update it iteratively. As the subgraph processed by every host is different, the amount of data for graph synchronization for every host can be very different, thus necessitating different bandwidths. We summarize the communication time of graph synchronization for PageRank, one of the graph workloads, in Figure 9(a). DRack reduces the communication time by an average of 58.5% in the worst case (θ=0.125), and 32.8% in all cases. When the host throughput

is eventually limited by the maximum achievable throughput or MPSoC CPU capability (θ=1), any DRack's host cannot efficiently utilize the NIC pool, thus resulting in a similar performance to ToRack's. To show the root cause of DRack's performance gain, we present network utilization of one rack normalized to the NIC pool, sampled during one run of PageRank in Figure 9(f). Hosts finish iterating and updating their subgraph, *e.g.*, vertices and edges, asynchronously. This is because hosts process subgraphs with varying sizes, leading to different computation times and thus varying amounts of data for communication. Thus, one host may use the NIC pool exclusively during the synchronization stages. Besides, in the time interval with shadowing in Figure 9(f), two hosts can share the NIC pool based on the volume of flows, while the overall utilization is close to 1.

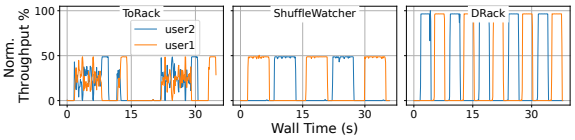
ML Training. We train ResNet18 and TinyStories-33M with all-reduce [96] and parameter server (PS) [97], the two well-known distributed training architectures based on data parallelism. For all-reduce, each host of eight sends its gradient to its right neighbor, while receiving other gradients from its left neighbor during communication stage. For PS, we have 4 hosts as parameter servers, while the others are workers. Every worker sends 1/4 of its gradients to every PS for synchronization and then receives globally-aggregated ones during communication. We summarize the communication time when running ML jobs over DRack and ToRack, respectively, and show the results in Figure 9(c) 9(d). For ResNet training, DRack reduces the communication time by 59.8% and an average of 39.6% in the worst and all cases, respectively. For LLM training, DRack reduces the communication time by an average of 39.9%. This is because 1) for all-reduce, only one host sends/receives data across racks, thus DRack achieves higher inter-rack throughput by allocating the NIC pool capacity to that host, while ToRack's inter-rack throughput is limited by the host's NIC (Figure 9(h)). 2) for PS, Since workers and PSs send and receive data in an interleaved manner – specifically, when workers are sending, PSs are receiving, and vice versa – either can fully utilize the NIC pool over different phases (Figure 9(i)). 3) DRack achieves a high intra-rack throughput with pass-by-reference by removing one data copy compared to ToRack.

7.2.2 Latency-sensitive Applications

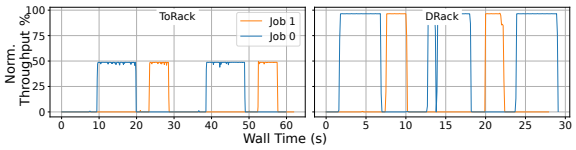
Key-value Store. We organize three racks to form a Redis Cluster and run clients on the remaining hosts to issue Get/Set requests. We set up the experiment as: 1) synthetic workload generated by Memtier_benchmark, 2) a 10:1 Get/Set ratio, and 3) 10 M key-value pairs uniformly distributed across 3 racks, 4) a Gaussian query key distribution. We plot the average and tail latency (*i.e.*, 99th percentile) of the requests in Figure 9(b). Because of the key skewness, the host in ToRack with hot keys can be congested by imbalanced requests, *i.e.*, incast, at its downlink, leading to packet losses and large tail latency. In contrast, DRack reduces p99 and average latency



(a) Workflow of MapReduce tasks.



(b) Throughput of shuffling data.



(c) NIC pool improve comm. throughput.

Figure 10: DRack improves job schedulers' performance.

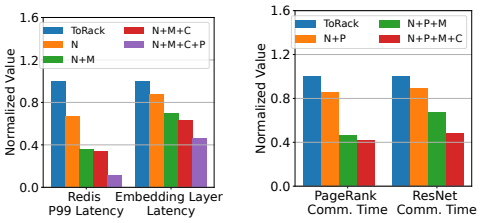
by 62.2% and 29.2%, respectively. This is because a host with hot keys can utilize the NICs of other hosts within the same rack, which have cold keys. The incast requests are then efficiently absorbed by the memory pool without any loss. Subsequently, the host processes these requests using CXL.mem. Similarly, a hot host can send the corresponding values with the NIC pool. The allocated bandwidth from the NIC pool is proportional to the number of requests/flows that the host is serving, while the hot host in ToRack can only use its limited local NIC (Figure 9(g)).

DLRM Inference. We run a deep learning recommendation model (DLRM [91]) inference task that embedding tables are uniformly distributed across 8 hosts. We plot the average latency of embedding layer processing in Figure 9(e). Because of frequent access to hot vectors, the traffic load at each NIC within the same rack is imbalanced in ToRack that network congestion leads to higher latency and reduced throughput. DRack reduces average latency by an average of 37.4% in all cases, as the NIC pool's bandwidth is proportional to the hot vectors the host stored, as shown in Figure 9(j).

7.3 User Cases

To show the performance gain if existing job schedulers use DRack for the job communication stage, we emulate two user cases (§2.4), ShuffleWatcher [27] and Crux [26].

ShuffleWatcher. Figure 10(a) shows that two users of the same priority issue a series of MapReduce jobs, each comprising 2 map and 2 reduce tasks. The map and reduce tasks are placed in racks 0 and 1, respectively. Assuming there are 2 hosts in rack 0, each with 2 cores, each user rents 1 core per host to run a mapper and shuffler, iteratively. The shuffler begins after the mapper finishes, and then sends the mapper's intermediate output to the reducer in rack 1. All mappers generate intermediate data of equal size. In ToRack without any scheduler, the shuffling tasks of the two users send data to



(a) Perf. breakdown on latency-sensitive APPs.

(b) Perf. breakdown on throughput-sensitive APPs

Figure 11: Key designs' impact on DRack's performance.

the reducers simultaneously, increasing the average shuffling time. With ShuffleWatcher, the two users use the host's NIC to shuffle data alternately (Figure 10(b)). Specifically, user 2 delays its shuffling if user 1 occupies the NIC, while scheduling another map task to reuse the CPU left by shuffling, thus reducing the average shuffling time by half. In DRack, a single CPU of a host can use the NIC pool to send each shuffling task, increasing CPU utilization and reducing the average shuffling time. Although DRack takes the same time as ShuffleWatcher to complete the same number of shuffling tasks (10 in Figure 10(a)), DRack frees up more cores for additional map tasks, improving overall throughput by 20.8%.

Crux. We evaluated Crux on DRack with 3 jobs. Each rack has two hosts. Job 0 (ResNet50 training) spans 4 hosts across 4 racks, while jobs 1 and 2 (ResNet18 training) each span 2 hosts across 2 racks. DRack utilizes the priority flow control function in the optical fiber port IP to throttle the throughput of low-priority hosts (or MPSoC). When high-priority hosts are scheduled by Crux, the network emulator sends a pause packet to the low-priority job's host, causing its hardware IP to cease packet transmission. As shown in Figures 10(c), when running Crux in ToRack, the communication time for job 0 is suboptimal, as it cannot leverage the NICs of jobs 1 and 2's hosts, even if they are deprioritized and halted by Crux. In contrast, DRack enables job 0 to leverage the NICs of hosts assigned to lower-priority jobs, thereby reducing communication time by 47.7%. Once job 0's communication stage is completed and job 1 ascends to the highest priority, its communication can be accelerated by 49.5% with the NIC pool, including NICs from job 0's hosts.

7.4 Performance breakdown

We introduce several key components, including the NIC pool, memory pool, DRAM cache, and MPTCP, to build an efficient DRack. To determine the performance contribution of each component, we incrementally add these components one by one, starting from ToRack. We then evaluate the performance gains of each incremental addition using two latency-sensitive and two throughput-sensitive applications.

As shown in Figure 11(a), for latency-critical Redis, by forming a NIC pool with existing NICs from ToRack, when equipped with only the NIC pool (denoted as DRack-N), reduces tail latency by 32.9%. This reduction is attributed to the fact that hot-key requests are received by the NIC pool, distributively between the NICs. They are first buffered in the NIC pool's headroom before being transferred to the host's local memory with the CXL link. Thus, compared to the case in ToRack, where hot-key requests are received solely by the NIC of the host, DRack has twice the headroom (2 NICs) for buffering those requests, significantly reducing packet loss. By augmenting the memory pool to DRack-N (DRack-N+M), the packet loss is further reduced, resulting in 63.9% P99 latency reduction. The DRAM cache (DRack-N+M+C) fails to reduce the latency further because searching and reading key-value pairs is random memory access that lacks locality (DRAM cache hit rate 59.6%). MPTCP reduces the latency further by 65.9% (DRack-N+M+C+P). This is because a bursty flow is fixed to a single NIC, resulting in imbalanced utilization of NICs. By splitting it into multiple subflows, with each directing to a NIC, MPTCP further reduces the burstiness of the flow and flow collision. The observation is similar for DLRM inference as shown in Figure 11(a).

As shown in Figure 11(b), for data-intensive ResNet training and PageRank, by using the NIC pool and MPTCP (DRack-N+P), the communication time is reduced by an average of 12.7%, as cross-rack traffic can be sent via 2 NICs. However, the throughput is limited by the local memory (or CXL link) bandwidth, thus preventing full utilization of the NIC pool. With an increasing memory bandwidth from the memory pool (DRack-N+P+M), the communication time is reduced by an average of 38.1%. By introducing a DRAM cache (DRack-N+P+M+C), the communication time is reduced by 28.6% for ResNet training. This improvement is due to the DRAM cache that exploits the spatial locality of memory accessing in TCP stack and model computation, achieving hit rates exceeding 86.9%. While DRAM cache only reduces PageRank's communication time by 9.9%, its gain is limited by a cache hit rate of only 56.2%, caused by the irregular memory accessing.

8 Conclusion

We present DRack, a novel rack architecture that provides high inter-rack bandwidth and network utilization with a NIC pool, and optimizes its efficiency with several novel designs such as a shared memory pool, DRAM Cache, and MPTCP. We built a quad-rack prototype and validated its performance with both microbenchmarks and real applications.

Acknowledgments

We would like to thank our shepherd, Prof. Kang Chen, and the anonymous reviewers for their valuable feedback in improving the paper. The work was supported in part by the National Key Research and Development Plan of China (No. 2022YFB4500400), National Natural Science Foundation of China (No. 62072439, No. 62302268), the Natural Science Foundation of Shandong Province (No. 2023HWYQ-045, No. ZR2023QF060), and the Taishan Scholar Project of Shandong Province (No. tsqn202312051). Ke Liu is the corresponding author (liuke@ict.ac.cn).

References

[1] G. Malewicz et al. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, 2010.

[2] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, jan 2008.

[3] W. Liu et al. A survey of deep neural network architectures and their applications. *Neurocomputing*, 234:11–26, 2017.

[4] L. G. Valiant. A bridging model for parallel computation. *Commun. ACM*, aug 1990.

[5] A. Roy et al. Inside the social network's (datacenter) network. *SIGCOMM Comput. Commun. Rev.*, 45(4):123–137, August 2015.

[6] Y. Jiang et al. A unified architecture for accelerating distributed DNN training in heterogeneous GPU/CPU clusters. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pp. 463–479, November 2020.

[7] Z. Jia et al. A distributed multi-gpu system for fast graph processing. *Proc. VLDB Endow.*, 11(3):297–310, November 2017.

[8] N. Engelhardt and H. K.-H. So. Gravf: A vertex-centric distributed graph processing framework on fpgas. In *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 1–4, 2016.

[9] X. Han et al. Isolated scheduling for distributed training tasks in gpu clusters, 2023.

[10] K. Qian et al. Alibaba hpn: A data center network for large language model training. In *Proceedings of the ACM SIGCOMM 2024 Conference*, pp. 691–706, 2024.

[11] J. Dong et al. Eflops: Algorithm and system co-design for a high performance distributed training platform. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2020.

[12] W. Wang et al. RDC: Energy-Efficient data center network congestion relief with topological reconfigurability at the edge. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pp. 1267–1288, 2022.

[13] X. Zhou et al. Mirror mirror on the ceiling: flexible wireless links for data centers. *SIGCOMM Comput. Commun. Rev.*, 42(4):443–454, August 2012.

[14] H. Liu et al. Circuit switching under the radar with REACToR. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pp. 1–15, April 2014.

[15] M. Zaharia et al. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, pp. 265–278, 2010.

[16] M. Isard et al. Quincy: fair scheduling for distributed computing clusters. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, pp. 261–276, 2009.

[17] F. R. Dogar et al. Decentralized task-aware scheduling for data center networks. *SIGCOMM Comput. Commun. Rev.*, 44(4):431–442, August 2014.

[18] M. Chowdhury et al. Efficient coflow scheduling with varys. *SIGCOMM Comput. Commun. Rev.*, 44(4):443–454, August 2014.

[19] H. Zhao et al. HiveD: Sharing a GPU cluster for deep learning with guarantees. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pp. 515–532, November 2020.

[20] Y. Zhao et al. Multi-resource interleaving for deep learning training. In *Proceedings of the ACM SIGCOMM 2022 Conference*, pp. 428–440, 2022.

[21] Y. Jiang et al. A unified architecture for accelerating distributed DNN training in heterogeneous GPU/CPU clusters. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, November 2020.

[22] D. Zang et al. Decentralized nic-switching architecture using sr-iov pci express network device. *IEEE Micro*, 2014.

[23] C.-C. Tu et al. Secure i/o device sharing among virtual machines on multiple hosts. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, 2013.

[24] C.-C. Tu et al. Marlin: A memory-based rack area network. In *2014 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, 2014.

[25] D. D. Sharma et al. An introduction to the compute express link (cxl) interconnect. *arXiv preprint arXiv:2306.11227*, 2023.

[26] J. Cao et al. Crux: Gpu-efficient communication scheduling for deep learning training. In *Proceedings*

of the ACM SIGCOMM 2024 Conference, pp. 1–15, 2024.

[27] F. Ahmad et al. ShuffleWatcher: Shuffle-aware scheduling in multi-tenant MapReduce clusters. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pp. 1–13, June 2014.

[28] H. N. Schuh et al. Cc-nic: a cache-coherent interface to the nic. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, pp. 52–68, New York, 2024. ACM.

[29] A. Cho et al. A case for cxl-centric server processors, 2023.

[30] R. Abdullah et al. Salus: Efficient security support for cxl-expanded gpu memory. In *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 1–15, Piscataway, 2024. IEEE.

[31] H. A. Maruf et al. Tpp: Transparent page placement for cxl-enabled tiered-memory. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pp. 742–755, New York, 2023. ACM.

[32] Y. Chang et al. DoCE: Direct extension of on-chip interconnects over converged ethernet for rack-scale memory sharing. In *Proc. Workshop on Emerging Technologies for software-defined and reconfigurable hardware-accelerated Cloud Datacenters (ETCD)*, 2017.

[33] V. Persico et al. Measuring network throughput in the cloud: The case of amazon ec2. *Computer Networks*, 93:408–422, 2015. Cloud Networking and Communications II.

[34] S. Shi et al. Towards scalable distributed training of deep learning on public cloud clusters, 2020.

[35] A. Hadoop. Hadoop: Capacity scheduler, 2019.

[36] R. Panigrahy et al. Heuristics for vector bin packing. *research. microsoft. com*, 2011.

[37] A. Ghodsi et al. Dominant resource fairness: Fair allocation of multiple resource types. In *8th USENIX symposium on networked systems design and implementation (NSDI 11)*, 2011.

[38] Z. Wang et al. Hi-speed dnn training with espresso: Unleashing the full potential of gradient compression with near-optimal usage strategies. In *in European Conference on Computer Systems (EuroSys 23)*, May 2023.

[39] M. Abadi et al. TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems. White paper, Google Research, 2015.

[40] A. Agache et al. Increasing datacenter network utilisation with grin. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, 2015.

[41] T. Benson et al. Network traffic characteristics of data centers in the wild. In *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement*, pp. 267–280, 2010.

[42] T. Benson et al. Understanding data center traffic characteristics. *SIGCOMM Comput. Commun. Rev.*, 40(1):92–99, jan 2010.

[43] C. Delimitrou et al. Echo: Recreating network traffic maps for datacenters with tens of thousands of servers. In *2012 IEEE International Symposium on Workload Characterization (IISWC)*, pp. 14–24, 2012.

[44] H. Ballani et al. Sirius: A flat datacenter network with nanosecond optical switching. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pp. 782–797, 2020.

[45] W. Wang et al. {RDC}:{Energy-Efficient} data center network congestion relief with topological reconfigurability at the edge. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pp. 1267–1288, 2022.

[46] T. White. *Hadoop: The definitive guide*. 2012.

[47] J. E. Gonzalez et al. Graphx: Graph processing in a distributed dataflow framework. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI '14)*, Broomfield, Colorado, October 2014.

[48] A. Guo et al. Software-hardware co-design of heterogeneous smartnic system for recommendation models inference and training. In *Proceedings of the 37th ACM International Conference on Supercomputing*, ICS '23, pp. 336–347, 2023.

[49] D. Mudigere et al. Software-hardware co-design for fast and scalable training of deep learning recommendation models. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ISCA '22, pp. 993–1011, 2022.

[50] M. Adnan et al. Accelerating recommendation system training by leveraging popular choices. *Proc. VLDB Endow.*, 15(1):127–140, September 2021.

[51] Y. Dong et al. High performance network virtualization with sr-iov. *Journal of Parallel and Distributed Computing*, 2012.

[52] CXL Consortium. Compute Express Link Specification Revision 2.0. https://www.computeexpresslink.org/download-the-specification.

[53] Y. Sun et al. Demystifying cxl memory with genuine cxl-ready systems and devices. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 105–121, New York, 2023. ACM.

[54] N. Nassif et al. Sapphire rapids: The next-generation intel xeon scalable processor. In *2022 IEEE International Solid-State Circuits Conference (ISSCC)*, volume 65, pp. 44–46. IEEE, 2022.

[55] I. Calciu et al. Rethinking Software Runtimes for Disaggregated Memory. In *Proc. of ASPLOS*, 2021.

[56] Y. Zhong et al. Unimem: Redesigning disaggregated memory within a unified local-remote memory hierarchy. In *2024 USENIX Annual Technical Conference*, pp. 463–477, Berkeley, 2024. USENIX Association.

[57] X. Zhang et al. Morpheus: An adaptive dram cache with online granularity adjustment for disaggregated memory. In *2023 IEEE 41st International Conference on Computer Design (ICCD)*, pp. 134–141, Piscataway, 2023. IEEE.

[58] A. Singh et al. Jupiter rising: A decade of clos topologies and centralized control in google's datacenter network. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, pp. 183–197, 2015.

[59] C. Wang et al. Cxl over ethernet: A novel fpga-based memory disaggregation design in data centers. In *2023 IEEE 31st Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2023.

[60] D. Gouk et al. Direct access, High-Performance memory disaggregation with DirectCXL. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pp. 287–294, 2022.

[61] M. Zhang et al. Partial failure resilient memory management system for (cxl-based) distributed shared memory. In *Proceedings of the 29th Symposium on Operating Systems Principles*, 2023.

[62] Intel. Intel® 82599eb 10 gigabit ethernet controller. https://ark.intel.com/content/www/us/en/ark/products/32207/intel-82599eb-10-gigabit-ethernet-controller.html.

[63] Z. Wang et al. Rcmp: Reconstructing rdma-based memory disaggregation via cxl. *ACM Transactions on Architecture and Code Optimization*, 21(1):1–26, 2024.

[64] D. Gibson et al. Aquila: A unified, low-latency fabric for datacenter networks. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, April 2022.

[65] W. Lin et al. Supernic: An fpga-based, cloud-oriented smartnic. In *Proceedings of the 2024 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pp. 130–141, 2024.

[66] M. Alizadeh et al. Conga: distributed congestion-aware load balancing for datacenters. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, pp. 503–514, 2014.

[67] H. Zhang et al. Resilient datacenter load balancing in the wild. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pp. 253–266, 2017.

[68] N. Katta et al. Clove: How i learned to stop worrying about the core and love the edge. In *Proceedings of the 15th ACM Workshop on Hot Topics in Networks*, pp. 155–161, 2016.

[69] D. Wischik et al. Design, implementation and evaluation of congestion control for multipath tcp. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation (NSDI '11)*, Berkeley, CA, USA, March 2011.

[70] C. Raiciu et al. How hard can it be? designing and implementing a deployable multipath tcp. In *Proceedings the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, San Jose, CA, April 2012.

[71] A. Ford et al. Rfc 8684: Tcp extensions for multipath operation with multiple addresses. https://www.rfc-editor.org/rfc/rfc8684.html, 2020.

[72] A. Xilinx. Zynq™ ultrascale+™ mpsoc. https://www.xilinx.com/products/silicon-devices/soc/zynq-ultrascale-mpsoc.html.

[73] DPDK. Data plane development kit. https://www.dpdk.org/about/.

[74] H. Li et al. Pond: Cxl-based memory pooling systems for cloud platforms. In *Proceedings of the 28th ACM*

International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, pp. 574–587, New York, 2023. ACM.

[75] W. Huangfu et al. Beacon: Scalable near-data-processing accelerators for genome analysis near memory pool with the cxl support. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 727–743, Piscataway, 2022. IEEE.

[76] S.-P. Yang et al. Overcoming the memory wall with CXL-Enabled SSDs. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, July 2023.

[77] J. Lowe-Power et al. The gem5 simulator: Version 20.0+. *arXiv preprint arXiv:2007.03152*, 2020.

[78] X. Zhang et al. Rethinking design paradigm of graph processing system with a cxl-like memory semantic fabric. In *2023 IEEE/ACM 23rd International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, pp. 25–35, Piscataway, 2023. IEEE.

[79] W. Bai et al. Enabling ecn in multi-service multi-queue data centers. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation (NSDI '16)*, Santa Clara, CA, March 2016.

[80] W. Bai et al. Enabling ecn over generic packet scheduling. In *Proceedings of the 12th International on Conference on Emerging Networking EXperiments and Technologies (CoNEXT '16)*, Irvine, CA, December 2016.

[81] A. von Bechtolsheim et al. Why big data needs big buffer switches. 2016.

[82] W. Yoon et al. DiLOS: Do Not Trade Compatibility for Performance in Memory Disaggregation. In *Proc. of EuroSys*, 2023.

[83] X. Zhu et al. Gemini: A Computation-Centric distributed graph processing system. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pp. 301–316, November 2016.

[84] J. Leskovec and A. Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, June 2014.

[85] J. Carlson. *Redis in action*. Simon and Schuster, 2013.

[86] Redis Labs. Memtier Benchmark. https://github.com/RedisLabs/memtier_benchmark.

[87] K. He et al. Deep residual learning for image recognition, 2015.

[88] A. Krizhevsky. Learning multiple layers of features from tiny images. Technical report, 2009.

[89] R. Eldan and Y. Li. Tinystories: How small can language models be and still speak coherent english?, 2023.

[90] Y. Wang. Mapreduce lite. https://github.com/wangkuiyi/mapreduce-lite.

[91] M. Naumov et al. Deep learning recommendation model for personalization and recommendation systems. *CoRR*, abs/1906.00091, 2019.

[92] O. Chapelle. Criteo dataset. https://www.kaggle.com/datasets/mrkmakr/criteo-dataset.

[93] A. Xilinx. 10g axi ethernet checksum offload example design. https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/2425749532/10G+AXI+Ethernet+Checksum+Offload+Example+Design.

[94] JEDEC. Jedec memory controller standard – for compute express link (cxl). https://www.jedec.org/standards-documents/docs/jesd319.

[95] L. W. McVoy et al. Lmbench: Portable tools for performance analysis. In *USENIX annual technical conference*, 1996.

[96] P. Patarasuk and X. Yuan. Bandwidth optimal all-reduce algorithms for clusters of workstations. *J. Parallel Distrib. Comput.*, 69(2):117–124, February 2009.

[97] M. Li et al. Scaling distributed machine learning with the parameter server. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pp. 583–598, October 2014.

[98] P. Kumar and H. H. Huang. GraphOne: A data store for real-time analytics on evolving graphs. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pp. 249–263, February 2019.

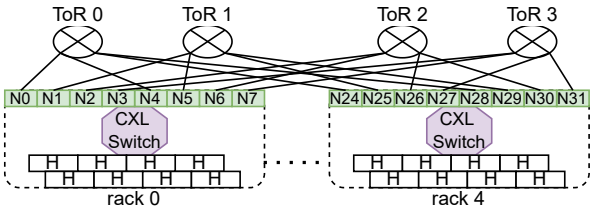
[99] J. He et al. Fastermoe: modeling and optimizing training of large-scale dynamic pre-trained models. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '22, pp. 120–134, 2022.

[100] A. Dixit et al. On the impact of packet spraying in data center networks. In *2013 proceedings ieee infocom*, pp. 2130–2138. IEEE, 2013.

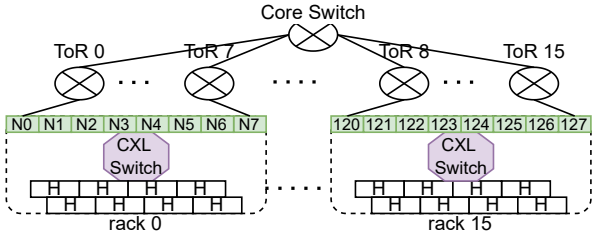
[101] C. Hopps. Rfc2992: Analysis of an equal-cost multi-path algorithm, 2000.

A Appendix

A.1 Topology Example



(a) 4 racks and 8 hosts per rack. ($m = 4, n = 8$)



(b) 16 racks and 8 hosts per rack. ($m = 16, n = 8$)

Figure 12: Example topology with DRack. Host(H), NIC(N).

As shown in Figure 12, there exist two system configurations leading to two different topologies.

① DRack topology has more hosts (n) than racks (m), $n > m$. As shown in Figure 12(a), $n = 8$ and $m = 4$, every rack can connect with all aggregated switches (or ToR switches in original ToR-based architecture). Recall, NIC i is linked with the aggregated switch (i mod m), where $0 \leq i < mn$. Thus, NIC 0, 4, 8,.., is linked to switch 0, NIC 1, 5, 9,.., is linked to switch 1, *etc.*.

② DRack has less hosts than racks, $m > n$. As shown in Figure 12(b), $n = 8$ and $m = 16$, a rack can only connect with part switches. In this case, every rack connect with 8 ToR switches, rack 0, 2, 4,..., connects with ToR switch 0 to 7, while rack 1, 3, 5,..., connects with ToR 8 to 15. They both form two pods, respectively. The two pods are interconnected via one or more core switches, with the pod-to-core wiring topology preserving the conventional structure employed in ToR-based architectures. The wiring between NICs of a DRack and ToR switches still follows the principle, *i.e.*, NIC i is linked with the aggregated switch (i mod m), where $0 \leq i < mn$.

A.2 Application semantics

Table 3 provides more application examples that can result in irrgular computation and skewed data accessing.

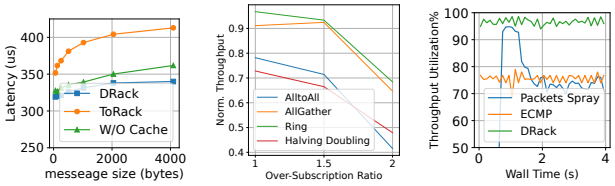
A.3 An Example of NIC Pool Accessing

When a MPSoC0's application calls *send*, its driver classifies whether it targets MPSoC1 (*i.e.*, intra-rack), or the one across racks (*e.g.*, MPSoC3) based on the IP. For the across-rack case, the server's CPU is notified via a CXL-DoCE transaction carrying an interrupt, sent by the driver. Then, it DMA reads the data to the "vNIC" software queue from the MPSoC0's memory via CXL.io. Packets are then sent to the emulated network queue, delaying for $propagation_delay + queueing_delay + \frac{packet_size}{bandwidth}$ before writing it to the software queue of the other "vNIC", which emulates the network delay. For the intra-rack case, the driver stores data descriptors into the reference queues of MPSoC1 with a CXL-like MemWr. This also embeds interrupt within the CXL transaction to notify MPSoC1's CPU. Upon receiving the interrupt, the CPU initiates a CXL-like memory read (MemRd) to retrieve the data from the memory pool.

Table 3: Application Semantics

Application Category	Example	Causes
Graph Computing	Evolving graphs [98]	Different subgraphs running on each host contribute varying compute time
Database & KV Store	Redis [85]	Imbalanced Hot keys distribution & Varied value size
Model Training	MoE Expert Parallelism	Skewed token load on experts [99]
DLRM Training Inference	Embedding layer [48]	Imbalanced Hot Embedding lookups on Embedding tables

A.4 Communication Efficiency



(a) DRack reduces intra-rack delay via pass-by-reference and DRAM Cache.

(b) Throughput of 4 collective operations across ECMP and packet spray. network settings

(c) MPTCP outperforms

Figure 13: Key designs' impact on DRack's performance.

In this section, we eliminate computations and focus on studying the maximum communication efficiency that DRack can boost, as well as the impact of each individual component.

Intra-rack communication. In the rack, `DRack` eliminates one memory copy by using pass-by-reference, and hides the remote memory latency via DRAM Cache. We evaluate the latency reduction achieved by `DRack` for intra-rack communica-

tion including reliable validation of TCP stack. We conducted the TCP transaction latency test using lmbench [95], which involves repeatedly sending a packet filled with dummy data in a ping-pong fashion. The latency for DRack is measured from the moment the sender stores the packet's reference until the receiver receives the reference, loads its data, and completes its TCP stack processing. As shown in Figure 13(a), DRack achieves an average of 15.9% lower latency than ToRack's pass-by-value. The DRAM cache accounts for 6% of the latency reduction, as the CPU will load a packet's header for multiple times during TCP stack processing that presents high cache locality. Note that the major latency of pass-by-value comes from the TCP protocol stack and controlling DMA for data copy, while DRack eliminates the later one.

Inter-rack Throughput. DRack facilitates a range of collective communications at full capacity, irrespective of the network configuration, *i.e.*, network-oblivious communication. This property frees application designers to tailor communication collectives across network settings (*e.g.*, oversubscription ratio). As shown in Figure 13(b), we run four collective communication operations using Gloo over 8 hosts and increase the rack size, the number of hosts in a rack, from 2 to 4, to have an over-subscription ratio from 1 to 2. As DRack uses the NIC pool as the ToR layer, it provides full-bisectional bandwidth across racks, consistently offering throughput gains for all collectives compared to ToRack. ToRack limits the throughput of collectives with intensive cross-rack traffic due to the ToR's uplinks (*e.g.*, AlltoAll), while it cannot fully utilize the uplink bandwidth for collectives with reduced cross-rack traffic (*e.g.*, ring and halving doubling).

We evaluate the necessity of MPTCP in bandwidth efficiency. DRack leverages MPTCP to balance the inter-rack traffic load between 2 available NICs. We first evaluate the throughput with three popular traffic distribution mechanisms (with the other DRack's component enabled): 1) Packets spray: scattering packets between all NICs [100], 2) ECMP: hashing a flow to a NIC [101], 3) MPTCP. We run Gloo AlltoAll with the above mechanisms, respectively. As shown in Figure 13(c), MPTCP fully utilizes the NIC pool capacity, while ECMP cannot utilize the bandwidth efficiently as a single flow can only use one NIC, even if the other NIC is free, packet spray can result in out-of-order packet arrivals, causing TCP congestion control that actively slows down the throughput.

ShieldReduce: Fine-Grained Shielded Data Reduction

Jingyuan Yang[†], Jun Wu[†], Ruilin Wu[†], Jingwei Li[†,*], Patrick P. C. Lee[‡], Xiong Li[†], Xiaosong Zhang[†]
[†]*University of Electronic Science and Technology of China* [‡]*The Chinese University of Hong Kong*

Abstract

Storage savings and data confidentiality are two primary yet conflicting goals in outsourced backup management. While deduplication-aware encryption has been extensively studied to make deduplication viable for encrypted data, it is incompatible with fine-grained delta and local compression for further storage savings. We present ShieldReduce, a secure outsourced storage system that aims for fine-grained shielded data reduction by applying deduplication, delta compression, and local compression to data in a trusted execution environment based on Intel SGX, so as to achieve high storage savings with security guarantees. To mitigate the I/Os of accessing base chunks for delta compression in SGX, ShieldReduce adopts bi-directional delta compression via a novel hybrid inline and offline compression design to maintain the physical locality of base chunks. Evaluation on various backup workloads shows that ShieldReduce achieves significant speedups over a shielded baseline without bi-directional delta compression, while maintaining comparable storage savings to fine-grained data reduction for plain data.

1 Introduction

Outsourced storage in third-party clouds provides a cost-effective solution for organizations to manage backups for multiple clients. Practical outsourced storage systems should be designed with two major goals in mind: (i) *storage savings*, which reduce the storage footprints of outsourced data to save management costs; and (ii) *data confidentiality*, which protects outsourced data against unauthorized access by malicious users and even cloud operators. The goals, however, are inherently conflicting: clients should encrypt their data to ensure confidentiality, but conventional symmetric-key encryption, which uses unique user-specific keys, makes redundant data from different clients incompressible, thereby prohibiting storage savings through compression.

To resolve this conflict, one class of work in the literature focuses on *encrypted deduplication* (e.g., convergent encryption [20] and message-locked encryption [10, 11]), which relaxes conventional symmetric-key encryption by encrypting each data chunk with a deterministic key derived from the chunk content, so as to allow duplicate data chunks from different clients to be mapped into duplicate encrypted chunks that can be removed via cross-client deduplication. However, encrypted data has high entropy, making encrypted deduplication incompatible with delta and local compression for further storage savings (see §2.2 for details).

We explore *fine-grained shielded data reduction*, which performs deduplication, delta compression, and local compression in sequence for high storage savings, and uses shielded execution to realize the whole data reduction workflow with conventional symmetric-key encryption for data confidentiality. We leverage Intel's Software Guard Extensions (SGX) [6] to provide a trusted execution environment called the *enclave* for shielded execution. Client-side organizations can host an enclave-enabled virtual machine in the cloud, ensuring that even cloud operators cannot access any content inside the enclave. The virtual machine can perform fine-grained shielded data reduction for the data chunks originating from multiple clients inside the enclave, encrypt the reduced outputs, and persistently store the encrypted chunks in the cloud. Fine-grained shielded data reduction is compliant with conventional symmetric-key encryption by removing content redundancies *before* encryption inside the enclave [60], so as to maintain the confidentiality guarantees of conventional symmetric-key encryption.

Realizing fine-grained data reduction in an enclave is non-trivial, since the enclave incurs high context-switch overhead to interact with untrusted host applications outside of the enclave [18, 28]. Delta compression, in particular, works by encoding the content difference between a new data chunk and a previously stored *base chunk* identified as similar to the new data chunk [45, 52, 53, 62]. This challenges the management of base chunks for a large number of data chunks: even though the latest SGX generation [9] allows hundreds of gigabytes in enclave memory, the machine hosting the enclave may not have sufficient memory to support a large-size enclave for buffering all base chunks. Storing base chunks in persistent storage and loading them into the enclave on demand incurs significant overhead in both disk I/Os and context switches.

We present ShieldReduce, an outsourced backup storage system based on fine-grained shielded data reduction. Inspired by the locality of backup workloads [38, 66], ShieldReduce operates on data chunks on a per-batch basis and loads the base chunks in batches from persistent storage into the enclave for fine-grained data reduction, as the base chunks are likely to be physically stored together due to locality. This mitigates I/O overhead compared to reading individual base chunks. However, as more versions of backups are stored, the base chunks become more scattered across different versions

*Corresponding author: Jingwei Li (jwli@uestc.edu.cn)

of backups. Thus, ShieldReduce maintains *physical locality* across different versions of backups via *bi-directional delta compression* by examining the physical distribution of base chunks: if the base chunks are stored together (i.e., strong locality), ShieldReduce delta-compresses new data chunks with respect to the corresponding base chunks (i.e., forward); otherwise, if the base chunks are physically scattered (i.e., weak locality), ShieldReduce treats the new data chunks as base chunks and delta-compresses the old base chunks with respect to the new data chunks (i.e., backward). Since the new data chunks are likely to be logically adjacent, making them base chunks can reconstruct physical locality for future backups with low I/O overhead. ShieldReduce adopts a *hybrid* inline and offline compression approach (i.e., on and off the write path, respectively) to limit the I/O interference of bi-directional delta compression with normal backup operations. We emphasize that ShieldReduce differs from a recent work LoopDelta [61], which applies backward delta compression only to mitigate chunk fragmentation. In contrast, ShieldReduce adaptively switches between forward and backward delta compression to balance the trade-off between I/O performance and storage savings based on a single configurable parameter (see §6 for detailed comparisons).

We extend ShieldReduce based on DEBE [60], and compare ShieldReduce in a networked environment with several baselines using various backup workloads (e.g., source code versions, binary snapshots, websites, and operating system images). ShieldReduce achieves an upload throughput gain of up to 3.5× over a baseline without bi-directional delta compression and maintains comparable storage savings as in fine-grained data reduction for plain data (i.e., without data confidentiality). Compared with an existing performance-oriented approach [67] that we adapt to shielded execution, ShieldReduce achieves a comparable upload speed inline, while achieving additional storage savings of up to 3.6× via offline compression. The source code of our ShieldReduce prototype is at: `https://github.com/YangJingyuan99/shieldreduce`.

2 Background and Problem

2.1 Plain Data Reduction

Overview of data reduction. We explore three data reduction approaches for plain data (without encryption): deduplication, delta compression, and local compression. *Deduplication* has been widely adopted in backup systems [12, 37, 38, 55, 57, 66] and achieves storage savings of around 10× in production backup workloads [55]. It partitions file data into *chunks* (e.g., 4 KiB or 8 KiB each [66]), and identifies each chunk by a *fingerprint*, generated by the cryptographic hash of the chunk content. Assuming that fingerprint collisions for non-duplicate chunks are highly unlikely [14], two chunks are said to be *duplicate* if they have the same fingerprint. Only unique copies (called *physical chunks*) are stored, while duplicate chunks refer to the same physical chunk via small-size references. The fingerprint of each physical chunk and its chunk location are kept in a key-value store called the *fingerprint index*.

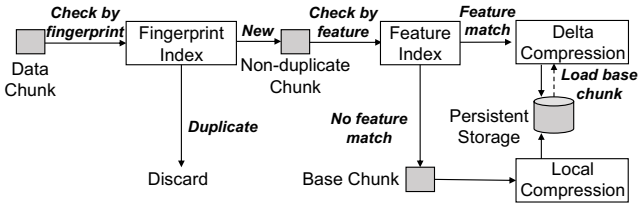


Figure 1: Fine-grained data reduction for plain data.

Delta compression removes duplicate content among non-duplicate but *similar* chunks, which contain large fractions of duplicate content with differences in only a few chunk regions. It can achieve extra storage savings of around 2× on average beyond deduplication and local compression in production backup workloads [53]. One approach of finding similar chunks is to derive *features* from the chunk content [16, 22, 47, 53, 62]; for example, a feature can be the maximum Rabin fingerprint over all sliding windows of the chunk content [22, 40, 47, 53, 62]. Two chunks are considered similar if they share one or more common features. Delta compression stores one of the data chunks (called the *base chunk*) and the content differences from the base chunk to the remaining similar chunks (called *delta chunks*), with each delta chunk typically being smaller than the original chunk. Each feature and its associated base chunk (identified by its fingerprint) are kept in a key-value store called the *feature index*.

Local compression encodes the content of a data chunk into fewer bits in a *lossless* manner for persistent storage, such that the encoded bits can be decoded without information loss. We do not perform local compression on delta chunks, since they are already encoded and local compression only achieves marginal storage savings [53].

Fine-grained data reduction. We apply deduplication, delta compression, and local compression in sequence to achieve fine-grained data reduction for high storage savings [45, 47, 53], as shown in Figure 1. Suppose that a chunk of a file is written to a storage system. The system first queries the fingerprint index to check if the chunk is a duplicate; if so, the chunk is discarded.

If the chunk is new, the system updates the fingerprint index to track the new chunk. It extracts features from the chunk and queries the feature index to check for any similar chunks with common features. If no common feature is found, the chunk is treated as a base chunk. The system locally compresses the base chunk and stores it in persistent storage, as well as adds the feature mappings of the base chunk to the feature index.

If at least one feature of the new chunk is found, the system retrieves the base chunk with the first matching feature from persistent storage and delta-compresses the new chunk with

respect to the base chunk into a delta chunk, which is then stored in persistent storage.

For file reconstruction, the system keeps a *file recipe* in persistent storage to track the fingerprints of delta chunks and corresponding base chunks of each file. All chunks (of several KiB each) in persistent storage are packed and managed in large fixed-size units called *containers* [37] of several MiB each (e.g., 4 MiB). All I/O operations are performed in units of containers. This mitigates the I/O overhead of accessing many small-size chunks.

2.2 Encrypted Deduplication and Its Limitations

Encrypted deduplication. To support data reduction on encrypted data, one extensively studied approach is *encrypted deduplication* [10, 11, 35, 39], in which a client symmetrically encrypts each data chunk to be outsourced to the cloud via a key derived from the chunk content (e.g., via the cryptographic hash of the data chunk [20], or via a server-aided approach where a dedicated key manager returns the same key for the same fingerprint [10]). This ensures that duplicate data chunks, even from different clients, are deterministically mapped to duplicate encrypted chunks that can be removed by deduplication. The client uploads encrypted chunks to the cloud, which cannot access the original data chunks but can still perform cross-client deduplication on the encrypted chunks for storage savings.

Limitations. Encrypted deduplication has two major limitations. First, it must use a deterministic encryption approach [10, 11, 20] to preserve the identical contents even after encryption. This inevitably leaks the number of occurrences of each data chunk in the original plain data, allowing an adversary to launch frequency analysis to infer the plain data chunks based on the frequency distribution of encrypted chunks [33]. From a cryptography perspective, no cryptographic primitive for data reduction can achieve the confidentiality guarantees of conventional symmetric-key encryption [11]. Second, since encryption inherently generates high-entropy encrypted data, it is infeasible to further reduce the sizes of encrypted chunks via delta and local compression for additional storage savings.

One may argue for a simple approach in which clients first perform data reduction on their own data, encrypt the reduced data via conventional symmetric-key encryption, and upload the encrypted reduced data to the cloud. However, this approach prohibits cross-client data reduction for further storage savings in some workloads; for example, multiple clients may perform regular backups on their virtual machine images in cloud environments, while the virtual machine images from different clients commonly share identical content in operating system files [29, 36] that cannot be removed.

2.3 Fine-grained Shielded Data Reduction

Instead of designing new cryptographic solutions, we leverage shielded execution to design *fine-grained shielded data reduction*. We focus on Intel's Software Guard Extensions (SGX) [6] for shielded execution due to its widespread use [19, 30, 42, 46, 49, 50, 60] and long-term support for Intel Xeon platforms [48]. SGX creates a secure running context, called an *enclave*, for code and data processing with confidentiality and integrity guarantees. This contrasts with AMD SEV [2] and Intel TDX [5], which protect the execution of the entire virtual machine (i.e., more coarse-grained) and rely on a sizable trusted computing base [17].

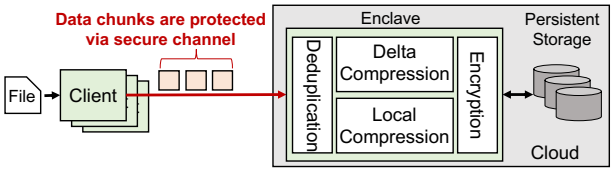


Figure 2: ShieldReduce architecture.

Architecture. We consider an organization that leverages an untrusted cloud to deliver secure backup services to its users. The organization can deploy an enclave-enabled virtual machine instance in the cloud and allow multiple clients to send plain data chunks via secure channels to the enclave [60]. The enclave performs the complete data reduction workflow (i.e., deduplication, delta compression, and local compression) on all received data chunks, encrypts the reduced outputs, and persistently stores the encrypted chunks in the cloud. It eliminates the need for deterministic encryption on data chunks as in encrypted deduplication to mitigate frequency leakage, since fine-grained data reduction is now performed *before* the reduced outputs are encrypted. It further supports delta and local compression, which can now be done before encryption. Note that DEBE [60] also makes a similar observation and supports deduplication-before-encryption, but it does not address delta compression.

We design ShieldReduce to realize fine-grained shielded data reduction for secure and space-efficient backup storage (Figure 2). To upload a backup, a client first sets up a random one-time *session key* with the enclave via the Diffie-Hellman key exchange protocol and establishes a secure session with the enclave, so that any adversary cannot feasibly infer the transferred chunks [60]. The client then partitions the backup file into data chunks, encrypts each data chunk with the random one-time session key, and sends the encrypted chunks to the enclave. Since client-enclave communications are protected by random one-time session keys, they do not leak the frequency information of each data chunk [34]. The enclave decrypts received data chunks with the session key and performs fine-grained shielded data reduction (§2.2). Such *target-based* data reduction [36] prevents a malicious client from inferring the deduplication pattern and launching side-channel attacks [26, 27]. The enclave also maintains a *data key* to encrypt the reduced output data, which is then stored in the cloud for persistence. Note that ShieldReduce does not require a dedicated key manager on the critical I/O path as in

server-aided encrypted deduplication [10] (§2.2).

Threat model. We aim to provide confidentiality guarantees for outsourced backups, which may contain sensitive data. Our threat model considers an adversary that aims to infer the original plain backup data. The adversary can compromise the cloud to access the encrypted contents stored in both untrusted memory and persistent storage, as well as monitor the enclave's interactions with the *untrusted memory* outside the enclave. It can also compromise some clients to access their private information, so as to infer the backup data of other non-compromised clients.

We assume that the adversary and even cloud operators cannot access the contents inside the enclave, even though the enclave is hosted in the cloud. Also, the enclave is securely initialized, and any application code running inside it is authenticated. Enclave authentication is achievable with the aid of a third party (e.g., the organization that provides backup services) [49, 60]. Specifically, the organization compiles the application code into a shared object. It distributes the shared object and its signature to the cloud, which bootstraps the enclave by loading the shared object. The enclave can be publicly verified by remote attestation [6]. Note that the organization *cannot* access any plain data, since the enclave is initialized with the necessary code only without any client data. After attesting the enclave, the organization can go offline, and clients can send backups via secure channels.

The enclave runs the deployed application and verifies the authenticity of the client for each connection via SSL mutual authentication to prevent unauthorized data access. We assume that current countermeasures [15, 24, 44] can be used to protect against attacks for extracting secret information from the enclave [13,43], and we do not further consider such attacks in this work. Finally, the source code of the enclave's binaries can be publicly verified to ensure that it does not include any backdoors. Note that backdoor-free enclaves are commonly assumed in state-of-the-art shielded deduplicated storage [25, 42, 49, 60].

2.4 Challenges

Although previous studies explore SGX-based secure deduplication [25, 42, 49, 60], implementing the complete data reduction process in SGX remains non-trivial.

Resource constraints of SGXv2. In this work, we focus on *SGXv2* [9], the latest SGX generation that supports up to 512 GiB of *enclave page cache (EPC)* (an isolated memory region for an enclave) per CPU socket [23]. SGXv2 significantly mitigates the limited EPC space of its predecessor SGXv1 (which supports up to 128 MiB), yet it still has resource constraints. First, the maximum physical size of an EPC is subject to the available memory size of the hosting machine. In practical cloud environments, an EPC uses only 50% of the overall memory of a virtual machine instance [4]. When an enclave reaches the EPC size limit, it swaps encrypted EPC pages with untrusted memory, leading to expensive paging overhead [18, 23, 28]. Also, SGX provides interfaces for the enclave to securely interact with untrusted memory: *ECalls* for outside applications to securely access in-enclave contents, and *OCalls* for the enclave to access unprotected memory. However, these interfaces incur high overhead, approximately 8,000 CPU cycles per invocation (compared to 150 CPU cycles for a standard system call) [56], due to (i) context switching in the CPU, (ii) operations for preserving the confidentiality of trusted application data, and (iii) flushing of CPU and address translation cache [18, 28].

Expensive base chunk management of delta compression. Fine-grained shielded data reduction should support delta compression (§2.1), yet efficiently managing base chunks is non-trivial, especially when delta compression is applied to numerous data chunks [53, 67]. One option is to keep base chunks inside the enclave. However, the GiB-level memory available in commodity machines for hosting the EPC remains insufficient to buffer all base chunks. Another option is to store base chunks in persistent storage and load them into the enclave on demand. This relieves EPC usage, but incurs significant overhead both in disk I/Os for retrieving the required base chunks from persistent storage and in context switches for loading base chunks from unprotected memory into the enclave.

3 ShieldReduce Design

ShieldReduce is an outsourced backup storage system that aims for the following goals: (i) *storage savings* via deduplication, delta compression, and local compression, (ii) *data confidentiality* via shielded data reduction, and (iii) *high performance* in enclave usage. It builds on DEBE [60] to apply deduplication to data chunks and extend DEBE with delta compression. DEBE mitigates the enclave resource overhead via frequency-based deduplication, which performs first-phase deduplication on the chunks with high duplicate counts using a small fingerprint index inside the enclave, followed by second-phase deduplication on the remaining chunks using a full fingerprint index outside of the enclave. We do not claim novelty in the deduplication design. Instead, our focus is on designing lightweight delta compression and local compression for post-deduplicated chunks (all of which are non-duplicate) inside the enclave and mitigating the base chunk management overhead in delta compression (§2.4).

3.1 Main Idea

To achieve high performance against the challenges in §2.4, ShieldReduce leverages *locality* in backup workloads [32, 41, 53], in which modifications to a backup are often clustered in a few regions. If a data chunk (say M') in the latest backup is modified from a data chunk (say M) in the previous backup, then M is likely to be the base chunk of M', and the neighboring chunks of M are also likely to be the base chunks of the neighboring modified chunks of M'. Our rationale is that small changes are unlikely to alter the features of

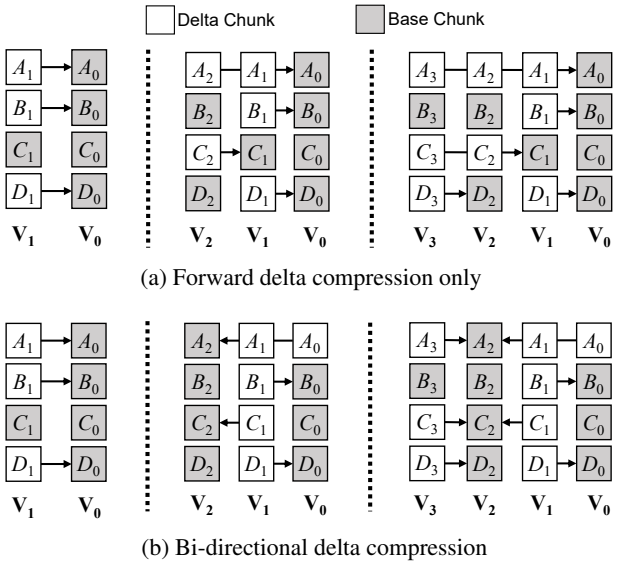
data chunks, so the modified data chunks are likely to follow the same sequence as the corresponding base chunks in the previous backup. Specifically, ShieldReduce operates on data chunks on a *per-batch* basis. It loads the base chunks for a batch of data chunks from persistent storage into the enclave and performs delta compression on the data chunks. Due to locality, the base chunks for a batch of data chunks are likely to be stored in only a few containers, thereby mitigating the I/Os of accessing persistent storage.

However, the locality across two backups in persistent storage (i.e., *physical locality*) gradually decreases as they are separated by an increasing number of backups. Existing fine-grained data reduction techniques [53, 62] often choose the *first* data chunk with new features as the base chunk, so earlier backups typically include many base chunks. Thus, a data chunk M' in a recent backup tends to refer to a base chunk M in an older backup. As changes accumulate across backups, the neighboring physical chunks of M may be heavily updated and no longer be the base chunks of neighboring chunks of M'. This causes ShieldReduce to issue more I/Os to load base chunks from persistent storage into the enclave as the number of backups increases.

To preserve physical locality, we propose *bi-directional delta compression*. Our insight is that chunk similarity is *symmetric*, meaning that if a new data chunk M' is similar to a base chunk M, then M is also similar to M'. This allows delta compression to be performed in two directions: (i) *forward delta compression*, which delta-compresses M' with respect to M, and (ii) *backward delta compression*, which makes M' a new base chunk and delta-compresses M with respect to M'. For a batch of data chunks, ShieldReduce performs either forward or backward delta compression *based on the physical distribution of the corresponding base chunks*. If the base chunks are stored in a limited number of containers (i.e., physical locality exists), ShieldReduce loads these base chunks from persistent storage to the enclave and performs forward delta compression on the batch of data chunks. Otherwise, if the base chunks are scattered across many containers (i.e., physical locality drops), ShieldReduce performs backward delta compression to delta-compress old base chunks with respect to new data chunks. Since the new data chunks are likely to be logically adjacent prior to data reduction, letting them be base chunks can reconstruct physical locality.

ShieldReduce's bi-directional delta compression differs from LoopDelta's backward delta compression [61], which focuses on mitigating chunk fragmentation and does not consider the storage-performance trade-off and security guarantees (§6). We choose Finesse [62] as the major similarity matching technique for ShieldReduce to pair base chunks with their similar chunks due to its high performance and delta-compression savings, and extend Finesse to preserve physical locality. There are possibly other similarity matching techniques, and we pose the analysis as future work.

Example. Figure 3 motivates via an example the need for bi-directional delta compression. Suppose that we store a sequence of backups denoted by $\mathbf{V_0}$, $\mathbf{V_1}$, $\mathbf{V_2}$, and $\mathbf{V_3}$, each of which has a single batch of data chunks. $\mathbf{V_0}$ is already stored.



(a) Forward delta compression only

(b) Bi-directional delta compression

Figure 3: Comparison of forward delta compression only and bi-directional delta compression. The relation $X \rightarrow Y$ for two data chunks X and Y means: (i) both X and Y are similar, (ii) Y is the base chunk of X, and (iii) X is delta-compressed with respect to Y.

Figure 3(a) shows an example of using forward delta compression only. Suppose that $\mathbf{V_1}$ is to be stored, in which A_1, B_1, and D_1 are slightly updated from (and hence are similar to) A_0, B_0, and D_0 in $\mathbf{V_0}$, respectively. Forward delta compression identifies A_0, B_0, and D_0 in $\mathbf{V_0}$ as the base chunks of A_1, B_1, and D_1 in $\mathbf{V_1}$, respectively. Such base chunks are close in persistent storage and are also likely stored in the same container. Suppose that $\mathbf{V_2}$ is to be stored, in which A_2 and C_2 are slightly updated from A_1 and C_1 in $\mathbf{V_1}$. Forward delta compression chooses A_0 in $\mathbf{V_0}$ as the base chunk of A_2, and C_1 as the base chunk of C_2. Now, the base chunks are in both backups $\mathbf{V_0}$ and $\mathbf{V_1}$. When $\mathbf{V_3}$ is stored, we see that the base chunks are scattered in all backups $\mathbf{V_0}$, $\mathbf{V_1}$, and $\mathbf{V_2}$. In other words, physical locality drops.

Figure 3(b) shows an example of bi-directional delta compression. When $\mathbf{V_1}$ is stored, bi-directional delta compression selects the same base chunks in $\mathbf{V_0}$ as in forward delta compression only. When $\mathbf{V_2}$ is stored, it can apply backward delta compression to treat the data chunks A_2 and C_2 in $\mathbf{V_2}$ as the base chunks for A_1 and C_1 in $\mathbf{V_1}$, respectively. Also, it treats A_2 as the new base chunk for the old base chunk A_0, which will be retrieved from persistent storage for delta compression. When $\mathbf{V_3}$ is stored, it can apply forward delta compression again and select A_2, C_2, and D_2 from $\mathbf{V_2}$ as the base chunks for A_3, C_3, and D_3, respectively. The base chunks are still from the same backup and can be readily retrieved.

Design roadmap. Figure 4 presents the design roadmap for ShieldReduce. ShieldReduce adopts a hybrid *inline* (i.e., on

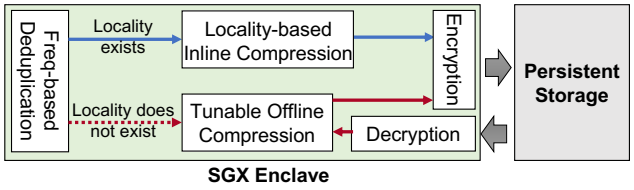


Figure 4: Design roadmap of ShieldReduce. Frequency-based deduplication is proposed by DEBE [60].

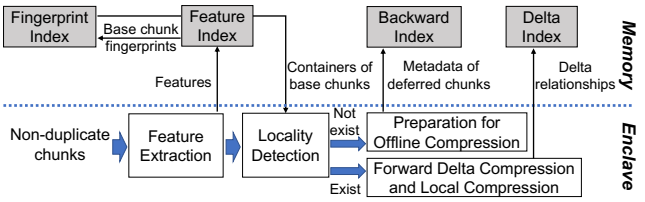


Figure 5: Locality-based inline compression in ShieldReduce.

the write path) and *offline* (i.e., off the write path) approach. After deduplication, if physical locality exists, the enclave performs *locality-based inline compression*, which includes both forward delta compression and local compression, and encrypts and stores the compressed chunks in persistent storage. Locality-based inline compression aims to achieve high performance on the write path.

If physical locality drops, the enclave performs *tunable offline compression*, which reconstructs physical locality via backward delta compression and also provides a tunable mechanism to balance the storage-performance trade-off. As backward delta compression delta-compresses an old base chunk (say M) with respect to a new data chunk (say M'), it should also reapply delta compression on the data chunks that are originally delta-compressed with respect to M; otherwise, any subsequent reconstruction of such data chunks needs to first retrieve and decrypt M' from persistent storage to delta-decompress M, thereby triggering multiple I/Os. ShieldReduce exports a single user-configurable parameter to balance the trade-off between the performance overhead and storage savings from backward delta compression.

ShieldReduce ensures that the enclave always receives data chunks from multiple clients via secure sessions (§2.3) and only stores encrypted data in persistent storage, so as to achieve end-to-end security for the data path.

3.2 Locality-based Inline Compression

Overview. Figure 5 presents the workflow of locality-based inline compression in ShieldReduce. It processes a batch of n data chunks before deduplication ($n = 128$ by default). After deduplication, it extracts three features [62] from each data chunk, queries the feature index for the base chunk (§2.1), and retrieves the container IDs of these base chunks from the fingerprint index. It then detects the physical locality of the base chunks for the batch of data chunks. If physical locality exists, it performs forward delta compression and local compression; otherwise, it prepares for offline compression.

Locality detection. ShieldReduce detects the physical locality of the base chunks of a batch of data chunks based on how they are physically stored in containers. Given n data chunks of a batch, let p be the number of data chunks that have the base chunks and q be the number of containers that contain the base chunks, where $q \leq p \leq n$. ShieldReduce quantifies physical locality as q/n; a small q/n implies strong locality as the base chunks are clustered in a small number of containers. If q/n is no more than some pre-defined *locality threshold* t, we say that physical locality exists. We evaluate the sensitivity of t in §5.3.

Forward delta compression and local compression. If physical locality exists, ShieldReduce performs forward delta compression and local compression in two cases.

- For the p data chunks that have base chunks, it loads the corresponding encrypted base chunks from persistent storage into the enclave. The enclave decrypts the encrypted base chunks, locally decompresses them, and delta-compresses the data chunks with respect to the base chunks into delta chunks. It encrypts and stores the delta chunks in persistent storage. It also updates the file recipe to track the base chunks of such data chunks.
- For the remaining data chunks that do not have base chunks (at most $n-p$ such data chunks due to deduplication), ShieldReduce treats them as new base chunks. The enclave locally compresses, encrypts, and stores the new base chunks in persistent storage. It updates the file recipe and feature index to track the features of the new base chunks.

ShieldReduce maintains a key-value store called the *delta index* in untrusted memory to track the delta relationships, which are later needed by backward delta compression (§3.3) to reapply delta compression on the data chunks originally delta-compressed with respect to old base chunks. Each entry in the delta index maps the encrypted fingerprint of a base chunk to a set of encrypted fingerprints of the corresponding delta chunks. After forward delta compression, ShieldReduce updates the delta index to track the data chunks that have been delta-compressed with respect to the base chunks.

Preparation for offline compression. If physical locality does not exist (i.e., $q/n > t$), ShieldReduce defers the compression for the batch of data chunks offline and performs the following preparation steps. It locally compresses each data chunk and stores the encrypted compressed chunks in persistent storage. It also maintains a key-value store called the *backward index* in untrusted memory to track the chunks for backward delta compression. Each entry in the backward index maps the encrypted fingerprint of a base chunk to a set of encrypted fingerprints of corresponding data chunks that will be delta-compressed offline.

The backward index keeps the mappings until backward delta compression is performed, and the mappings can be cleared afterwards. Thus, ShieldReduce bounds the maxi-

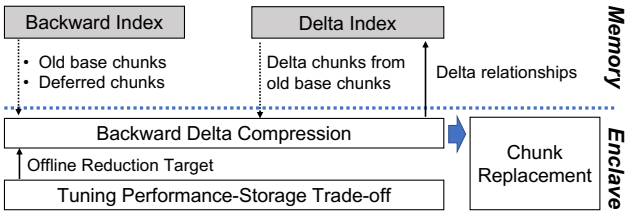


Figure 6: Tunable offline compression in ShieldReduce. The dotted line means that ShieldReduce retrieves chunks from persistent storage based on the backward index and delta index.

mum size of the backward index to limit resource overhead. For example, suppose that each encrypted fingerprint has 32 bytes (from SHA-256) and each data chunk has an average size of 8 KiB. If we configure the backward index with a maximum size of 256 MiB, it can keep the mappings for at least 256 MiB / (32 + 32) bytes $\times$ 8 KiB $\approx$ 32 GiB of data chunks (assuming that each data chunk has a distinct base chunk). If the backward index is full, ShieldReduce performs forward delta compression on the remaining data chunks.

3.3 Tunable Offline Compression

Overview. Figure 6 depicts the workflow of tunable offline compression in ShieldReduce. Based on the mappings in the delta index and backward index, ShieldReduce retrieves from persistent storage the (encrypted) physical copies of (i) the old base chunks, (ii) the data chunks deferred to be delta-compressed, and (iii) the delta chunks already compressed with respect to the old base chunks. It performs backward delta compression based on a user-configurable parameter to balance the storage-performance trade-off. It also replaces the existing physical copies of the above chunks in persistent storage with the new ones that are delta-compressed with respect to new base chunks in offline compression.

Backward delta compression. Recall that each entry of the backward index maps an encrypted fingerprint of a base chunk (say M) to a set of encrypted fingerprints of the data chunks (§3.2). ShieldReduce selects the *most recent* data chunk (say M') in the backward index as the new base chunk. It delta-compresses M, as well as any other data chunk (say $\hat{M}$) that is also mapped by M in the backward index, with respect to M'. Our rationale is that future data chunks are likely to be modified from the most recent data chunks with minimum content differences [61], so choosing the most recent data chunks as new base chunks likely brings more storage savings in delta compression. Note that M' and $\hat{M}$ may not have any common feature, yet we expect that delta-compressing $\hat{M}$ with respect to M' shows storage savings (Exp#1 in §5.2), as they are similar to the same old base chunk M.

We elaborate on the details of backward delta compression. For each old base chunk M, the enclave chooses the most recent data chunk that is mapped by M as the new base chunk M', delta-compresses M with respect to M', and encrypts and stores the delta chunk in persistent storage. It reapplies delta compression on the affected data chunks in two cases. First, for the remaining data chunks that are also mapped by M and deferred to be delta-compressed as tracked by the backward index, the enclave decrypts and recovers such data chunks via local decompression. Second, for the others that are already delta-compressed with respect to M as tracked by the delta index, the enclave recovers them via delta decompression. Then, for both cases, the enclave delta-compresses the data chunks with respect to M', and encrypts and stores the delta chunks in persistent storage. It updates (i) the feature index to track the new base chunk M', (ii) the delta index to track the delta compression relationships, (iii) the fingerprint index to track the new physical copies of the related chunks that are delta-compressed with respect to M', and (iv) the file recipe to change the corresponding base chunks to M'.

Tunable performance-storage trade-off. Since backward delta compression incurs extra I/O and computational overhead (§3.1), ShieldReduce skips the backward delta compression of some old base chunks (i.e., the old base chunks are now directly stored without delta compression). The trade-off is that the physical copies of the skipped base chunks are no longer delta-compressed and the storage size increases. To make the trade-off readily tunable, ShieldReduce introduces a configurable parameter called the *offline reduction target* α (where $0 \leq \alpha \leq 1$) based on the data reduction ratio that is expected to be achieved by offline compression: a smaller (larger) α means that ShieldReduce will remove more (less) data in offline compression.

When processing a backup, ShieldReduce monitors the current storage space occupied by the backup, and tracks the current reduction ratio of offline compression as $\alpha_{current} = S_{current}/S_{inline}$, where S_{inline} is the actual storage space occupied by the backup after inline compression and $S_{current}$ is the currently occupied storage space during offline compression. It collects the old base chunks from the backward index, and sorts them based on the number of data chunks delta-compressed with respect to each old base chunk. It prefers to choose the old base chunk with the least number of data chunks for backward delta compression, and keeps processing the old base chunks until either $\alpha_{current} < \alpha$ or all old base chunks have been processed.

For any remaining old base chunks, ShieldReduce skips their backward delta compression and directly updates the feature index based on the new base chunks. Also, it removes their delta relationships from the delta index to mitigate index overhead. The rationale is that such skipped old base chunks are no longer used for delta compression.

Chunk replacement. After offline compression, ShieldReduce performs the *mark-and-sweep* approach [21] to reclaim the storage space of the old physical copies of the chunks that are delta-compressed by offline compression (e.g., after the old base chunk is delta-compressed with respect to the new data chunk, its physical copy still exists in physical

storage); note that it does not compromise data integrity, as the new delta-compressed copies have been stored in persistent storage. ShieldReduce maintains a temporary *deletion map* in unprotected memory to track the chunks whose old copies need to be deleted. Each entry in the deletion map maps a container ID to a list of deleted chunks, each of which is identified by its offset and length within the container. The deletion map only marks the chunks for logical deletion. Once all chunks have been processed in offline compression, ShieldReduce re-writes the marked containers to reclaim storage space and clears the deletion map.

3.4 Storage Management

Container organization by semantics. ShieldReduce sets each container (§2.1) with a fixed size of 4 MiB, and decouples the management of chunks into two types of containers: (i) the *base chunk container*, which stores encrypted and compressed base chunks, and (ii) the *delta chunk container*, which stores encrypted delta chunks. Our rationale is to improve the I/O efficiency of inline compression (§3.2), since only base chunks are retrieved for delta compression.

Metadata management. ShieldReduce manages the fingerprint index and feature index in unprotected memory, and protects sensitive data fields in both indexes via encryption. In addition to the data key that protects chunk data (§2.3), the enclave persistently manages a *meta key* to encrypt fingerprints. Each entry of the fingerprint index maps an encrypted fingerprint of a data chunk to a container ID where the corresponding chunk data are stored, while that of the feature index maps a feature to the encrypted fingerprint of the corresponding base chunk. Each feature is derived from the original chunk content using a *collision-allowed hash function* [53, 62], so different chunks can have the same features, yet an adversary cannot readily identify the original base chunks based on features only.

ShieldReduce maintains a file recipe (§2.1) for each backup file, so as to track all data chunks of the file. Each entry in the file recipe stores the fingerprint of an encrypted chunk. If the chunk is delta-compressed, the entry also stores the fingerprint of the corresponding base chunk. ShieldReduce encrypts and stores the file recipe in persistent storage.

Download. To restore a backup, a client submits a download request to the cloud. The enclave verifies the client's authenticity (§4). It decrypts the file recipe and restores each data chunk. If the data chunk is a base chunk (i.e., not delta-compressed), the enclave reads the encrypted data chunk, decrypts it, and performs local decompression to recover the original data chunk; otherwise, if the data chunk is delta-compressed, the enclave restores the corresponding base chunk and decompresses the data chunk based on the base chunk. Finally, the enclave sends all data chunks back to the client via a secure channel.

3.5 Security Discussion

We discuss the security guarantees of ShieldReduce against the threat model in §2.3.

Eavesdropping on untrusted memory and cloud storage. An adversary can access the fingerprint index, feature index, delta index, and backward index in untrusted memory of the cloud. Nevertheless, ShieldReduce encrypts fingerprints and generates features via collision-allowed hash functions, so it prevents the adversary from identifying the data chunks from the indexes (§3.4). Note that the adversary may learn from both the delta index and backward index which data chunks are similar to a base chunk, yet it still cannot identify original data chunks, so the practical damage remains an open question. We can still fully encrypt the delta index and backward index to prevent such leakage, yet this adds the overhead of decrypting the indexes inside the enclave for updates and queries. The adversary can access the chunks and file recipes in persistent storage of the cloud, yet ShieldReduce ensures that they are fully encrypted to prevent unauthorized access.

Monitoring interface calls. An adversary can monitor the enclave's boundary on three types of cross-boundary function calls: (i) *index queries*, which query the fingerprint index and feature index based on the encrypted fingerprints and features of data chunks, respectively; (ii) *index updates*, which update the backward index and delta index based on encrypted fingerprints; and (iii) *data transfers*, which move encrypted data via the enclave. The adversary can learn (i) from index queries which encrypted fingerprints and features correspond to a batch of data chunks, (ii) from index updates which chunks will be offloaded to offline compression, and (iii) from data transfers how much data is reduced for each batch of data chunks. Such learned knowledge, however, cannot help the adversary infer original data. For (i), the underlying frequency-based deduplication mitigates the leakage of querying the fingerprint index [60] and ShieldReduce maps distinct chunks to the same feature (§3.4). For (ii), the chunks and fingerprints are encrypted. For (iii), while such leakage is inevitable to support data reduction, an adversary only sees encrypted data without useful information.

Collusion between the cloud and clients. An adversary can compromise both the cloud and a client to additionally access the compromised client's session key and its data chunks. Nevertheless, since the session key is random and used only once, it does not leak other clients' uploaded data that are protected by different session keys.

However, a powerful adversary may now exploit the leakage of how much data is reduced (see above). It can proactively upload artificial chunks via the compromised client and monitor the cloud to see how much data is actually stored. It can learn whether the artificial chunks have been stored by other clients (due to deduplication) and how similar they are to the chunks of other clients (due to delta compression). To defend, we can use selective deduplication [27] to obfus-

cate deduplication patterns and pad delta chunks with random dummy bytes to mitigate compression leakage, at the expense of increasing storage overhead. Alternatively, we can also rate-limit clients' queries to slow down the attack [10].

4 Implementation

We prototyped ShieldReduce based on Intel SGX SDK Linux 2.15. It implements in-memory index structures with `std::unordered_map` and encrypts fingerprints (computed via SHA-256) in each index structure via AES-256 in CBC mode with a fixed zero initialization vector [60], so that identical plain fingerprints are mapped to identical encrypted fingerprints for duplicate detection. It uses multi-threading to parallelize internal operations, and the cloud maintains an in-memory 256-MiB cache to hold recently accessed containers. Our prototype includes 10.5 K lines of C++ code.

Chunking and deduplication. Each client uses FastCDC [59] for content-defined chunking with minimum, average, and maximum chunk sizes of 4 KiB, 8 KiB, and 16 KiB, respectively. It uses the Diffie-Hellman key exchange (§2.3) based on the NIST P-256 elliptic curve. Each client is associated with a per-client master key, which is sent to the enclave during uploads through a secure channel. The enclave encrypts file recipes with the master key to ensure backup integrity. The enclave implements frequency-based deduplication as in DEBE [60] by keeping a small fingerprint index of the 256 K most frequent chunks inside the enclave and a full fingerprint index in untrusted memory.

Compression and storage. The enclave processes batches of 128 data chunks and uses Finesse [62] to extract three features (32 bytes each) from each non-duplicate chunk after deduplication. It uses Edelta [58] for delta compression and LZ4 [8] for local compression. It encrypts each compressed chunk via AES-256 in GCM mode with a random 16-byte initialization vector. It batches the encrypted fingerprints from each batch of chunks to update the delta index, and from all chunks in the entire backup to update the backward index. It stores the encrypted compressed chunks and corresponding initialization vectors in 4 MiB containers.

5 Evaluation

5.1 Methodology

Testbed. We deploy ShieldReduce on Alibaba Cloud [1], using *ecs.r7t.xlarge* virtual machine instances to host cloud storage and multiple clients. Each instance has 32 GiB RAM and a 4-core 2.7 GHz Intel Xeon CPU to support SGXv2, and is installed with Ubuntu 20.04. All instances are connected with a 3 GbE network. The cloud storage instance is attached with a 1 TiB Aliyun SSD (PL0) offering up to 10 K IOPS for persistent storage, except for Exp#4, where we use the Aliyun SSD (PL3) with a maximum of 1 M IOPS to boost aggregate download performance.

Datasets. Our evaluation needs access to data contents, yet no public backup datasets with sensitive contents are available. Thus, we consider three public real-world datasets to mimic sensitive backup workloads: (i) *Linux* [7], which includes 209 versions (from v2.6.11 to v6.4-rc7) of Linux source code with a total of 185.7 GiB of logical data; (ii) *Web* [61, 62, 68], which includes 78 versions (from June 13 to September 1 in 2016) of website backups of `news.sina.com` with a total of 210.8 GiB of logical data; and (iii) *Docker* [3], which includes 95 versions (from v2.1.14 to v4.1) of Docker image snapshots of Cassandra from Docker Hub with a total of 32.2 GiB of logical data. These datasets simulate the development of copyrighted software (Linux) and the maintenance of sensitive services (Web) and environments (Docker). We also consider a synthetic dataset, *SimOS*, where we apply controlled modifications [38] to an operating system image to simulate a user's private activities. We create an initial snapshot from the CentOS 7 virtual disk image (containing 1.25 GiB of data) with a total of 8 GiB of space. We generate a sequence of 30 SimOS snapshots, with a total of 240 GiB of logical data, from the initial image, such that each snapshot is created from the previous one by randomly picking 30% of files, modifying 30% of their contents, and adding 10 MiB of new data.

Baselines. We compare ShieldReduce with three baselines: *DEBE* [60], *ForwardDelta*, and *SecureMeGA*. DEBE performs frequency-based deduplication and local compression in an enclave, but without delta compression (§3). Both ForwardDelta and SecureMeGA extend DEBE with delta compression on a per-batch basis. ForwardDelta loads the base chunks (in containers) from persistent storage into the enclave for forward delta compression (§3.1), but does not perform backward delta compression. SecureMeGA implements MeGA [67] for delta compression in the enclave. It loads the containers with at least three base chunks [67] into the enclave for delta compression and skips the delta compression of data chunks whose base chunks are not in such containers. We use the open-sourced DEBE prototype [60] to implement ForwardDelta and SecureMeGA in C++.

Default configurations. We configure ShieldReduce and all baselines to track the fingerprints of up to 256 K chunks in the enclave for frequency-based deduplication. For ForwardDelta, SecureMeGA, and ShieldReduce, we fix the batch size as 128 data chunks and use three threads to extract features in parallel (except Exp#2, which runs in a single thread to study performance breakdown). We set the locality threshold $t = 0.03$ and the offline reduction target $\alpha = 0$; we study the sensitivity of the parameters in Exp#5.

5.2 Storage Efficiency

Exp#1 (Analysis of data reduction). We compare the data reduction ratios of different approaches. We do not consider metadata overhead, but will study the index overhead of ShieldReduce in Exp#6. Note that ForwardDelta implements

Datasets	DEBE	ShieldReduce				Secure MeGA	Forward Delta
		α=0	α=0.5	α=0.7	α=1		
Linux	5.8	25.8	12.2	8.9	6.3	12.1	25.1
Web	13.1	58.6	27.9	19.9	14.0	16.1	60.6
Docker	8.6	14.9	14.4	12.9	10.3	14.0	15.0
SimOS	59.3	63.6	63.6	63.6	61.0	60.2	63.3

Table 1: (Exp#1) Analysis of data reduction ratio. For $\alpha = 0$, ShieldReduce aims for the highest possible storage savings via backward delta compression, while for $\alpha = 1$, ShieldReduce only updates indexes and skips backward delta compression (§3.3).

fine-grained data reduction (Figure 1) in the enclave and achieves the same data reduction ratio as fine-grained data reduction for plain data.

Table 1 shows the results. ShieldReduce, ForwardDelta, and SecureMeGA achieve higher storage savings than DEBE due to delta compression. When $\alpha = 0$, ShieldReduce has a data reduction ratio comparable to ForwardDelta, with a minor difference attributed to the variation in base chunks used in backward delta compression. ShieldReduce achieves up to a 3.6× data reduction ratio in Web compared with SecureMeGA, since Web includes a large fraction of similar chunks, yet SecureMeGA skips the delta compression of most similar chunks for high performance. Also, the data reduction ratio of ShieldReduce generally decreases with α, since a larger α implies that fewer chunks are compressed offline (§3.3). ShieldReduce keeps almost the same data reduction ratio (63.6×) in SimOS for $0 \leq \alpha \leq 0.7$, as SimOS has very few similar chunks and ShieldReduce's offline compression has limited extra storage savings. In Exp#5 (§5.3), we study the inline and offline time durations to achieve the presented data reduction ratios for different t and α.

5.3 Performance

Exp#2 (Microbenchmarks). We study the performance breakdown of ShieldReduce in uploading backups. We configure ShieldReduce to run with a single thread. We let a client upload the first 20 backups of each dataset and measure the average computational time of different steps on the write path. Table 2 presents a breakdown of the computational time per 1 MiB of logical data processed. ShieldReduce's bi-directional delta compression, including locality detection and delta compression, takes only 2.8-8.8% of the overall computational time. Notably, feature generation takes 55.9% and 52.6% of the overall computational time in Linux and Docker, respectively, and throttles the performance of ShieldReduce, while it only takes 41.3% in Web and 15.2% in SimOS. The discrepancy is due to the relatively small data reduction ratios from deduplication in Linux and Docker (see the data reduction ratio of DEBE in Table 1), so ShieldReduce needs feature generation from more non-duplicate chunks.

Exp#3 (Inline performance). We compare the inline performance of different approaches. We consider a single client that loads a backup into its memory (to remove client-side disk I/Os) and uploads each backup one by one. We measure the *upload speed* by dividing the logical backup size by the total time of writing backup data into persistent storage.

Steps	Linux	Web	Docker	SimOS
Chunking	0.930	0.756	0.841	0.877
Secure session setup	0.208	0.205	0.202	0.190
Deduplication	1.647	1.186	1.499	0.898
Feature generation	6.541	1.978	4.705	0.375
Locality detection	0.138	0.061	0.094	0.011
Delta compression	0.749	0.168	0.688	0.059
Local compression	1.282	0.364	0.750	0.043
Encryption	0.201	0.067	0.168	0.012

Table 2: (Exp#2) The computational time (unit: ms) per processing 1 MiB of logical data in each step on the write path. The results are averaged from the uploads of 20 backups, and the variances are less than 10.5% of the corresponding average values.

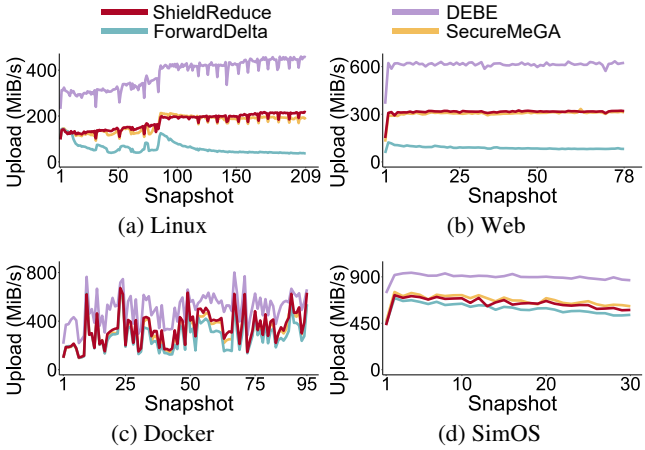
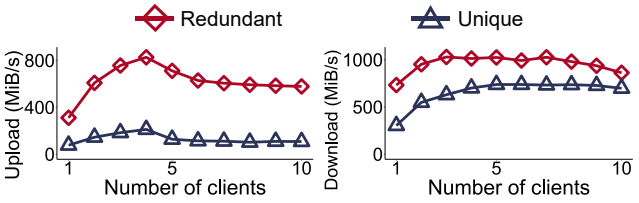


Figure 7: (Exp#3) Upload speed in long-term backups.

Figure 7 shows the upload speeds. DEBE achieves a 1.5-2.2× speedup over ShieldReduce since it does not perform delta compression. Compared with ForwardDelta, ShieldReduce achieves an upload throughput gain of 1.1-3.5× since it preserves physical locality to limit the I/Os of loading base chunks from persistent storage, and further moves delta compression offline if physical locality drops. Notably, SecureMeGA trades storage savings for performance (Exp#1), yet ShieldReduce still accelerates SecureMeGA by up to 7.3%.

We also measure the *CPU utilization* of the cloud machine as the ratio of the actual CPU time to the total processing time on the write path (not shown in figures). ShieldReduce has the highest CPU utilization (50.4%) in Linux among all datasets, since Linux has the most non-duplicate chunks that need encryption. Compared with DEBE, ShieldReduce adds an absolute CPU utilization of 1.1-11.6% for feature extraction and delta compression.

Exp#4 (Multi-client performance). We evaluate ShieldReduce when multiple clients concurrently upload and download backups. We configure ten virtual machines to simultaneously upload (download) data files to (from) the cloud for



(a) Multi-client uploads (b) Multi-client downloads

Figure 8: (Exp#4) Multi-client uploads and downloads.

stress testing. We consider two cases: (i) *Unique*, where each client has a 2-GiB data file with globally unique content (i.e., no redundancies) and (ii) *Redundant*, where each client's data is generated based on controlled modifications like SimOS from the same CentOS 7 virtual disk image.

We measure the *aggregate upload speed* as the ratio of the uploaded data size to the total time taken for the cloud to write all clients' data into persistent storage (in the inline stage). Figure 8(a) shows that the aggregate upload speed of Redundant first increases with the number of clients and achieves 826.7 MiB/s for four clients, and then drops to 578.1 MiB/s for ten clients due to resource contention in the cloud-side enclave. Enhancing the parallelization of SGX is beyond the scope of this paper. However, ShieldReduce can be extended using frameworks like Occulum [51], which supports multi-process execution within a single enclave. Similarly, the aggregate upload speed of Unique gradually increases from 76.3 MiB/s to 211.2 MiB/s for four clients, and is then bounded by disk I/Os in persistent storage.

We also measure the *aggregate download speed* as the ratio of the downloaded data size to the total time taken for all clients to reconstruct files (client-side disk I/Os are not considered). Figure 8(b) shows that the aggregate download speed of Redundant (Unique) first increases to 1024.6 MiB/s (738.1 MiB/s), and then decreases to 862.7 MiB/s (698.0 MiB/s) due to read contention among multiple clients. ShieldReduce in Redundant has a 1.2-1.4× speedup compared with that in Unique, since the container cache (§4) mitigates the disk I/Os of retrieving duplicate data from persistent storage.

Exp#5 (Sensitivity of configurable parameters). We study the impact of the locality threshold t (§3.2) and offline reduction target α (§3.3) on ShieldReduce's performance. We upload each backup by a client, followed by offline compression. We measure the time durations of the data reduction process on and off the write path.

Figure 9 shows the results averaged across all backups when we fix $\alpha = 0$ and vary t. Note that $t = 0$ implies that ShieldReduce locally compresses all data chunks on the write path and defers delta compression off the write path. As t increases, ShieldReduce needs more time to process uploads on the write path, since it loads many containers from persistent storage into the enclave for delta compression. The increase of t only slightly increases inline duration (i.e., the

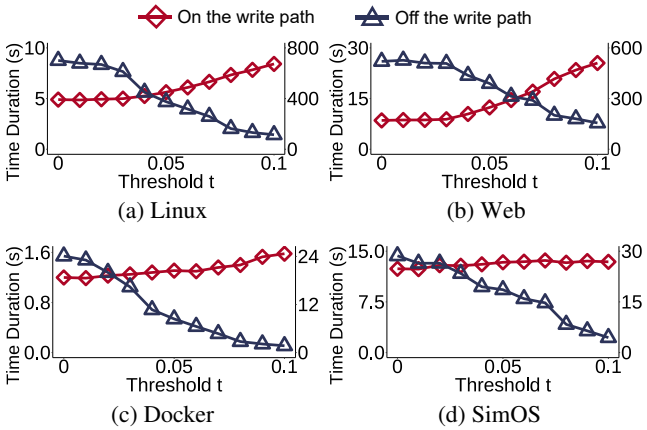


Figure 9: (Exp#5) Impact of t on inline (referred to the left Y-axis) and offline (referred to the right Y-axis) durations.

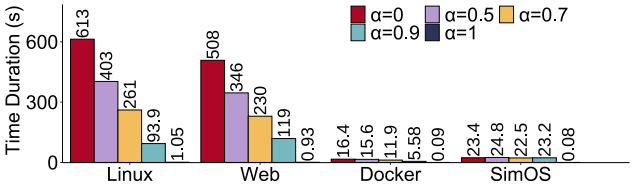


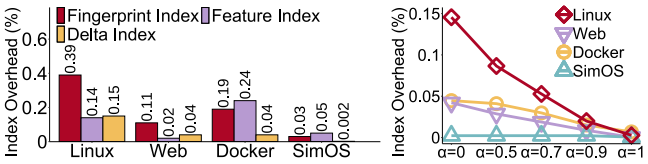
Figure 10: (Exp#5) Impact of α on offline duration.

left Y-axis), but dramatically reduces offline duration (i.e., the right Y-axis), as it avoids reapplying delta compression to many chunks that are already compressed with respect to old base chunks. For example, when t increases from 0.05 to 0.1, the inline (offline) time duration increases (decreases) from 12.4s (389.3s) to 25.5s (155.7s) in Web. Compared to Linux, the write path performance in Docker and SimOS is less affected by t, as they contain fewer similar chunks (Exp#1 in §5.2) and increasing t adds only a small number of chunks for inline delta compression. As future work, we plan to adaptively configure t based on workload characteristics.

Figure 10 shows the average time duration (across all backups) off the write path when we fix $t = 0.03$ and vary α. When $\alpha = 1$, ShieldReduce updates metadata without performing backward delta compression on old base chunks. Clearly, with the increase of α, we can effectively reduce the time duration of offline compression, with the cost of additional storage overhead (Exp#1). Due to the limited number of similar chunks in Docker and SimOS, varying α only slightly affects the number of delta compression operations, leading to a modest impact on the time duration.

5.4 Resource Overhead

Exp#6 (Index overhead). We evaluate the size overhead of index structures in ShieldReduce, including the full fingerprint index outside the enclave, feature index, and delta index. We do not consider the small fingerprint index (based on DEBE) inside the enclave, as it only stores up to 256 K fingerprints (§4) and has a limited size of 8 MiB (assuming



(a) Fraction of index size over logical size

(b) Delta index overhead versus α

Figure 11: (Exp#6) Analysis of index overhead.

32-byte fingerprints). We also do not consider the backward index and deletion map, as both of them will be cleared after offline compression (§3.3). We report the index overhead as the fraction of the size of each index after the last backup over the total amount of logical data.

Figure 11(a) shows that the major index overhead generally comes from the fingerprint index and feature index, which reach up to 0.39% in Linux and 0.24% in Docker. We can mitigate the memory overhead of the fingerprint index and feature index via locality-preserved caching [66] and the stream-informed approach [52], respectively, without introducing extra storage overhead. The delta index incurs 0.14% overhead in Linux. We can increase α to skip the backward delta compression of some old base chunks being used to delta-compress many data chunks (§3.3) to reduce the size of the delta index, at the expense of extra storage overhead (Exp#1 in §5.2). For example, increasing α to 0.5 reduces the overhead of the delta index to 0.09% in Linux (Figure 11(b)).

Exp#7 (Enclave overhead). We evaluate the enclave overhead of ShieldReduce in interface calls. We follow Exp#3 to upload each backup and count ECalls and OCalls on and off (for ShieldReduce only) the write path. We focus on three types of OCalls (§3.5) on the write path: index queries, index updates (for ShieldReduce only), and data transfers.

Table 3 shows the numbers of ECalls and OCalls. On the write path, all approaches issue the same number of ECalls to batch data chunks into the enclave. Compared with DEBE (which issues OCalls to query the full fingerprint index outside the enclave for deduplication), ShieldReduce adds up to 2.4× OCalls (Docker) for two reasons: (i) querying the feature index and loading base chunks into the enclave for delta compression, and (ii) updating the backward index and delta index to offload delta compression off the write path. ShieldReduce reduces the total number of OCalls for ForwardDelta by up to 83.3% (Linux) by preserving physical locality and offloading delta compression from the write path.

To justify the significance of preserving physical locality in ShieldReduce, we disable the offloading of delta compression to offline compression by manually clearing the backward index (i.e., no reconstruction of physical locality). We measure the average amount of data reduced by delta compression per OCall for data transfers and index updates with and without offloading (we exclude index queries as the same number of OCalls are issued in both cases). Table 4 shows that ShieldReduce (with offloading) has up to 4.6× data reduction per

Datasets			DEBE	ForwardDelta	ShieldReduce	
					Inline	Offline
Linux	#ECall		0.2 M			209
	#OCall	Index queries	0.2 M	0.4 M		963.6 M
		Index updates	-	-	0.04 M	
		Data transfers	0.03 M	2.6 M	0.06 M	
		Total	0.23 M	3.0 M	0.5 M	
Web	#ECall		0.2 M			78
	#OCall	Index queries	0.2 M	0.4 M		250.4 M
		Index updates	-	-	0.02 M	
		Data transfers	0.03 M	1.4 M	0.05 M	
		Total	0.23 M	1.8 M	0.47 M	
Docker	#ECall		0.03 M			95
	#OCall	Index queries	0.02 M	0.04 M		19.5 M
		Index updates	-	-	0.01 M	
		Data transfers	0.005 M	0.03 M	0.01 M	
		Total	0.025 M	0.07 M	0.06 M	
SimOS	#ECall		0.1 M			30
	#OCall	Index queries	0.03 M	0.05 M		8.8 M
		Index updates	-	-	0.01 M	
		Data transfers	0.02 M	0.05 M	0.03 M	
		Total	0.05 M	0.10 M	0.09 M	

Table 3: (Exp#7) Enclave overhead. We measure the overhead by the number of interface function calls.

Approaches	Linux	Web	Docker	SimOS
Without offloading	14.3	16.8	42.4	8.1
With offloading	65.9	29.9	47.5	10.9

Table 4: (Exp#7) Average amount of data reduced by delta compression per OCall for data transfers and index updates (KiB) with and without offloading (i.e., performing delta compression offline on similar chunks whose base chunks exhibit weak physical locality).

OCall (Linux) compared to without offloading. The reason is that preserving physical locality allows ShieldReduce to keep more base chunks in the same containers and issues fewer OCalls to load containers in data reduction.

Off the write path, ShieldReduce issues ECalls to start offline compression after each backup upload, and many OCalls to load chunks into the enclave and update the delta index after backward delta compression. We argue that the latter only incurs up to a few hundred seconds for the offline compression of each backup (Figure 9 in Exp#5). We can tune α to reduce the duration of offline compression (Figure 10 in Exp#5) by trading storage savings (Table 1 in Exp#1).

6 Related Work

Performance improvements for delta compression. Earlier studies [22,40,47,53] generate features based on Rabin fingerprints. Finesse [62] mitigates the computational overhead of calculating Rabin fingerprints via fine-grained locality within similar chunks. DeepSketch [45] improves the accuracy of base chunk search based on deep learning with specialized hardware. ShieldReduce builds on Finesse [62] to search for base chunks without specialized hardware.

Recent studies explore fast delta compression that trades storage savings for performance. MeGA [67] uses selective delta compression to avoid loading containers with only a

few base chunks, but it significantly degrades storage savings (Exp#1 in §5.2). LoopDelta [61] proposes inverse delta compression similar to ShieldReduce by treating new data chunks as base chunks and delta-compressing old base chunks with respect to the new data chunks, so as to be compatible with rewriting techniques for the mitigation of chunk fragmentation [37, 61]. However, LoopDelta does not address the I/O overhead especially when reconstructing the data chunks that are originally delta-compressed with respect to old base chunks. ShieldReduce maintains I/O efficiency by performing forward or backward delta compression based on physical locality and proposing tunable offline compression to balance the storage-performance trade-off when applying backward delta compression. Also, LoopDelta only considers plain data reduction, while ShieldReduce realizes shielded data reduction and addresses the enclave resource overhead.

Confidentiality for data reduction. Several studies propose encryption mechanisms to allow the reduction of encrypted data via deduplication [10, 11, 20, 35, 54], local compression [31, 65], or a combination of both along with delta compression [64]. However, they necessitate weaker encryption to preserve content redundancies after encryption, and incur information leakage [33]. Shielded execution has been used for secure data reduction. SeGShare [25] and S2Dedup [42] use an enclave to mitigate the key management overhead of deduplication on encrypted data. SGXDedup [49] leverages SGX to speed up secure deduplication. DEBE [60] mitigates the key management overhead by performing deduplication and local compression before encryption inside an enclave. SeedSync [63] builds on SGX to securely reduce transferred data during cross-cloud synchronization. All the above studies do not consider delta compression.

7 Conclusion

We present ShieldReduce, which leverages SGX to realize fine-grained shielded data reduction for outsourced storage. The main novelty of ShieldReduce is bi-directional delta compression, which performs either forward or backward delta compression based on the physical distribution of base chunks, so as to preserve physical locality and mitigate I/O overhead. Experiments show that ShieldReduce achieves high performance and high storage savings.

Acknowledgments

We thank our shepherd, Steven Swanson, and the anonymous reviewers for their comments. This work was supported in part by National Key R&D Program of China (2022YFB4501200), National Natural Science Foundation of China (62332018), Sichuan Science and Technology Program (2024NSFTD0031, 2024YFHZ0339 and 2025ZNSFSC0497), and Innovation and Technology Commission of Hong Kong (GHX/076/20).

References

[1] Alibaba cloud. https://www.alibabacloud.com/.

[2] AMD secure encrypted virtualization (SEV). https://www.amd.com/en/developer/sev.html.

[3] Cassandra: Open source NoSQL database. https://cassandra.apache.org/_/index.html.

[4] g7t, security-enhanced general-purpose instance family. https://www.alibabacloud.com/help/en/ecs/user-guide/general-purpose-instance-families. Accessed in January 2025.

[5] Intel trust domain extensions (Intel TDX) isolation, confidentiality, and integrity at the virtual machine (VM) level. https://www.intel.com/content/www/us/en/developer/tools/trust-domain-extensions/overview.html.

[6] Intel(R) software guard extensions. https://www.intel.com/content/www/us/en/developer/tools/software-guard-extensions/overview.html.

[7] The linux kernel archives. https://www.kernel.org/.

[8] LZ4. https://lz4.org/.

[9] Supporting Intel SGX on multi-socket platforms. https://www.intel.com/content/www/us/en/architecture-and-technology/software-guard-extensions/supporting-sgx-on-multi-socket-platforms.html.

[10] Mihir Bellare, Sriram Keelveedhi, and Thomas Ristenpart. DupLESS: Server-aided encryption for deduplicated storage. In *Proc. of USENIX Security*, 2013.

[11] Mihir Bellare, Sriram Keelveedhi, and Thomas Ristenpart. Message-locked encryption and secure deduplication. In *Proc. of EuroCrypto*, 2013.

[12] Deepavali Bhagwat, Kave Eshghi, Darrell D.E. Long, and Mark Lillibridge. Extreme binning: Scalable, parallel deduplication for chunk-based file backup. In *Proc. of IEEE MASCOTS*, 2009.

[13] Andrea Biondo, Mauro Conti, Lucas Davi, Tommaso Frassetto, and Ahmad-Reza Sadeghi. The guard's dilemma: Efficient code-reuse attacks against Intel SGX. In *Proc. of USENIX Security*, 2018.

[14] John Black. Compare-by-hash: A reasoned analysis. In *Proc. of USENIX FAST*, 2006.

[15] Ferdinand Brasser, Srdjan Capkun, Alexandra Dmitrienko, Tommaso Frassetto, Kari Kostiainen, and Ahmad-Reza Sadeghi. Dr.SGX: Automated and adjustable side-channel protection for SGX using data location randomization. In *Proc. of ACM ACSAC*, 2019.

[16] Andrei Broder. On the resemblance and containment of documents. In *Proc. of Compression and Complexity of Sequences*, 1997.

[17] Pau-Chen Cheng, Wojciech Ozga, Enriquillo Valdez, Salman Ahmed, Zhongshu Gu, Hani Jamjoom, Hubertus Franke, and James Bottomley. Intel TDX demystified: A top-down approach. *ACM Computing Surveys*, 56(9):1--33, 2024.

[18] Tu Dinh Ngoc, Bao Bui, Stella Bitchebe, Alain Tchana, Valerio Schiavoni, Pascal Felber, and Daniel Hagimont. Everything you should know about Intel SGX performance on virtualized systems. In *Proc. of ACM SIGMETRICS*, 2019.

[19] Judicael B Djoko, Jack Lange, and Adam J Lee. NEXUS: Practical and secure access control on untrusted storage platforms using client-side SGX. In *Proc. of IEEE/IFIP DSN*, 2019.

[20] John R Douceur, Atul Adya, William J Bolosky, P Simon, and Marvin Theimer. Reclaiming space from duplicate files in a serverless distributed file system. In *Proc. of IEEE ICDCS*, 2002.

[21] Fred Douglis, Abhinav Duggal, Philip Shilane, Tony Wong, Shiqin Yan, and Fabiano Botelho. The logic of physical garbage collection in deduplicating storage. In *Proc. of USENIX FAST*, February 2017.

[22] Fred Douglis and Arun Iyengar. Application-specific delta-encoding via resemblance detection. In *Proc. of USENIX ATC*, June 2003.

[23] Muhammad El-Hindi, Tobias Ziegler, Matthias Heinrich, Adrian Lutsch, Zheguang Zhao, and Carsten Binnig. Benchmarking the second generation of Intel SGX hardware. In *Proc. of Data Management on New Hardware*, 2022.

[24] Shufan Fei, Zheng Yan, Wenxiu Ding, and Haomeng Xie. Security vulnerabilities of SGX and countermeasures: A survey. *ACM Computing Surveys*, 54(6), 2021.

[25] Benny Fuhry, Lina Hirschoff, Samuel Koesnadi, and Florian Kerschbaum. SeGShare: Secure group file sharing in the cloud using enclaves. In *Proc. of IEEE/IFIP DSN*, 2020.

[26] Shai Halevi, Danny Harnik, Benny Pinkas, and Alexandra Shulman-Peleg. Proofs of ownership in remote storage systems. In *Proc. of ACM CCS*, 2011.

[27] Danny Harnik, Benny Pinkas, and Alexandra Shulman-Peleg. Side channels in cloud services: Deduplication in cloud storage. *IEEE Security & Privacy*, 8(6):40--47, 2010.

[28] Danny Harnik, Eliad Tsfadia, Doron Chen, and Ronen Kat. Securing the storage data path with SGX enclaves. `https://arxiv.org/abs/1806.10883`, 2018.

[29] Keren Jin and Ethan L. Miller. The effectiveness of deduplication on virtual machine disk images. In *Proc. of ACM SYSTOR*, 2009.

[30] Taehoon Kim, Joongun Park, Jaewook Woo, Seungheun Jeon, and Jaehyuk Huh. ShieldStore: Shielded in-memory key-value storage with SGX. In *Proc. of ACM EuroSys*, 2019.

[31] Demijan Klinc, Carmit Hazay, Ashish Jagmohan, Hugo Krawczyk, and Tal Rabin. On compression of data encrypted with block ciphers. In *Proc. of Data Compression Conference*, 2009.

[32] Erik Kruus, Cristian Ungureanu, and Cezary Dubnicki. Bimodal content defined chunking for backup streams. In *Proc. of USENIX FAST*, 2010.

[33] Jingwei Li, Patrick P. C. Lee, Chufeng Tan, Chuan Qin, and Xiaosong Zhang. Information leakage in encrypted deduplication via frequency analysis: Attacks and defenses. *ACM Transactions on Storage*, 16(1):4:1--4:30, 2020.

[34] Jingwei Li, Chuan Qin, Patrick P. C. Lee, and Xiaosong Zhang. Information leakage in encrypted deduplication via frequency analysis. In *Proc. of IEEE/IFIP DSN*, 2017.

[35] Jingwei Li, Zuoru Yang, Yanjing Ren, Patrick P. C. Lee, and Xiaosong Zhang. Balancing storage efficiency and data confidentiality with tunable encrypted deduplication. In *Proc. of ACM Eurosys*, 2020.

[36] Mingqiang Li, Chuan Qin, and Patrick P. C. Lee. CDStore: Toward reliable, secure, and cost-efficient cloud storage via convergent dispersal. In *Proc. of USENIX ATC*, 2015.

[37] Mark Lillibridge, Kave Eshghi, and Deepavali Bhagwat. Improving restore speed for backup systems that use inline chunk-based deduplication. In *Proc. of USENIX FAST*, 2013.

[38] Mark Lillibridge, Kave Eshghi, Deepavali Bhagwat, Vinay Deolalikar, Greg Trezis, and Peter Camble. Sparse indexing: Large scale, inline deduplication using sampling and locality. In *Proc. of USENIX FAST*, 2009.

[39] Jian Liu, Nadarajah Asokan, and Benny Pinkas. Secure deduplication of encrypted data without additional independent servers. In *Proc. of ACM CCS*, 2015.

[40] Udi Manber. Finding similar files in a large file system. In *Proc. of USENIX ATC*, January 1994.

[41] Dirk Meister, Andre Brinkmann, and Tim Süß. File recipe compression in data deduplication systems. In *Proc. of USENIX FAST*, 2013.

[42] Mariana Miranda, Tânia Esteves, Bernardo Portela, and João Paulo. S2Dedup: SGX-enabled secure deduplication. In *Proc. of ACM SYSTOR*, 2021.

[43] Kit Murdock, David Oswald, Flavio D. Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. Plundervolt: Software-based fault injection attacks against Intel SGX. In *Proc. of IEEE S&P*, 2020.

[44] Oleksii Oleksenko, Bohdan Trach, Robert Krahn, Mark Silberstein, and Christof Fetzer. Varys: Protecting SGX enclaves from practical side-channel attacks. In *Proc. of USENIX ATC*, 2018.

[45] Jisung Park, Jeonggyun Kim, Yeseong Kim, Sungjin Lee, and Onur Mutlu. DeepSketch: A new machine learning-based reference search technique for post-deduplication delta compression. In *Proc. of USENIX FAST*, 2022.

[46] Christian Priebe, Kapil Vaswani, and Manuel Costa. EnclaveDB: A secure database using SGX. In *Proc. of IEEE S&P*, 2018.

[47] Kulkarni Purushottam, Fred Douglis, Jason LaVoie, and John M. Tracey. Redundancy elimination within large collections of files. In *Proc. of USENIX ATC*, June 2004.

[48] Anil Rao. Rising to the challenge—data security with Intel confidential computing. `https://community.intel.com/t5/Blogs/Products-and-Solutions/Security/Rising-to-the-Challenge-Data-Security-with-Intel-Confidential/post/1353141`, 2022.

[49] Yanjing Ren, Jingwei Li, Zuoru Yang, Patrick P. C. Lee, and Xiaosong Zhang. Accelerating encrypted deduplication via SGX. In *Proc. of USENIX ATC*, 2021.

[50] Felix Schuster, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. VC3: Trustworthy data analytics in the cloud using SGX. In *Proc. of IEEE S&P*, 2015.

[51] Youren Shen, Hongliang Tian, Yu Chen, Kang Chen, Runji Wang, Yi Xu, Yubin Xia, and Shoumeng Yan. Occlum: Secure and efficient multitasking inside a single enclave of Intel SGX. In *Proc. of ASPLOS*, 2020.

[52] Philip Shilane, Mark Huang, Grant Wallace, and Windsor Hsu. WAN optimized replication of backup datasets using stream-informed delta compression. In *Proc. of USENIX FAST*, 2012.

[53] Philip Shilane, Grant Wallace, Mark Huang, and Windsor Hsu. Delta compressed and deduplicated storage using stream-informed locality. In *Proc. of USENIX HotStorage*, 2012.

[54] Tong Sun, Bowen Jiang, Borui Li, Jiamei Lv, Yi Gao, and Wei Dong. SimEnc: A high-performance similarity-preserving encryption approach for deduplication of encrypted docker images. In *Proc. of USENIX ATC*, 2024.

[55] Grant Wallace, Fred Douglis, Hangwei Qian, Philip Shilane, Stephen Smaldone, Mark Chamness, and Windsor Hsu. Characteristics of backup workloads in production systems. In *Proc. of USENIX FAST*, 2012.

[56] Ofir Weisse, Valeria Bertacco, and Todd Austin. Regaining lost cycles with hotcalls: A fast interface for SGX secure enclaves. In *Proc. of ACM ISCA*, 2017.

[57] Wen Xia, Hong Jiang, Dan Feng, and Yu Hua. SiLo: A similarity-locality based near-exact deduplication scheme with low RAM overhead and high throughput. In *Proc. of USENIX ATC*, 2011.

[58] Wen Xia, Chunguang Li, Hong Jiang, Dan Feng, Yu Hua, Leihua Qin, and Yucheng Zhang. Edelta: A word-enlarging based fast delta compression approach. In *Proc. of USENIX HotStorage*, 2015.

[59] Wen Xia, Yukun Zhou, Hong Jiang, Dan Feng, Yu Hua, Yuchong Hu, Qing Liu, and Yucheng Zhang. FastCDC: A fast and efficient content-defined chunking approach for data deduplication. In *Proc. of USENIX ATC*, 2016.

[60] Zuoru Yang, Jingwei Li, and Patrick P. C. Lee. Secure and lightweight deduplicated storage via shielded deduplication-before-encryption. In *Proc. of USENIX ATC*, 2022.

[61] Yucheng Zhang, Hong Jiang, Dan Feng, Nan Jiang, Taorong Qiu, and Wei Huang. LoopDelta: Embedding locality-aware opportunistic delta compression in inline deduplication for highly efficient data reduction. In *Proc. of USENIX ATC*, 2023.

[62] Yucheng Zhang, Wen Xia, Dan Feng, Hong Jiang, Yu Hua, and Qiang Wang. Finesse: Fine-grained feature locality based fast resemblance detection for post-deduplication delta compression. In *Proc. of USENIX FAST*, 2019.

[63] Jia Zhao, Yanjing Ren, Jingwei Li, and Patrick P. C. Lee. SGX-enabled encrypted cross-cloud data synchronization. In *Proc. of IEEE ICDCS*, 2025.

[64] Jia Zhao, Zuoru Yang, Jingwei Li, and Patrick P. C. Lee. Encrypted data reduction: Removing redundancy from encrypted data in outsourced storage. *ACM Transactions on Storage*, 20(4):1--30, 2024.

[65] Wenting Zheng, Frank Li, Raluca Ada Popa, Ion Stoica, and Rachit Agarwal. MiniCrypt: Reconciling encryption and compression for big data stores. In *Proc. of ACM EuroSys*, 2017.

[66] Benjamin Zhu, Kai Li, and Hugo Patterson. Avoiding the disk bottleneck in the data domain deduplication file system. In *Proc. of USENIX FAST*, 2008.

[67] Xiangyu Zou, Wen Xia, Philip Shilane, Haijun Zhang, and Xuan Wang. Building a high-performance fine-grained deduplication framework for backup storage

with high deduplication ratio. In *Proc. of USENIX ATC*, 2022.

[68] Xiangyu Zou, Jingsong Yuan, Philip Shilane, Wen Xia, Haijun Zhang, and Xuan Wang. The dilemma between deduplication and locality: Can both be achieved? In *Proc. of USENIX FAST*, 2021.

A Artifact Appendix

Abstract

Our artifact includes the ShieldReduce prototype, which simultaneously supports all baseline approaches, along with scripts for downloading the datasets used in our evaluation and scripts for running all our experiments in §5. The ShieldReduce prototype is a secure outsourced storage system designed to enhance storage efficiency via fine-grained data reduction, while ensuring data confidentiality via shielded execution. It supports upload and download operations and allows multiple clients to securely outsource their backup storage to the cloud. It implements bi-directional delta compression to improve performance and incorporates the hybrid inline and offline designs described in §3. It builds on DEBE [60] to support frequency-based deduplication.

Scope

The artifact can be used to validate all results shown in the figures and tables in §5, especially for our main claims.

- ShieldReduce achieves a similar data reduction ratio to ForwardDelta and higher data reduction ratios than SecureMeGA and DEBE (Exp#1).
- ShieldReduce achieves a similar upload throughput to SecureMeGA and higher upload throughputs than ForwardDelta and DEBE (Exp#3).
- ShieldReduce with offloading reduces more data per OCall compared to without offloading (Exp#7).

Contents

The artifact comprises the following sub-directories:

- `/Prototype` includes the implementation of the ShieldReduce prototype.
- `/MultiClient` includes the implementation of parallel processing of the ShieldReduce prototype (dedicated to Exp#4).
- `/Result` stores evaluation results and scripts for generating plots based on the results.
- `/Dataset` includes scripts for downloading datasets used in our paper. Note that the Web dataset is private and not included in the artifact.
- `/ClientScript` includes scripts for uploading backups from a client.
- `/ServerScript` includes scripts for reproducing all experimental results in §5.

Hosting

The artifact is available on GitHub. Users can obtain the artifact from the repository `https://github.com/YangJingyuan99/shieldreduce`. The version we provided for the artifact evaluation is marked with the `atc25ae` tag. We encourage users to use the latest version of the repository since it may include bug fixes. We also release scripts for downloading (for Linux and Docker) and generating (for SimOS) datasets used in §5. The README file (`https://github.com/YangJingyuan99/shieldreduce/blob/master/README.md`) describes the detailed instructions to produce these datasets.

Requirements

We implement ShieldReduce based on Intel SGX SDK Linux 2.15, Intel SGX SSL 1.1.1l, and OpenSSL 1.1.1l in Ubuntu 20.04 LTS. See our README file for detailed dependencies. We evaluate ShieldReduce on Alibaba Cloud using `ecs.r7t.xlarge` virtual machine instances to host cloud storage and multiple clients. Each instance has 32 GiB RAM and a 4-core 2.7 GHz Intel Xeon CPU to support SGXv2, and is installed with Ubuntu 20.04. All instances are connected with a 3 GbE network. See §5.1 for the detailed evaluation setup.

Workflow

To reproduce the experiments in §5, users can refer to `./AEInstructions.md` for detailed instructions.

MemoryTrap: Booby Trapping Memory to Counter Memory Disclosure Attacks with Hardware Support

Chenke Luo[1,2], Jiang Ming[2], Dongpeng Xu[3], Guojun Peng[1], Jianming Fu[1]
[1]*Key Laboratory of Aerospace Information Security and Trusted Computing, Ministry of Education, School of Cyber Science and Engineering, Wuhan University*
[2]*Tulane University*
[3]*University of New Hampshire*

Abstract

Code-reuse attacks harvest reusable code gadgets from the vulnerable program's executable memory, posing a severe threat to the widely deployed executable-space protection. With the advent of address space layout randomization, a more complicated tactic of code-reuse attacks, known as just-in-time return-oriented programming (JIT-ROP), has emerged. JIT-ROP relies on repeated memory disclosure to search for available code gadgets in real-time. In response, a series of techniques have surfaced to impede memory disclosure or to prevent disclosed code from subsequently being executed. The most representative countermeasures involve enforcing a stricter memory permission policy, such as execute-only memory or destructive code reads. However, existing methods are either vulnerable to emerging code inference attacks or disallow a mixture of code and data, which is a fundamental property of the von Neumann architecture.

In this paper, we present *MemoryTrap*, a hardware-assisted technique to counter direct memory disclosure attacks while simultaneously allowing the mixture of code and data. MemoryTrap sprinkles unreadable "booby traps" in the program at compile time. Once JIT-ROP attackers land in a booby trap area during memory disclosure at runtime, MemoryTrap can immediately detect and stop the ongoing attack. We take advantage of a hardware feature from Intel, Memory Protection Keys, to offer an efficient memory permission control mechanism for booby traps. MemoryTrap supports the security hardening of applications, shared libraries, and dynamically generated JIT code. Our security evaluation demonstrates that MemoryTrap can reliably thwart the threat of disclosing executable memory in real JIT-ROP attacks and synthetic code inference attacks. Performance experiments with both microbenchmarks and macrobenchmarks show that MemoryTrap only introduces negligible runtime overhead.

1 Introduction

System programs are often developed using memory-unsafe languages (e.g., C/C++), making them prone to memory corruption vulnerabilities. The cyber arms race on this topic has transformed into an intensive tug-of-war [1–4]. The widely deployed Data Execution Prevention in modern OSs has pushed cybercriminals to reuse the existing code snippets from the vulnerable program to craft attacks. Adversaries gather the code snippets, called "gadgets," by searching through the disassembled binary code [5]. With these gadgets, attackers carefully chain them together to construct malicious payloads, subsequently hijacking the normal control flow to these gadgets to complete the attack.

A precise implementation of Control-Flow Integrity (CFI) [6–17] offers significant potential for protecting applications against ROP attacks by effectively preventing control-flow hijacking. However, prior research has demonstrated that CFI implementations can still be bypassed under certain conditions [18–27], and may introduce performance overhead [28]. Therefore, from a defense-in-depth perspective [29], it is imperative for critical systems to incorporate a combination of security measures to effectively mitigate potential threats. Address space layout randomization (ASLR) [30–42] is one of the typical representatives of defense-in-depth. ASLR frustrates the construction of gadgets by shuffling the code layout in memory. However, code randomization has suffered from memory disclosure, which makes the randomized code layout visible to attackers [43]. JIT-ROP (just-in-time return-oriented programming) [44] attackers repeatedly abuse memory disclosure to harvest code gadgets on the fly, based on the leakage of code pointers residing on code pages. Recent ASLR measurement [45] demonstrates that even the expensive re-randomization still leaves a time window for JIT-ROP attackers. The premise of JIT-ROP is memory disclosure—attackers have to first traverse sufficient executable memory pages to collect necessary gadgets. Therefore, the key to JIT-ROP defense is to impede disclosing memory pages or to prevent disclosed code from subsequently being executed. To achieve this goal, many security solutions enforce a stricter memory permission policy.

Execute-only memory (XoM) [46–49] emerges as a representative defense against memory disclosure. XoM disallows

the reading of executable memory, and thus attackers lose the ability to disclose the code layout after randomization. XoM methods typically require the strict separation of code and data, so that the unreadable permission can be safely enforced only on code areas. Unfortunately, the mixture of code and data is not rare in practice. For example, libraries frequently contain hand-written assembly routines that incorporate data within their executable code segments [50–52], and JavaScript JIT engines can generate native code containing both code and data in the same memory pages. Later on, a new class of memory permission enforcement policies, called *destructive code reads* [53–56], are proposed to prevent the execution of disclosed code. They allow memory pages to be disclosed but forbid the disclosed code to be executed subsequently. In particular, as soon as the code read occurs, the disclosed code is garbled to avoid being executed. At the same time, legitimate read operations for data embedded in the code still operate correctly. However, the emerging code inference attacks [57] can penetrate the defense of destructive code reads. They enable the reuse of gadgets that have not been previously read, highlighting the need for further research in restricting adversaries' ability to leverage memory disclosure.

In this paper, we continue the line of research on enforcing memory permission from the perspective of cyber deception [58–60]. The building blocks of our new defense are *booby traps*, which are variable-size code snippets but do not have the read permission. They stand aside from the program's normal execution paths and can only be triggered by JIT-ROP attackers. Our research has led us to the critical observation that, with the deployment of code randomization, JIT-ROP attackers only have a small time window (e.g., a few seconds [45]) to complete the search of the whole gadget chain. By inserting booby traps where these attackers anticipate a gadget, we can effectively detect a JIT-ROP attack when it traverses the code page. In addition to stopping the ongoing attack, we also save the context information for further forensic investigation. Unlike previous XoM methods, we do not revoke the readable permission on code pages except for booby traps, and thus legitimate read operations to code pages are completely unaffected. Compared with destructive code reads, our approach is naturally resistant to various code inference attacks caused by code cloning or implicit reads.

One of our design goals is to efficiently manage the unreadable permission at a fine-grained control of booby traps rather than memory pages. To this end, we capitalize on a hardware feature provided by Intel, Memory Protection Keys (MPK) [61, 62], to manage read requests to different areas (i.e., the original code vs. booby traps) at the kernel level. Our solution is more efficient than using the hardware virtualization support (e.g., via Intel Extended Page Tables). In addition, our study also answers another important question: how to insert booby traps cost-effectively so that attackers can trigger them with a high likelihood? We offer a practical insertion strategy at compile time to strike a delicate balance between security and performance penalty.

We develop our proposed solution, called *MemoryTrap*, to offer a comprehensive security hardening for applications, shared libraries, and JIT compiled code. With MemoryTrap, we aim to address the limitations of existing methods and keep our systems safe from JIT-ROP attacks. We first perform an empirical study to measure the effectiveness of MemoryTrap using the real-world exploits that leverage memory disclosure vulnerabilities. We count the distribution of booby traps between gadget intervals to show that attackers are bound to trigger booby traps under different code page traversal strategies. Then, we use code cloning via JIT compilation as an example to present MemoryTrap's defensive ability against the more sophisticated code inference attacks. At last, we conduct a set of performance experiments with both microbenchmarks and macrobenchmarks, including SPEC CPU 2017 [63], three web servers, four database software, and a custom Chromium browser running Kraken JavaScript Benchmark [64]. In contrast to prevailing XoM methods, MemoryTrap only incurs a marginal runtime overhead, averaging from 0.74% to 1.85%.

In a nutshell, we make the following key contributions:

- We present a new cyber deception idea, MemoryTrap, to detect memory disclosure attempts and eventually prevent JIT-ROP attacks. MemoryTrap tolerates legitimate data reads in executable memory while seamlessly defending against powerful code inference attacks.
- We take advantage of Intel's MPK feature to develop an efficient, fine-grained memory permission control mechanism. Our work advances the proper use of hardware features in systems security.
- We offer a versatile implementation of MemoryTrap to secure regular ELF programs, shared libraries, and JIT-compiled code. Our extensive evaluation shows that MemoryTrap has the potential to tip the balance of the memory war toward the side of defenders.

Open Source We have released MemoryTrap's source code and evaluation data sets to facilitate reproduction, replication, and reuse, as all found at Zenodo.

2 Background, Related Work, and Motivation

We first provide the background about JIT-ROP attacks. Then, we summarize the existing countermeasures against memory disclosure. Their limitations motivate our research. At last, we introduce the underlying hardware feature we leveraged.

2.1 Just-In-Time ROP

With the development of fine-grained randomization [30, 31, 36, 39, 42], traditional code-reuse attacks [65] have upgraded to JIT-ROP to generate the ROP payload at runtime. As shown in Figure 1, a typical JIT-ROP attack involves two stages: 1) adversaries first leverage a memory disclosure vulnerability to

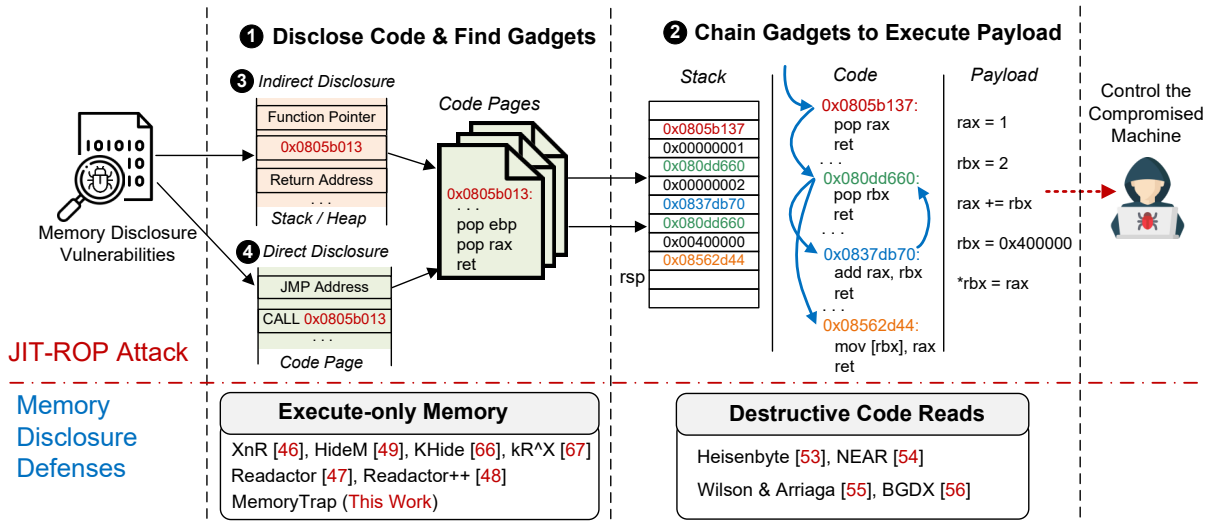


Figure 1: Two stages of a typical JIT-ROP attack workflow and countermeasures on enforcing memory permission.

recursively scan code pages for usable gadgets collection (❶ in Figure 1); 2) after that, they chain the collected gadgets to build the payload and then trigger a memory corruption error (e.g., buffer overflow, use after free, or double free) to hijack the control flow (❷ in Figure 1). In the first stage, attackers can harvest code pointers in a direct way or an indirect way to find executable memory pages. In a direct disclosure, attackers are able to directly read code pointers from code pages. These pointers are typically embedded in instructions such as direct jumps/calls. In an indirect way, code pointers are read from data pages such as stack and heap. This type of pointer is usually the function pointer passed as a parameter of another function or the return address in the stack. The key to the success of a JIT-ROP attack is disclosing enough code pages to find usable gadgets. Programs vulnerable to JIT-ROP attacks primarily fall into two categories. The first category includes server-side applications, which allow multiple user interactions(e.g., web servers and database systems). The second category encompasses client-side applications, which execute user-provided scripts (e.g., Matlab and JavaScript engines).

2.2 Stricter Memory Permission Policies

A set of memory disclosure defense methods have surfaced to combat JIT-ROP attacks. Next, we focus on two types of papers that enforce stricter memory permission policies, because they are the works most germane to our research.

Execute-only Memory The first type of preventative measures focuses on hindering the first stage of a JIT-ROP attack (❶ in Figure 1). They try to disable the read permission of code pages to enforce execute-only memory (XoM). XnR [46] is the first one to achieve the XoM idea via software emulation, without leveraging hardware features. XnR configures the PTE_PRESENT bit in PTE (Page Table Entry) of code pages as the "not_present" state, and thus all read operations will be caught by XnR's page fault handler. Then, XnR detects memory disclosure attempts by checking whether a read request is pointing to a code page. However, due to the large overhead caused by XnR's design, it makes a trade-off to tolerate the co-existence of several code pages with the state of present. As a result, XnR will miss read operations to these co-existence code pages, leaving memory disclosure opportunities for attackers. Readactor [47] and its follow-up work [48] address the security and performance issues of XnR by utilizing the hardware-assisted virtualization via Intel Extended Page Tables [61]. They remove the read permission of all code pages when mapping the virtual machine physical address to the host physical address. HideM [49] implements the XoM idea by desynchronizing ITLB (Instruction Translation Lookaside Table) and DTLB (Data Translation Lookaside Table), so that the same virtual address of code and data is mapped to different physical addresses. In this way, HideM separates data from code pages by redirecting the read operations targeting code pages to the separated data page. KHide [66] uses Hardware Assisted Paging to enforce XoM on kernel, and kR^X [67] implements kernel level XoM by using the Intel Memory Protection Extensions.

Mixture of Code and Data A common assumption of the aforementioned XoM methods is the strict separation of code and data. Such a prerequisite leads to false alarms in practice—XoM will treat legal data-in-code reads as memory disclosure attacks. Although modern GCC and LLVM favor separating code and data, non-code bytes, such as jump table data and static read-only data, often appear in code sections [50]. Pang et al.'s recent study on mainstream binary disassembly tools [52] also confirms that the mixture of code and data is not rare in programs. For example, the authors find 295 hard-coded bytes from the code pages of three test cases and 21,586 jump tables embedded in the code pages of 57 programs. Another example is that handwritten assembly code

in libraries often embeds data in code sections [51]. For example, OpenSSL, BoringSSL, and FFmpeg use handwritten assembly to accelerate their calculation, and VirtualBox uses handwritten assembly for dynamic function loading and its virtual extensible firmware interface. In addition, if a binary dynamically links a library that mixes code and data, the code section of this binary will also contain embedded data. At the same time, XoM conflicts with some security mechanisms. For example, KCFI [8] embedded some function signatures in the code section, and requires read access to the code pages to verify indirect function calls. As acknowledge by the author, KCFI cannot work with execute-only memory.

Destructive Code Reads The second type of memory permission policy, called *destructive code reads* (DCR) [53–56], reacts against the second stage of a JIT-ROP attack (❷ in Figure 1). They allow memory disclosure but prevent executing the previously disclosed code by destroying the disclosed code right after it is read. Heisenbyte [53] is the first approach implementing the idea of DCR. It marks each executable memory page as execute-only and maintains a duplicate copy for each execute-only page. When a read operation occurs in the execute-only page, Heisenbyte overwrites the read data with random bytes and returns the corresponding data values from the duplicate page. In this way, legitimate read operations for data-in-code work properly, but attackers cannot run the disclosed executable memory. NEAR [54] and Wilson & Arriaga's work [55] share a similar idea with Heisenbyte. However, the illusion of protection provided by DCR was soon shattered by the emergence of code inference attacks (a.k.a. ZombieGadgets) [57], in which attackers manage to reuse code gadgets that have not been previously read, rendering DCR's protection ineffective. We provide more details about code inference attacks in Appendix F. BGDX [56] applies a hybrid approach combining DCR and XoM at the byte-granular level and claims to effectively prevent code inference attacks. However, BGDX requires a strict distinction between code and data, and incorrect classification can lead to program crashes. Consequently, BGDX compromises a key advantage of DCR, which is the elimination of the need to differentiate code from data, resulting in reduced compatibility.

CHERI [68] also implements a byte-granularity memory permission control mechanism. However, CHERI requires hardware redesign, including a redesign of the instruction set architecture (ISA), the introduction of multiple new instructions, and the addition of several new registers. CHERI increases the pointer size to 128 bits, resulting in greater cache pressure. Moreover, it demands the integration of a new coprocessor into the Memory Management Unit (MMU) to support its newly introduced fine-grained memory permissions. Additionally, to support the redesigned hardware, CHERI necessitates a CHERI-aware build toolchain and operating system. In contrast, MemoryTrap leverages the existing hardware mechanism, MPK, to achieve fine-grained memory permission control, requiring only minimal modifications to the OS kernel and compiler, thus providing effective security protection with much lower deployment complexity.

Indirect Memory Disclosure Readactor [47] indicates that in some scenarios, attackers can harvest functions pointers stored in data pages (e.g., the stack and heap) and use the whole functions as gadgets to achieve JIT-ROP attacks. This type of attack is called the indirect disclosure attack (❸ in Figure 1). Readactor and Readactor++ [48] have provided a viable countermeasure against indirect JIT-ROP by redirecting code pointers to an XoM area to prevent function pointer disclosure. However, Readactor and Readactor++, together with other XoM methods, have common limitations to prevent direct JIT-ROP (❹ in Figure 1). Therefore, our work is dedicated to direct JIT-ROP defense by offering a new fine-grained memory permission policy.

2.3 Memory Protection Keys

To enforce non-readable permission for our inserted booby traps, we take advantage of Memory Protection Keys (MPK), a hardware feature to support stricter permission control on code pages, without requiring modification of page tables [61, 62]. MPK utilizes a protection key rights register (PKRU) to maintain the access rights of individual keys associated with specific pages, and it supports three different page permissions: read & write, read-only, and no access. Note that the execution permission is still controlled by the traditional permission management mechanism. We can leverage the MPK mechanism to configure a memory page's permission as execute-only by disabling the page's read and write permissions. MPK achieves superior performance because processes only need to execute a non-privileged instruction (*WRPKRU*) to update PKRU, which takes less than 20 cycles and requires no TLB flush and context switching [69]. MPK keys are thread-localized, which means each thread has its own 16 MPK keys. This could lead to inconsistency of MPK keys between different threads within the same process. We use the inter-thread key synchronization primitive provided by libmpk [62] to keep the execute-only MPK key synchronized between different threads in the same process.

MPK Applications in Security Most of the current security approaches rely on MPK for sensitive data isolation [70–74]. For example, ERIM [70] and Hodor [71], take advantage of MPK for efficient intra-process isolation, allowing only trusted code to access sensitive data areas. Jin et al. [73] applies MPK to cryptographic functions to protect sensitive key-related data. IskiOS [74] also uses MPK to revoke the read permission of kernel's code pages. Instead of using MPK as isolation mechanism or simply revoking read permission, we offer efficient fine-grained memory permission policy controlling memory blocks within page.

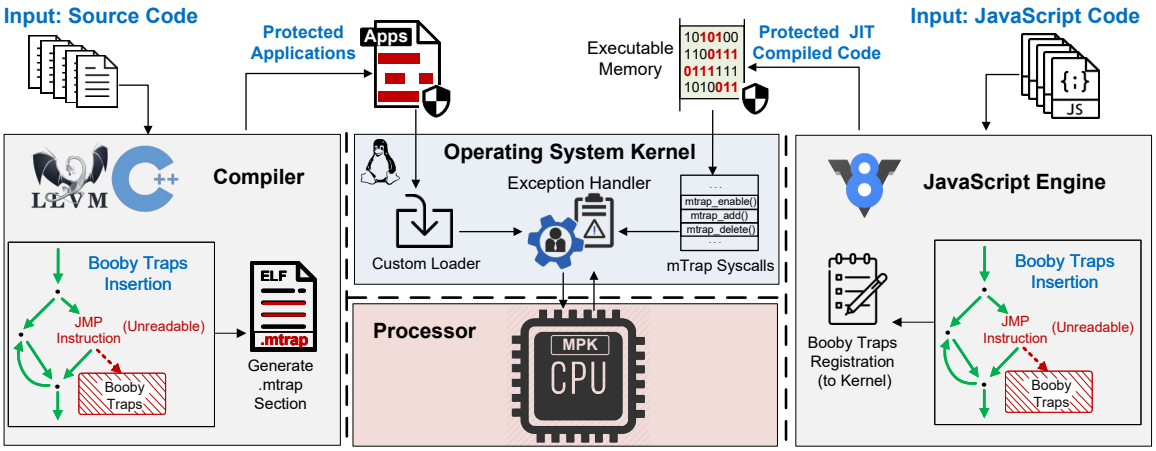


Figure 2: Overview of MemoryTrap.

3 Overview of MemoryTrap

Threat Model Throughout this paper, we assume the attackers possess powerful arbitrary read and write capabilities to the vulnerable program's memory. We do not impose any restrictions on embedding data in code areas. All the aforementioned methods, except for DCR, require that there is no embedded data within the code areas. We assume the target system is equipped with the following protections:

- W⊕X: Memory pages cannot be both executable and writable at the same time. This is the basic assumption of ROP defenses. Otherwise, attackers can directly execute the injected shellcode without performing ROP.
- Randomization: all static code from programs and libraries are randomized via load time fine-grained randomization.

These two protections are also held by related works on memory permission enforcement, such as Heisenbyte [53], Readactor [47], NEAR [54], and HideM [49].

Memory View of Booby Traps MemoryTrap aims to impede the first stage of a JIT-ROP attack (❶ in Figure 1). Different from the existing XoM and DCR, our memory permission policy only works on *booby traps*, which are unreadable, variable-size code snippets that we inserted into the protected program. Figure 3 shows the memory view before/after MemoryTrap's protection. Our insertion strategy ensures that JIT-ROP attackers will inevitably land in a booby trap area when searching for gadgets, while the program's normal execution will never reach any single booby trap. When a read operation to a booby trap happens, MemoryTrap's protection will be triggered instantly to respond to the attack: it terminates the compromised process to prevent further memory disclosure and saves the context information for the future investigation.

Architecture The target programs of memory disclosure attacks cover applications (e.g., web servers and databases), shared libraries (e.g., C standard library) [5], and dynamically

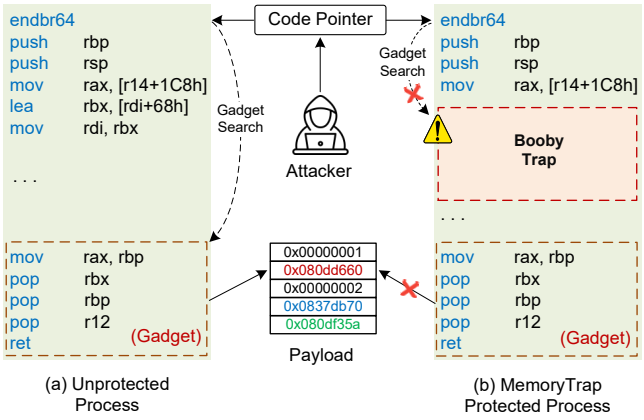


Figure 3: Memory view before/after booby trap insertion.

generated code by a JIT engine like V8 [75, 76]. Therefore, we develop MemoryTrap to support the security hardening of all three types of programs. Figure 2 shows MemoryTrap's architecture that bridges all layers of the software stack. At the user lever, we modify LLVM and TurboFan (V8's JIT compiler) to place booby traps at strategic trigger points. After that, we pass booby traps information (e.g., their numbers and the scope of each booby trap) to kernel components, either via a dedicated ELF section (".mtrap") for applications or via our customized "mTrap" syscalls for shared libraries and JIT compiled code. At the kernel level, MemoryTrap's custom loader loads the hardened applications into memory. With booby traps information, MemoryTrap's exception handler interacts with the low-level MPK mechanism to manage the unreadable permission at the level of booby traps, and it is also responsible for the memory disclosure attack response.

The primary characteristic of JIT-ROP attacks is that the attacker must search through a large amount of program code within a limited timeframe to gather sufficient gadgets. The key insight behind MemoryTrap is to strategically place booby traps along the path attackers follow when attempting to disclose code. To achieve optimal protection, these booby traps

must be distributed throughout memory at an appropriate density. A density that is too high would negatively impact performance, whereas a density that is too low would compromise security guarantees. In the following sections, we describe our approach for establishing insertion strategies that ensure booby traps are effectively distributed across memory at a suitable density.

MemoryTrap's Advantages As a new technique to achieve fine-grained memory permission management, MemoryTrap avoids the limitations of existing XoM and DCR methodologies and amplifies their benefits. Unlike the previous XoM methods [46–49], MemoryTrap allows the mixed use of code and data, and thus legitimate read to code pages works as expected. Compared to DCR [53–56], MemoryTrap can naturally withstand code inference attacks. As booby traps are weaved into the program, a code inference attempt via either code cloning or implicit reads cannot get rid of booby traps in any way. MemoryTrap leverages the new hardware feature MPK, which offers an efficient option to enforce fine-grained memory permission policies. The previous works either use software emulation that reveals high runtime overhead [46, 49], or they rely on the hardware virtualization support via Intel Extended Page Tables (EPT) [47, 48, 53], which is still more costly than MPK [69].

4 Application Protection

In this section, we follow the workflow of hardening an application to present MemoryTrap's details.

4.1 Insert Booby Traps into Execution Paths

We insert booby traps in the protected program at compile-time via a LLVM pass. We randomly generate a variable-size code snippet as a booby trap, which consists of 5 to 30 `NOP` instructions. We adopt variable sizes to further disrupt the code layout, making it more difficult to detect the positions of booby traps. We hide the booby traps from the protected program's normal execution paths by inserting a `JMP` instruction in front of inserted code snippets. When the execution encounters a booby trap, it will skip the inserted code snippets and follow the normal execution path. Please note that we set the whole booby trap, including the prefixed `JMP` instruction, as unreadable. This design enables booby traps to be seamlessly integrated into legitimate code execution paths, making them difficult to detect or circumvent. Because these traps are inherently embedded within valid execution flows, attackers cannot easily bypass them by simply tracking the program's control flow. Moreover, the opacity of inserted `JMP` instructions prevents attackers from identifying trap locations through pattern recognition or signature-based detection methods.

Table 1: The distance distribution of every two adjacent booby traps in the hardened applications' binary code.

	0∼100B	100B∼1KB	1∼2KB	2∼4KB	>4KB
Nginx	0%	73.2%	16.5%	10.3%	0%
Apache	27.5%	65.3%	3.6%	3.6%	0%
Lighttpd	16.0%	73.7%	4.7%	5.6%	0%
MySQL	30.6%	57.2%	6.5%	5.7%	0%
MongoDB	25.5%	62.3%	6.4%	5.8%	0%
Redis	26.1%	65.9%	4.1%	3.9%	0%
SQLite	20.9%	55.2%	7.1%	16.8%	0%
SPEC 2017	47.4%	44.6%	3.9%	4.1%	0%
Average	24.3%	62.2%	6.6%	6.9%	0%

4.2 Insertion Strategy of Booby Traps

We aim to sprinkle booby traps cost-effectively in execution paths to capture memory disclosure attempts.

JIT-ROP's Gadget Search Unlike offline ROP gadget search [77, 78], JIT-ROP requires "*unfettered access to a large number of the code pages*" [44]. JIT-ROP attacks [44, 45] first identify code pointers from data pages (e.g., the stack) and then use these pointers as starting points to locate code gadgets. To avoid disruptions caused by linear searches potentially accessing unmapped memory regions, JIT-ROP incrementally discloses and disassembles code to track control flow. When cross-page code pointers (e.g., long jumps and long calls) are encountered, JIT-ROP employs breadth-first or depth-first traversal strategies to explore memory pages, as shwon in Figure 4. According to the gadget search strategies of JIT-ROP, attackers use 4KB (one memory page's size) as the search granularity to avoid accessing unmapped pages. As a result, if we ensure that the distance between *every* two adjacent booby traps is small enough (e.g., less than 4KB), JIT-ROP attackers will trigger them with a high probability before they complete the search of all required gadgets. At the same time, attackers often use a combination of `pop` and `ret` instructions as gadgets to pass values from the stack into general-purpose registers [65], and such combinations tend to appear at the end of a function.

Insertion Strategy Therefore, our booby trap insertion strategy contains two rules:

1. we insert at least one booby trap in each function.
2. for the function whose size is larger than 4KB, we insert an additional booby trap for every 4KB code.

When adhering to these two rules, the insertion positions of booby traps are randomized. This approach prevents attackers from predicting the locations of booby traps based on fixed positions. Considering the aforementioned factors, our insertion strategy is effective in preventing disclosing sufficient code. We count the distances between every two adjacent booby traps after we apply MemoryTrap to real-world applications and SPEC CPU 2017. As shown in Table 1, the distances between every two adjacent booby traps are all within 4KB. Additionally, the percentage of distances within 1KB is 86.5% on average, representing a high distribution

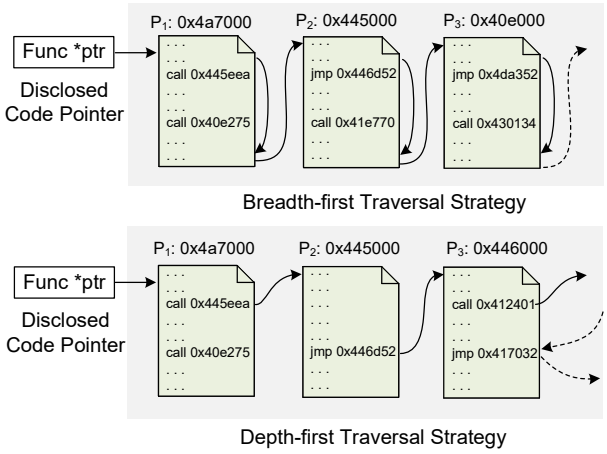


Figure 4: Code page traversal strategies.

density. We conducted an additional experiment to explore the impact of booby traps distribution on security and overhead. It was found that if a booby trap is inserted every 2KB, the average overhead increases by 1.3% for SPEC CPU 2017 benchmarks. On the other hand, if a booby trap is inserted every 8KB, the average overhead decreases by 0.7%, but it leaves a larger memory disclosure window for attackers.

4.3 New ELF File Format

The compiler also needs to pass the number and scope of booby traps to MemoryTrap's kernel components. This requires us to utilize the reserved field and optional section of the ELF file format to embed such metadata in binaries. We illustrate the new ELF file format with a figure provided in Appendix H.

First, we define a reserved byte in the ELF header as the specific flag byte, called **MTRAP_ENABLE**, to indicate whether this program is protected by MemoryTrap. This byte is the first byte in the EI_PAD array, which is a field of e_ident in the ELF header. By reading this byte, our custom loader can decide whether to enable MemoryTrap protection for this process. Then, we embed the number of booby traps and the start and end addresses for each booby trap into an optional section, called ".mtrap. " The beginning of the .mtrap section records the number of booby traps, followed by a booby trap list—each list item stores each booby trap's start and end addresses. We modify LLVM's backend to write booby trap information to the dedicated ".mtrap" section and then set the MTRAP_ENABLE flag to 1.

Please note that the new ELF format is backward compatible with non-customized loaders because they will ignore the MTRAP_ENABLE flag and the .mtrap section. The unmodified kernel can still run the MemoryTrap protected binary files properly as normal programs.

4.4 MemoryTrap's Kernel Components

Our kernel components consist of a custom loader, an exception handler, and mTrap syscalls. We also modify the kernel structures to support MemoryTrap. In this subsection, we introduce the first two components and leave mTrap syscalls in §5, because mTrap syscalls are specifically designed to support shared libraries and JIT compiled code. Our kernel modifications consist of only 312 lines of code, making our approach lightweight and minimally invasive to the kernel.

Custom Loader We customize the binary file loader in the kernel to load the applications protected by MemoryTrap and initialize related structures in the kernel.[1] The custom loader first checks the MTRAP_ENABLE flag in the ELF header to determine whether MemoryTrap's protection is enabled. If so, it will load the booby trap list stored in the .mtrap section. Otherwise, the standard binary file loading process will take over. After loading the booby trap list, the custom loader starts to map the code segment into memory. When mapping the code segment, the custom loader allocates an execute-only PKey, which is a part of the MPK mechanism to set the permission for a group of pages, to enforce the access permission as execute-only. Please note that this PKey is allocated during the loading process, thereby ensuring its availability. Consequently, the way the program utilizes PKeys at runtime does not influence the implementation of MemoryTrap. In this way, our exception handler component can capture every read operation to code pages, and then it further determines whether a memory disclosure attempt or a legitimate code read is occurring.

Exception Handler We implement an exception handler based on the original page fault handler for the MPK mechanism to prevent memory disclosure attacks. Since we have removed the read permission of all code pages via the MPK mechanism, any read request to a code page will trigger a page fault and be caught in our exception handler. Then, the exception handler checks whether the target address is located in the areas of booby traps. If so, we can promptly determine that the running program is under a memory disclosure attack, and thus we terminate the compromised process and save the context information for further forensics investigation. Note that legitimate read operations for data embedded in the code do not trigger MemoryTrap's attack response. Instead, we take the following actions to handle such data-in-code cases: 1) we restore the read privilege of the target page to allow the page to be readable temporarily; 2) we set the single-step trap flag to only execute the read instruction and stop at the next instruction; 3) after the completion of the legal read operation, we revoke the read permission of the code page and subsequently clear the single-step trap flag to enable the program to resume its normal execution. Please note that MemoryTrap only affects the permissions of code pages and primarily in-

[1] Regarding how we organize booby traps information in the kernel structures, please refer to Appendix A.

troduces overhead during code page reads, without impacting data pages. As a result, MemoryTrap does not affect the performance of copy-on-write (CoW) operations because CoW typically occurs on data pages.

4.5 Booby Traps Layout Randomization

MemoryTrap is compatible with load time fine-grained randomization methods to shuffle booby traps' location. We have adapted MemoryTrap to a representative tool, CCR [79]. CCR randomizes the code layout at the granularity of basic blocks by patching the binary code. When CCR randomizes the code layout, the locations of booby traps must be updated synchronously. The randomization process of CCR contains two phases: information collection at compile-time and binary rewriting. In the first phase, we modify CCR to insert booby traps first and then collect pointer and code layout information. In the second phase, we modify CCR's binary rewriter to update the addresses of booby traps when CCR randomizes the code layout. When shuffling a booby trap basic block, CCR will also update the metadata stored in the ".mtrap" section using the new shuffled addresses.

5 Shared Library & JIT Compiled Code

In this section, we discuss the security hardening of shared libraries and JIT compiled code. They exhibit some different challenges from the above application protection.

mTrap Syscalls The first challenge is that both the loader of shared libraries and the JIT engines are user-level programs, but the operations of booby traps, such as enabling MemoryTrap's protection and adding new booby traps, all happen in the kernel only. Therefore, they have to rely on specialized system calls to interact with the kernel and perform the booby trap operations. To this end, we have implemented three new system calls: `mtrap_enable`, `mtrap_add`, and `mtrap_delete`. The `mtrap_enable` enables MemoryTrap's protection explicitly, and the `mtrap_add` and `mtrap_delete` are used to add and delete booby traps, respectively.

Shared Library Protection For shared libraries, their booby trap insertion strategy and the new ELF file format are the same as our application protection. The major difference is that applications are loaded by the custom loader in the kernel, while shared libraries are loaded by the GNU C Library loader, a user-level program. We modify the GNU C Library in several ways to achieve our goal. 1) When loading a hardened shared library, the loader enables MemoryTrap's protection by invoking the syscall `mtrap_enable`. 2) Next, the loader maps the code segments as execute-only memory and fixes the addresses of booby traps due to the relocation of shared libraries. 3) At last, it loads the booby trap list from the ".mtrap" section and registers the booby trap list to kernel by invoking `mtrap_add`. After these steps, MemoryTrap's exception handler will promptly start to monitor memory disclosure attacks.

JIT Compiled Code Protection MemoryTrap's JIT version is built on top of V8's JIT compiler, TurboFan. As shown in Figure 2, we modify the JIT engine to compile JavaScript into the executable code with booby traps inserted. In particular, TurboFan first runs the graph creation and optimization passes to generate IR from source code, and then it compiles IR to the platform-specific machine code. We insert booby traps at the IR level using the same insertion strategy discussed at §4.2. After compilation, the modified TurboFan enforces the execute-only permission on the generated JIT code via MPK. Then, it registers embedded booby traps to the kernel by invoking mTrap syscalls so that the exception handler can start to monitor memory disclosure.

Furthermore, we also adjust our memory permission management accordingly to support some aggressive optimizations of JIT compilers. For example, to save memory, JIT compilers may write the new JIT code to the existing JIT code pages, while the read and write permissions on these pages have been removed to facilitate the booby trap checking. In this case, the modified TurboFan will temporarily recover read and write permissions on these code pages to allow the JIT code update. After that, it will disable read and write permissions for the updated JIT code pages again and invoke the `mtrap_add` syscall to register new booby traps. If TurboFan plans to abandon a generated JIT code snippet, it will also invoke the `mtrap_delete` syscall to unregister all of the booby traps in the abandoned JIT code.

6 Security Evaluation

We implement MemoryTrap on Linux kernel 5.10.11, LLVM 13.0.1, V8 9.7, and GNU C Library 2.31. The testbed is a server machine with Intel i9-13900KF 24 cores CPU and 64GB RAM. We evaluate how well MemoryTrap can defend against memory disclosure attacks under three different scenarios: 1) JIT-ROP attacks on applications; 2) JIT-ROP attacks on JIT compiled code; 3) code inference attacks [57].

6.1 Detect Memory Disclosure on Applications

We use a Nginx arbitrary memory disclosure vulnerability (CVE-2013-2028) [80] to demonstrate MemoryTrap's effectiveness. This is a fairly powerful stack overflow vulnerability that allows an adversary to perform arbitrary memory read. We compile the vulnerable version of Nginx with MemoryTrap protection enabled and run it as a web server. Then, we leverage the JIT-ROP attack framework, jitrop-native [81], to trigger the vulnerability and dynamically search for gadgets.

We collect all gadgets in the exploit of CVE-2013-2028 and measure how gadgets and booby traps distribute in memory. The gadget chain of this exploit consists of seven gadgets to implement two functions: storing a value in memory and

```
1. 0x0804b884: pop esi; pop ebp; ret;
... 192 Booby Traps ...
2. 0x0805ddbb: push ecx; ret;
... 162 Booby Traps ...
3. 0x0806e948: add ecx, [esi]; ret;
... 103 Booby Traps ...
4. 0x0807b586: mov [ecx], eax; mov [ecx+4], edx; mov eax, 0; ret;
... 47 Booby Traps ...
5. 0x08080a57: pop ecx; add al, 0x89; ret;
... 279 Booby Traps ...
6. 0x0809a65b: pop eax; add al, 0x89; ret;
... 50 Booby Traps ...
7. 0x0809ef79: push eax; add al, 0x83; ret;
```

Figure 5: Gadgets and booby traps distribution for the CVE-2013-2028 exploit. Each line of assembly code is a gadget, stating with its address. We show the number of booby traps distributed between two adjacent gadgets.

dereferencing a pointer. As shown in Figure 5, our insertion strategy ensures that booby traps spread over the interval of every two adjacent gadgets. The number of booby traps between two adjacent gadgets ranges from 47 to 279.

Next, we exhibit the attackers' ability to find potential gadgets under MemoryTrap's protection. We are interested in measuring under different code page traversal strategies, how much code can be disclosed, and how many usable gadgets can be found before triggering the first booby trap.

Starting Point & Traversal Strategy The recent work [45] demonstrates that the starting point of code traversal in a page does not affect the disclosure result, because the authors experimentally confirm that choosing any single random code pointer allows attackers to identify all instructions and code pointers in that code page. Besides, selecting which page to start gadget search is not important as well, because code pages are all very well connected [45]. Therefore, we start code disclosure from 100 random positions in every code page. Another factor we must consider is code page traversal strategy, which can affect the disclosure result [44]. We utilize Ahmed et al.'s traversal strategies [45], including depth-first and breadth-first traversal, to find all required gadgets.

The code segment size of compiled Nginx is 815.8KB, which equals 203.9 memory pages. Thus we start code page traversal using two different strategies from a total of 20,390 different positions; once any booby trap is triggered, we record how much code and how many usable gadgets are disclosed. Table 2 shows the results of our code page traversal attempts. When using the breadth-first policy, an average of 657 bytes of code can be disclosed before triggering booby traps, and an average of 299 bytes of code when using the depth-first search. Among all of our code page traversal attempts (40,780 in total), only 25 attempts using the breadth-first strategy and 14 attempts using the depth-first strategy can find **at most one gadget**. The rest of code page traversals are all trapped in booby traps before finding any usable gadget.

More Exploits We conduct a gadget distance statistical analysis on a set of real-world JIT-ROP exploits, which we collect from Metasploit [82] and Exploit-DB [83]. We use the same code traversal strategies to measure how many bytes can be disclosed under the protection of MemoryTrap, and the results are consistent with the previous findings. The disclosed code size ranges from 499∼687 bytes with breadth-first strategy, and 273∼340 bytes with depth-first strategy. As a contrast, the minimum distance from the first gadget to the last gadget is 94.0KB, indicating that the disclosed code is far from enough to find all gadgets. Similarly, only a few code traversal attempts get a chance to find **at most one gadget**. We provide the detailed results in Appendix E.

Table 2: Attackers' ability to find potential gadgets when MemoryTrap is enabled. The vulnerable program is Nginx.

Code Page Traversal Strategy	Average Bytes Disclosed	# of Attempts to Find One Gadget
Breadth-first	657	25
Depth-first	299	14

6.2 MemoryTrap-Aware Disclosure Strategies

If attackers become aware that MemoryTrap has been deployed, they may adopt more sophisticated code disclosure strategies to bypass the inserted booby traps. For instance, attackers might abandon continuous code disclosure and instead attempt to read M bytes of code at intervals of N bytes, or initiate disclosure from leaked code pointers located in the stack or heap. Therefore, we conducted additional MemoryTrap-aware experiments using the same binary described in §6.1 to evaluate attackers' capability to disclose code using these advanced strategies. Specifically, we attempted to read 50-byte code at intervals of 200 bytes, 300 bytes, and 500 bytes. Under the breadth-first search approach, the average amounts of disclosed code were 636 bytes, 649 bytes, and 678 bytes, respectively. Under the depth-first search approach, the corresponding averages were 319 bytes, 290 bytes, and 300 bytes. We also collected code pointers and initiated code disclosure directly from these pointers. Using this approach, breadth-first and depth-first searches triggered booby traps after disclosing an average of 672 bytes and 451 bytes, respectively. All the results presented above are consistent with those shown in Table 2, indicating that under the protection of MemoryTrap, the choice of code disclosure strategy does not significantly affect the probability of triggering a booby trap. In fact, because booby traps are opaque to attackers, the probability that any given disclosure strategy triggers a booby trap follows a hypergeometric distribution.

6.3 Retrofitting XoM for JIT Compiled Code

In this experiment, we evaluate MemoryTrap's protection on JIT compiled code by exploiting a recent vulnerability in V8 JIT engine, CVE-2020-16040 [84]. We modify the vulnerable version of V8 and compile it with the latest version of

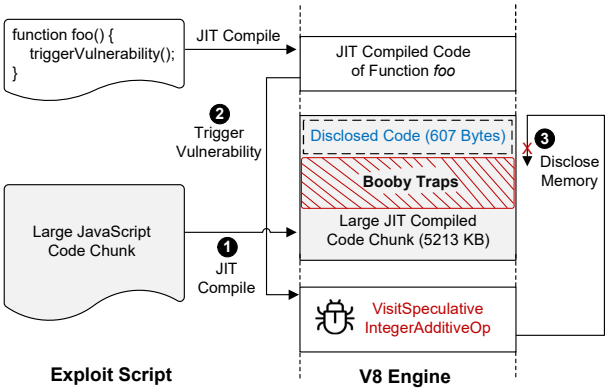


Figure 6: Using exploit of CVE-2020-16040 to disclose memory of JIT compiled code protected by MemoryTrap.

Chromium browser [85] to get a custom Chromium browser. This custom browser can trigger CVE-2020-16040 and generate MemoryTrap-protected JIT code. After that, we use the exploit of CVE-2020-16040 to trigger this vulnerability and start to disclose the generated JIT code.

CVE-2020-16040 is an integer overflow vulnerability caused by missing parameter type checking during the JIT optimization phase, resulting in arbitrary memory read and write. Figure 6 shows how we use the exploit to disclose the code of JIT compiled JavaScript functions. We first write a JavaScript function and a large JavaScript code chunk and trigger the JIT compile mechanism of V8 (❶ in Figure 6) to get a JIT compiled code chunk with the size of 5,213 KB. The modified JIT engine inserted a total of 7,025 booby traps into this large JIT compiled code chunk. When compiling the `foo` function, the `foo` function can trigger the vulnerability in "VisitSpeculativeIntegerAdditiveOp" function (❷ in Figure 6), which is a function of V8 engine used to optimize JIT compiled code. This vulnerability gives us the arbitrary read capability, so we try to disclose the code in the large JIT compiled code chunk. We use the breadth-first strategy to disclose code pages of JIT compiled code, because in Table 2 we found that the breadth-first strategy can disclose more code. When disclosing the JIT compiled code, the exploit triggers booby traps in a very short time. The exploit script only discloses 607 bytes of code before falling into booby traps. Such 607 bytes is the distance from the beginning of the large JIT compiled code chunk to the beginning of the first booby trap, as shown in ❸ of Figure 6.

To determine the amount of JIT compiled code that must be disclosed to find the whole gadget chain, we search for the necessary gadgets in the absence of MemoryTrap's protection. The findings indicate that a staggering 2,739 KB of code must be disclosed to successfully identify the entire chain of gadgets, rendering the previously estimated 607 bytes woefully insufficient for this task.

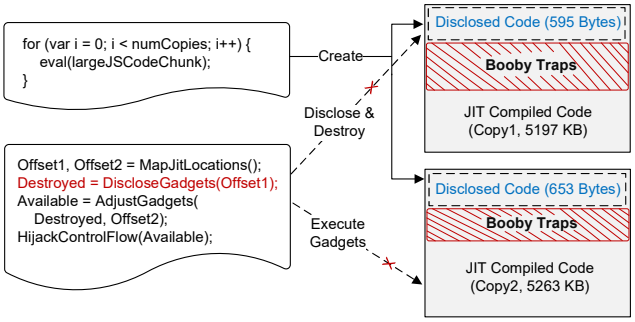


Figure 7: JavaScript JIT cloning attack maintains two copies, but our booby traps spread over in each copy.

6.4 Countering Code Inference Attacks

The first three types of code inference attacks described in [57] operate by maintaining multiple copies of code and inferring the contents of one copy through the disclosure of another. To evaluate the effectiveness of MemoryTrap against these three types of code inference attacks, we use the JIT-compiled code cloning attack as a representative example. We follow the detailed description in Snow et al.'s paper [57] to reproduce the JavaScript JIT compiled code cloning attack. We maintain two copies of a large JIT-compiled code chunk, each over 5MB in size, using the same JavaScript source code as in §6.3. The JIT engine compiles the first copy of code in 5,197KB, and it compiles the second copy of code in 5,263KB, with 7,025 booby traps inserted in each of them. Each copy has a different size due to the variable-size booby traps inserted. We also use the breadth-first code page traversal strategy to disclose these code chunks. Figure 7 illustrates how MemoryTrap stops this type of attack. No matter how many code copies the adversary maintains, our booby traps spread over each code copy. Similar to the evaluation result shown in Figure 6, our code page traversal attempt can only disclose a very small piece of code: 595 bytes in the first copy and 653 bytes in the second copy, before the code read operations were caught in a booby trap. Such little disclosed code is not enough to construct a malicious payload.

The last type of code inference attacks reveals certain portions of code in order to infer the content of related code and thereby evade detection or destruction. MemoryTrap inherently defends against this type of attack by distributing a large number of traps throughout the code, which significantly alters the original code layout. This approach invalidates the attacker's prior knowledge about the code structure, making it difficult to infer the actual content of the code.

7 Performance Evaluation

We tested standard benchmarks and a set of real-world applications to measure the slowdowns caused by MemoryTrap. We run microbenchmarks on the modified kernel to evaluate the overhead caused by kernel modification. We use

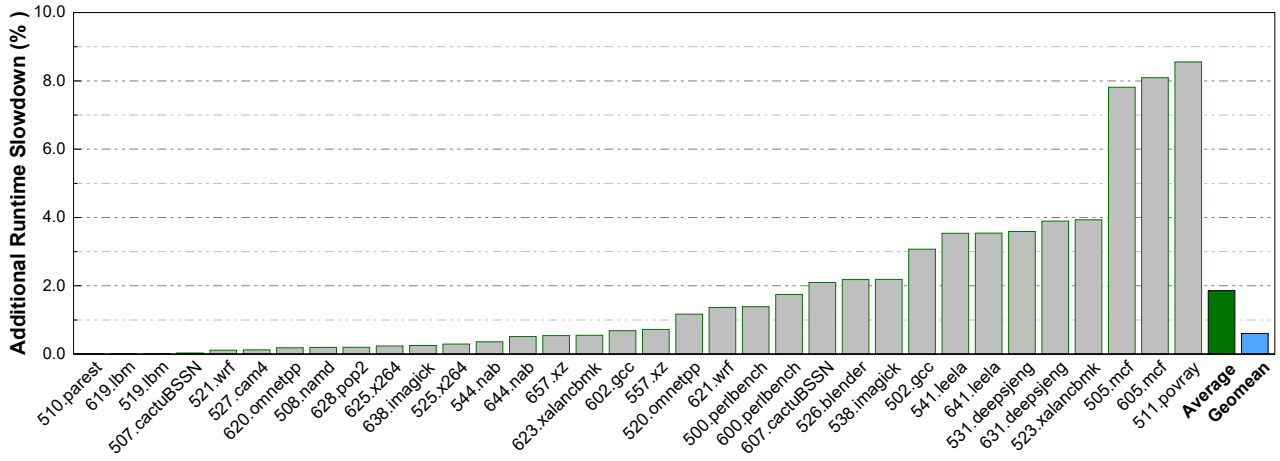


Figure 8: Additional runtime slowdown (%) of MemoryTrap on SPEC CPU 2017 (*ref* workload).

macrobenchmarks to evaluate the overhead of MemoryTrap on CPU-intensive programs. For real-world applications, we use mainstream web servers (Nginx-1.20.1, Apache-2.4.49, and Lighttpd-1.4.59), four popular databases (MySQL-8.0.26, MongoDB-4.2.17, Redis-6.2.5, and SQLite-3.36.0), and a custom V8 engine. Please note that in subsequent experiments, we run the standard version of benchmark or application on the standard kernel and the MemoryTrap protected version on the custom kernel. For the shared libraries that the real-world applications depend on (e.g., PCRE and zlib for Nginx and APR for Apache), we use their MemoryTrap protected version for the performance evaluation.

7.1 Microbenchmarks

MemoryTrap modifies the Linux kernel's binary loader, page fault handler, and process context structure. To evaluate the impact of these modifications on kernel operations, we run lmbench [86] on both the standard Linux kernel and our modified version, and compare the operation latencies of the standard kernel and the modified kernel.

Table 3 displays the running time of kernel operations related to process creation and page fault handling. The *Fork Proc* and *Exec Proc* are process creation operations that use *fork* and *exec*. These operations resulted in an overhead of 0.88% and 0.58%, respectively, due to the additional steps required for loading XoM metadata. Page Fault shows the overhead related to page fault handling, with a 0.52% overhead. Prot Fault denotes the overhead related to protection fault handling. MemoryTrap needs to scrutinize whether the protection fault is enabled, causing a 0.83% overhead.

In order to store MemoryTrap's metadata, such as MPK's PKey and executable data list, we included additional data members in the process context structure. However, this modification may result in overhead during context switches in the kernel. To evaluate the impact of our modification, we conducted experiments measuring the context switching time for two kernel versions, and results are presented in Table 4. The upper half of the first row displays the number of processes involved in the context switches, while the lower half shows the memory usage of each process. For instance, "2p/16k" represents a context switch between two processes, each utilizing 16 KB of memory. All entries exhibit overhead values distributed around zero, suggesting that MemoryTrap exerts an insignificant impact on the performance of kernel context switches. For additional lmbench results with limited correlation to MemoryTrap, please refer to Appendix C.

Table 3: Time for kernel operations related to process creation and page fault handling (in μs). Smaller is better. In the first row are the names of benchmarks in lmbench.

Kernel	Fork Proc	Exec Proc	Page Fault	Prot Fault
Standard	113	347	0.776	0.481
MemoryTrap	114	349	0.780	0.485
Overhead	0.88%	0.58%	0.52%	0.83%

7.2 Macrobenchmarks

SPEC CPU2017 is the latest generation of SPEC CPU benchmark with larger and more complex workloads than SPEC 2006. In the performance evaluation of macrobenchmarks, we compile the standard version of SPEC 2017 and the MemoryTrap protected version of SPEC 2017, and run them with the *ref* workload on our test machine. We take the running time of standard versions as the baseline to measure the overhead incurred by MemoryTrap's protection. We removed the pure Fortran benchmarks from our experiment, because LLVM's Fortran frontend is still under testing and not ready for developers to use [87].

Figure 8 shows the overhead of MemoryTrap protection for each benchmark in SPEC CPU 2017. The green and blue bars on the rightmost side show the average and the geometric mean value of overhead, respectively. From Figure 8, we can see that seven overhead values are very close to zero, and the peak value (8.5%) happens in `povray`. Upon further

Table 4: Context switch time (in μs). Smaller is better.

Kernel	2p	2p	2p	8p	8p	16p	16P
	0K	16K	64K	16K	64K	16K	64K
Standard	2.09	2.10	2.61	2.43	2.59	2.64	2.83
MemoryTrap	2.13	2.15	2.48	2.49	2.62	2.66	2.79
Overhead	1.91%	2.38%	-4.24%	2.47%	1.16%	0.76%	-1.43%

investigation, `povray` has a lot of dispatch loop structures that call numerous small handler functions. According to our booby trap insertion strategy, all of these handler functions are inserted with one booby trap. `Povray` spends most of its running time on executing handler functions, so the accumulated overhead is relatively high. `505.mcf` and `605.mcf` are both derived from `MCF`, a program used for single-depot vehicle scheduling in public mass transportation. Similar to `povray`, `MCF` spends most of its running time on iteratively executing three utility functions (getArcPosition, arc_compare, and spec_qsort), causing a relatively high overhead. Nonetheless, the average overhead of tested SPEC benchmarks is 1.85%, and the geometric mean is 0.60%, indicating a limited performance impact on CPU-intensive programs.

Real-World Applications In addition to SPEC CPU 2017, we also evaluated the performance of MemoryTrap using web servers, databases, and a JIT engine. For web servers, we tested Nginx, Apache, and Lighttpd. The maximum overhead observed for web servers is only 2.11%, with an average overhead of 0.74%; most overhead values are very close to zero. Regarding databases, we evaluated MySQL, MongoDB, Redis, and SQLite, and the average overhead incurred by these databases is merely 1.30%. Then we use Kraken Benchmark [64] to measure the overhead incurred by MemoryTrap, and the average overhead is only 1.54%. Due to the page limit, we put detailed performance results in Appendix B. We have also conducted experiments to evaluate MemoryTrap's disk and memory overhead. Please refer to Appendix G for details.

Worst-Case Benchmark We employ a worst-case benchmark that cyclically reads the code area for 10,000,000 times to trigger extensive embedded data accesses, causing the program to spend most of its execution time performing embedded data reads. This benchmark allows us to evaluate the performance of MemoryTrap under worst-case conditions. We compare the time required to complete this benchmark with and without the deployment of MemoryTrap. With MemoryTrap, the benchmark takes 3.23X longer to complete than without it, which means that MemoryTrap incurs a **3.23** times slowdown in this worst-case benchmark. However, the overall system performance is not significantly impacted by this seemingly unacceptable overhead. This is attributed to the fact that legitimate data-in-code reads are interspersed with numerous other instructions. A more in-depth analysis of this performance impact will be presented in §7.3.

7.3 Overhead Cause Analysis

There are two sources of performance overhead for MemoryTrap: the data-in-code read legitimacy check and the jump instructions introduced by booby traps. Next, we break down each component of MemoryTrap to provide insights into the causes of MemoryTrap's overhead.

As discussed in §7.1, the data-in-code read legitimacy check incurs a 3.23 times performance overhead. However, MemoryTrap causes only a 1.85% overhead on SPEC CPU 2017 and a 0.74% overhead on web servers, which is negligible compared to the read legitimacy check. Our further investigation reveals that this is because legitimate data-in-code reads are interspersed among numerous other instructions. We introduce the term "Read Intensity" here, defined by Equation 1, to assess the density of data-in-code read operations among all instructions.

$$Read\ Intensity = \frac{\#\ of\ Legitimate\ Reads}{\#\ of\ Executed\ Instructions} \quad (1)$$

We have gathered data on legitimate data reads and the total number of executed instructions for SPEC CPU 2017, web servers, and databases used in our evaluation. Additionally, we gathered the same information for OpenSSL, which contains significantly more data-in-code reads than other programs due to the handwritten implementation of cryptographic algorithms. The results show that the Read Intensities are 7.0E^{-11} for SPEC CPU 2017, 1.64E^{-10} for web servers, and 4.80E^{-12} for databases. Even for OpenSSL, it only has a Read Intensity of 1.40E^{-7}, which implies that one data-in-code read is performed for every million instructions executed. The average Read Intensity across all programs is 3.51E^{-8}, signifying that, on average, one data-in-code read is performed after executing ten million instructions. This illustrates that while data-in-code reads are not rare in practice, their frequency is extremely low, which significantly mitigates the impact of the legitimacy check overhead. We conducted another experiment to evaluate the overhead caused by the read legitimacy check. We ran each benchmark without booby trap insertion and found that the read legitimacy check only causes a 0.45% runtime slowdown on average. This indicates that most of MemoryTrap's overhead is caused by the jump instructions introduced by booby trap insertion.

7.4 Performance Comparison

We compare the performance of MemoryTrap with representative approaches shown in Figure 1, including XnR [46], Readactor [47], HideM [49], and Heisenbyte [53]. However, we encountered a major challenge that none of the papers listed in Figure 1 have made their tools publicly available, with the exception of MemoryTrap. In light of this, we compared their SPEC CPU 2006 performance data reported in their papers with ours. Figure 9 shows the performance comparison result. XnR is the first XoM approach; it implements

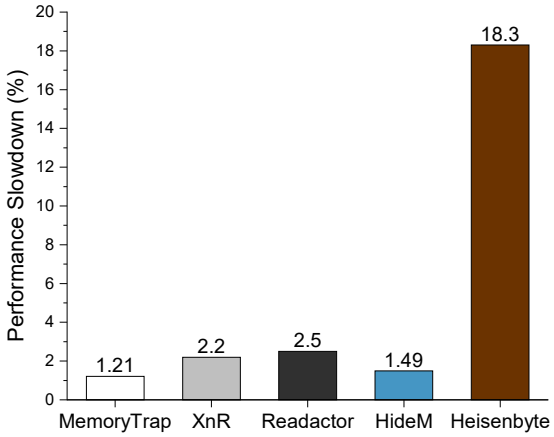


Figure 9: Performance comparison with XnR [46], Readactor [47], HideM [49], and Heisenbyte [53].

XoM through software emulation, incurring 2.2% performance overhead. Readactor uses Intel's virtualization technique to enforce the execute-only permission, exhibiting 2.2% overhead. HideM utilities the split-TLB to achieve XoM protection, which causes 1.4% overhead. Heisenbyte tries to tolerate legitimate data-in-code reads, but it reveals significantly higher overhead (18.3%) than other approaches due to its code destruction and read request redirection operations. In contrast, MemoryTrap exhibits the lowest overhead (1.21%) among all prototypes in Figure 9.

7.5 Compatibility with Embedded Data Reads

Among real-world programs we tested, OpenSSL exhibited the highest frequency of embedded data reads. Therefore, we use OpenSSL to demonstrate the compatibility of MemoryTrapwith embedded data reads. We compiled a MemoryTrap-protected version of OpenSSL and executed it using the `openssl speed` command [88], which is the official benchmarking tool used to measure the performance of cryptographic algorithms in OpenSSL. The compiled OpenSSL binary contains a total of 827,604 bytes of embedded data, distributed across 1,063 embedded data blocks. During the execution of the `openssl speed` command, a total of 656,509,327 embedded data read operations occurred, and the read intensity is $1.40E^{-7}$. All embedded data read operations were executed successfully, and all benchmark tests completed normally. This result demonstrates that MemoryTrap has excellent compatibility with embedded data reads. Notably, although OpenSSL contains a considerable amount of embedded data, the average size of each embedded data block is only 779 bytes. This implies that even if an attacker successfully locates an embedded data block, they can only access, on average, 779 bytes of embedded data. Such a limited amount of embedded data is insufficient to be successfully leveraged by a JIT-ROP attacker.

8 Discussion and Conclusion

Potential Security Problem of WRPKRU In the MPK mechanism, the WRPKRU instruction can write the value in EAX to PKRU to change the permission of page groups. Skilled attackers may try to bypass MemoryTrap's protection by granting read permissions to all page groups using the WRPKRU instruction. However, when executing the WRPKRU instruction, the value of ECX and EDX must be 0. Otherwise, a general-protection exception (#GP) will occur. If attackers change the page permission via WRPKRU, at least four gadgets are needed: 1) write EAX, 2) write ECX, 3) write EDX, and 4) execute WRPKRU. Fortunately, as we demonstrated in §6, with MemoryTrap deployed, at most one gadget can be disclosed before triggering MemoryTrap's protection.
DoS Risk In our current response to a detected memory disclosure attempt, MemoryTrap will terminate the vulnerable process immediately. While this measure can thwart attackers from gaining control of the system, it also poses a potential risk of Denial of Service (DoS). An alternative strategy is to combine MemoryTrap with re-randomization. Instead of terminating the compromised process, we activate a new randomization process to the code layout. This method can prevent the use of collected gadgets and simultaneously avoid the risk of DoS. However, the cost of doing so is the large overhead that comes with the re-randomization process itself.
Multi-Platform Support Currently, MemoryTrap is implemented based on Intel's MPK mechanism. MPK is supported on mainstream Intel and AMD desktop and server CPUs but is unavailable on other platforms. Nevertheless, other platforms provide similar mechanisms that can be used to implement MemoryTrap. For instance, ARM provides AP/XN [89], RISC-V offers PMP [90], and PowerPC includes Protection Keys [91]. These mechanisms can serve as alternatives to implement MemoryTrap. We discuss details in Appendix D.
Conclusion This paper introduces MemoryTrap, a hardware-assisted technique to counter memory disclosure attacks. MemoryTrap inserts unreadable, variable-size code snippets (booby traps) at strategic points to block unauthorized code page reads, utilizing Intel's MPK for efficient, fine-grained memory permission control. Our experiments demonstrate that MemoryTrap reveals a strong resistance to various memory disclosure attempts and negligible runtime overhead.

Acknowledgment

We thank our shepherd, Dr. Pierre Olivier, and all anonymous reviewers for their valuable comments to improve this paper. This work is supported by the National Nature Science Foundation of China under Grant No. 62272351, 61972297, and 62172308. Jiang Ming is supported by NSF grants 2312185 & 2417055, Google Research Scholar Award, and Tulane COR Fellowships. Dongpeng Xu is supported by NSF grants 2211905 and 2022279.

References

[1] László Szekeres, Mathias Payer, Tao Wei, and Dawn Song. SoK: Eternal War in Memory. In *Proceedings of the 34th IEEE Symposium on Security and Privacy (S&P'13)*, 2013.

[2] Volodymyr Kuznetzov, László Szekeres, Mathias Payer, George Candea, R Sekar, and Dawn Song. Code-Pointer Integrity. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI'14)*, 2014.

[3] Christian DeLozier, Kavya Lakshminarayanan, Gilles Pokam, and Joseph Devietti. Hurdle: Securing Jump Instructions Against Code Reuse Attacks. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'20)*, 2020.

[4] Haotian Zhang, Mengfei Ren, Yu Lei, and Jiang Ming. One Size Does Not Fit All: Security Hardening of MIPS Embedded Systems via Static Binary Debloating for Shared Libraries. In *Proceedings of the 27th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'22)*, 2022.

[5] Hovav Shacham. The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS'07)*, 2007.

[6] Martín Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. A theory of secure control flow. In *International Conference on Formal Engineering Methods*, pages 111–124. Springer, 2005.

[7] Martín Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information and System Security (TISSEC)*, 13(1):1–40, 2009.

[8] Jonathan Corbet. A new LLVM CFI implementation. https://lwn.net/Articles/898040/, June 17, 2022.

[9] Yueqiang Cheng, Zongwei Zhou, Yu Miao, Xuhua Ding, and Robert H Deng. ROPecker: A Generic and Practical Approach for Defending against ROP Attack. In *Proceedings of the 21st Annual Network and Distributed System Security Symposium (NDSS'14)*, 2014.

[10] Ali Jose Mashtizadeh, Andrea Bittau, Dan Boneh, and David Mazières. CCFI: Cryptographically Enforced Control Flow Integrity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS '15)*, pages 941–951, 2015.

[11] Ivan Fratrić. Ropguard: Runtime prevention of return-oriented programming attacks. *Technical report*, 2012.

[12] Vasilis Pappas, Michalis Polychronakis, and Angelos D Keromytis. Transparent rop exploit mitigation using indirect branch tracing. In *Proceedings of the 22nd USENIX Security Symposium (USENIX Security '13)*, pages 447–462, 2013.

[13] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, Laszlo Szekeres, Stephen McCamant, Dawn Song, and Wei Zou. Practical control flow integrity and randomization for binary executables. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy (S&P '13)*, pages 559–573. IEEE, 2013.

[14] Mingwei Zhang and R Sekar. Control flow integrity for cots binaries. In *Proceedings of the 22nd USENIX Security Symposium (USENIX Security '13)*, pages 337–352, 2013.

[15] Nathan Burow, Scott A Carr, Joseph Nash, Per Larsen, Michael Franz, Stefan Brunthaler, and Mathias Payer. Control-Flow integrity: Precision, security, and performance. *ACM Computing Surveys (CSUR)*, 50(1):1–33, 2017.

[16] Intel. Control Flow Enforcement Technology (CET). https://www.intel.com/content/dam/develop/external/us/en/documents/catc17-introduction-intel-cet-844137.pdf, [online].

[17] Qualcomm Technologies. Pointer Authentication on ARMv8.3. https://www.qualcomm.com/content/dam/qcomm-martech/dm-assets/documents/pointer-auth-v7.pdf, [online].

[18] Jianhao Xu, Luca Di Bartolomeo, Flavio Toffalini, Bing Mao, and Mathias Payer. WarpAttack: Bypassing CFI through Compiler-Introduced Double-Fetches. In *Proceedings of the 2023 IEEE Symposium on Security and Privacy (S&P '23)*, pages 1271–1288, 2023.

[19] Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. Counterfeit Object-Oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy (S&P '15)*, pages 745–762. IEEE, 2015.

[20] Nicholas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R Gross. Control-Flow Bending: On the Effectiveness of Control-Flow Integrity. In *Proceedings of the 24th USENIX Security Symposium (USENIX Security '15)*, pages 161–176, 2015.

[21] Nicholas Carlini and David Wagner. Rop is still dangerous: Breaking modern defenses. In *23rd USENIX Security Symposium (USENIX Security '14)*, pages 385–399, 2014.

[22] Lucas Davi, Ahmad-Reza Sadeghi, Daniel Lehmann, and Fabian Monrose. Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In *Proceedings of the 23rd USENIX Security Symposium (USENIX Security '14)*, pages 401–416, 2014.

[23] Enes Göktas, Elias Athanasopoulos, Herbert Bos, and Georgios Portokalidis. Out of control: Overcoming control-flow integrity. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy (S&P '14)*, pages 575–589. IEEE, 2014.

[24] Enes Göktaş, Elias Athanasopoulos, Michalis Polychronakis, Herbert Bos, and Georgios Portokalidis. Size does matter: Why using gadget-chain length to prevent code-reuse attacks is hard. In *Proceedings of the 23rd USENIX Security Symposium (USENIX Security '14)*, pages 417–432, 2014.

[25] Felix Schuster, Thomas Tendyck, Jannik Pewny, Andreas Maaß, Martin Steegmanns, Moritz Contag, and Thorsten Holz. Evaluating the effectiveness of current anti-rop defenses. In *International Workshop on Recent Advances in Intrusion Detection*, pages 88–108. Springer, 2014.

[26] Matteo Malvica. Bypassing Intel CET with Counterfeit Objects. https://www.offsec.com/offsec/bypassing-intel-cet-with-counterfeit-objects/, [online].

[27] Marcoramilli. From ROP to LOP Bypassing Control FLow Enforcement. https://shorturl.at/jDJM1, [online].

[28] Sabine Houy and Alexandre Bartel. Lessons Learned and Challenges of Deploying Control Flow Integrity in Complex Software: The Case of OpenJDK's Java Virtual Machine. In *Proceedings of the 2024 IEEE Secure Development Conference (SecDev '24)*, 2024.

[29] Arif Ali Mughal. The Art of Cybersecurity: Defense in Depth Strategy for Robust Protection. *International Journal of Intelligent Automation and Computing*, 1(1):1–20, 2018.

[30] Michael Backes and Stefan Nürnberger. Oxymoron: Making Fine-Grained Memory Randomization Practical by Allowing Code Sharing. In *Proceedings of the 23rd USENIX Conference on Security Symposium (USENIX Security'14)*, 2014.

[31] Lucas Davi, Christopher Liebchen, Ahmad-Reza Sadeghi, Kevin Z Snow, and Fabian Monrose. Isomeron: Code Randomization Resilient to (Just-In-Time) Return-Oriented Programming. In *Proceedings of the 22nd Annual Network and Distributed System Security Symposium (NDSS'15)*, 2015.

[32] PaX Team. Address Space Layout Randomization (ASLR). https://pax.grsecurity.net/docs/aslr.txt, 2003.

[33] Sandeep Bhatkar, Daniel C DuVarney, and Ron Sekar. Address Obfuscation: An Efficient Approach to Combat a Broad Range of Memory Error Exploits. In *Proceedings of the 12th Conference on USENIX Security Symposium (USENIX Security'03)*, 2003.

[34] Sandeep Bhatkar, Daniel C DuVarney, and R Sekar. Efficient Techniques for Comprehensive Protection from Memory Error Exploits. In *Proceedings of the 14th Conference on USENIX Security Symposium (USENIX Security'05)*, 2005.

[35] Kjell Braden, Lucas Davi, Christopher Liebchen, Ahmad-Reza Sadeghi, Stephen Crane, Michael Franz, and Per Larsen. Leakage-Resilient Layout Randomization for Mobile Devices. In *Proceedings of the 23rd Annual Network and Distributed System Security Symposium (NDSS'16)*, 2016.

[36] Cristiano Giuffrida, Anton Kuijsten, and Andrew S Tanenbaum. Enhanced Operating System Security Through Efficient and Fine-grained Address Space Randomization. In *Proceedings of the 21st Conference on USENIX Security Symposium (USENIX Security'12)*, 2012.

[37] Jason Hiser, Anh Nguyen-Tuong, Michele Co, Matthew Hall, and Jack W Davidson. ILR: Where'd My Gadgets Go? In *Proceedings of the 33rd IEEE Symposium on Security and Privacy (S&P'12)*, 2012.

[38] Andrei Homescu, Steven Neisius, Per Larsen, Stefan Brunthaler, and Michael Franz. Profile-guided Automated Software Diversity. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO'13)*, 2013.

[39] Chongkyung Kil, Jinsuk Jun, Christopher Bookholt, Jun Xu, and Peng Ning. Address Space Layout Permutation (ASLP): Towards Fine-Grained Randomization

of Commodity Software. In *Proceedings of the 22nd Annual Computer Security Applications Conference (ACSAC'06)*, 2006.

[40] Vasilis Pappas, Michalis Polychronakis, and Angelos D Keromytis. Smashing the Gadgets: Hindering Return-Oriented Programming Using In-place Code Randomization. In *Proceedings of the 33rd IEEE Symposium on Security and Privacy (S&P'12)*, 2012.

[41] Richard Wartell, Vishwath Mohan, Kevin W Hamlen, and Zhiqiang Lin. Binary Stirring: Self-randomizing Instruction Addresses of Legacy x86 Binary Code. In *Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS'12)*, 2012.

[42] David Williams-King, Graham Gobieski, Kent Williams-King, James P Blake, Xinhao Yuan, Patrick Colp, Michelle Zheng, Vasileios P Kemerlis, Junfeng Yang, and William Aiello. Shuffler: Fast and Deployable Continuous Code Re-Randomization. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*, 2016.

[43] Raoul Strackx, Yves Younan, Pieter Philippaerts, Frank Piessens, Sven Lachmund, and Thomas Walter. Breaking the Memory Secrecy Assumption. In *Proceedings of the 2nd European Workshop on Systems Security (EUROSEC'09)*, 2009.

[44] Kevin Z Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, and Ahmad-Reza Sadeghi. Just-In-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization. In *Proceedings of the 34th IEEE Symposium on Security and Privacy (S&P'13)*, 2013.

[45] Salman Ahmed, Ya Xiao, Kevin Z Snow, Gang Tan, Fabian Monrose, and Danfeng Yao. Methodologies for Quantifying (Re-)randomization Security and Timing under JIT-ROP. In *Proceedings of the 27th ACM Conference on Computer and Communications Security (CCS'20)*, 2020.

[46] Michael Backes, Thorsten Holz, Benjamin Kollenda, Philipp Koppe, Stefan Nürnberger, and Jannik Pewny. You Can Run but You Can't Read: Preventing Disclosure Exploits in Executable Code. In *Proceedings of the 21st ACM Conference on Computer and Communications Security (CCS'14)*, 2014.

[47] Stephen Crane, Christopher Liebchen, Andrei Homescu, Lucas Davi, Per Larsen, Ahmad-Reza Sadeghi, Stefan Brunthaler, and Michael Franz. Readactor: Practical Code Randomization Resilient to Memory Disclosure. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (S&P'15)*, pages 763–780. IEEE, 2015.

[48] Stephen J Crane, Stijn Volckaert, Felix Schuster, Christopher Liebchen, Per Larsen, Lucas Davi, Ahmad-Reza Sadeghi, Thorsten Holz, Bjorn De Sutter, and Michael Franz. It's a TRaP: Table Randomization and Protection against Function-Reuse Attacks. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS'15)*, 2015.

[49] Jason Gionta, William Enck, and Peng Ning. HideM: Protecting the Contents of Userspace Memory in the Face of Disclosure Vulnerabilities. In *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy (CODASPY'15)*, 2015.

[50] Xiaozhu Meng and Barton P. Miller. Binary Code is Not Easy. In *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA'16)*, 2016.

[51] Manuel Rigger, Stefan Marr, Stephen Kell, David Leopoldseder, and Hanspeter Mössenböck. An Analysis of x86-64 Inline Assembly in C Programs. In *Proceedings of the 14th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '18)*, 2018.

[52] Chengbin Pang and Ruotong Yu and Yaohui Chen and Eric Koskinen and Georgios Portokalidis and Bing Mao and Jun Xu. SoK: All You Ever Wanted to Know About x86/x64 Binary Disassembly But Were Afraid to Ask. In *Proceedings of the 42nd IEEE Symposium on Security and Privacy (S&P'21)*, 2021.

[53] Adrian Tang, Simha Sethumadhavan, and Salvatore Stolfo. Heisenbyte: Thwarting Memory Disclosure Attacks using Destructive Code Reads. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS'15)*, 2015.

[54] Jan Werner, George Baltas, Rob Dallara, Nathan Otterness, Kevin Z Snow, Fabian Monrose, and Michalis Polychronakis. No-Execute-After-Read: Preventing Code Disclosure in Commodity Software. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security (ASIACCS'16)*, 2016.

[55] Chris Wilson and Lita Arriaga. Secure Commodity Software by Enforcing Memory Permission Policies. https://www.academia.edu/download/50294358/Transparent_Enforcement_for_Software_Memory_Security.pdf, [online].

[56] Jannik Pewny, Philipp Koppe, Lucas Davi, and Thorsten Holz. Breaking and Fixing Destructive

Code Read Defenses. In *Proceedings of the 33rd Annual Computer Security Applications Conference (ACSAC'17)*, 2017.

[57] Kevin Z Snow, Roman Rogowski, Jan Werner, Hyungjoon Koo, Fabian Monrose, and Michalis Polychronakis. Return to the Zombie Gadgets: Undermining Destructive Code Reads via Code Inference Attacks. In *Proceedings of the 37th IEEE Symposium on Security and Privacy (S&P'16)*, 2016.

[58] Stephen Crane, Per Larsen, Stefan Brunthaler, and Michael Franz. Booby Trapping Software. In *Proceedings of the 2013 New Security Paradigms Workshop (NSPW'13)*, 2013.

[59] Mohammed H. Almeshekah and Eugene H. Spafford. *Cyber Security Deception*, pages 23–50. Springer International Publishing, Cham, 2016.

[60] Xiao Han, Nizar Kheir, and Davide Balzarotti. Deception Techniques in Computer Security: A Research Perspective. *ACM Computing Surveys*, 51(4), July 2018.

[61] Intel. Intel® 64 and IA-32 Architectures Software Developer's Manual. http://tiny.cc/7vdhuz, [online].

[62] Soyeon Park, Sangho Lee, Wen Xu, Hyungon Moon, and Taesoo Kim. libmpk: Software Abstraction for Intel Memory Protection Keys (Intel MPK). In *Proceedings of the 2019 USENIX Annual Technical Conference (USENIX ATC'19)*, 2019.

[63] Standard Performance Evaluation Corporation. SPEC CPU® 2017. https://www.spec.org/cpu2017/, 2021.

[64] Mozilla. Kraken JavaScript Benchmark. https://mozilla.github.io/krakenbenchmark.mozilla.org/index.html, [online].

[65] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. Return-Oriented Programming: Systems, Languages, and Applications. *ACM Transactions on Information and System Security*, 15(1), March 2012.

[66] Jason Gionta, William Enck, and Per Larsen. Preventing Kernel Code-Reuse Attacks Through Disclosure Resistant Code Diversification. In *Proceedings of 2016 IEEE Conference on Communications and Network Security (CNS'16)*, 2016.

[67] Marios Pomonis, Theofilos Petsios, Angelos D Keromytis, Michalis Polychronakis, and Vasileios P Kemerlis. kRˆX: Comprehensive Kernel Protection Against Just-In-Time Code Reuse. In *Proceedings of the Twelfth European Conference on Computer Systems (EuroSys'17)*, 2017.

[68] Robert N.M. Watson, Jonathan Woodruff, Peter G. Neumann, Simon W. Moore, Jonathan Anderson, David Chisnall, Nirav Dave, Brooks Davis, Khilan Gudka, Ben Laurie, Steven J. Murdoch, Robert Norton, Michael Roe, Stacey Son, and Munraj Vadera. CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (S&P'15)*, 2015.

[69] Zhe Wang, Chenggang Wu, Mengyao Xie, Yinqian Zhang, Kangjie Lu, Xiaofeng Zhang, Yuanming Lai, Yan Kang, and Min Yang. SEIMI: Efficient and Secure SMAP-Enabled Intra-process Memory Isolation. In *Proceedings of the 41st IEEE Symposium on Security and Privacy (S&P'20)*, 2020.

[70] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O. Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. ERIM: Secure, Efficient In-process Isolation with Protection Keys (MPK). In *Proceedings of the 28th Conference on USENIX Security Symposium (USENIX Security'19)*, 2019.

[71] Mohammad Hedayati, Spyridoula Gravani, Ethan Johnson, John Criswell, Michael L Scott, Kai Shen, and Mike Marty. Hodor: Intra-Process Isolation for High-Throughput Data Plane Libraries. In *Proceedings of the 2019 USENIX Annual Technical Conference (USENIX ATC'19)*, 2019.

[72] Nathan Burow, Xinping Zhang, and Mathias Payer. SoK: Shining Light on Shadow Stacks. In *Proceedings of the 40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

[73] Xuancheng Jin, Xuangan Xiao, Songlin Jia, Wang Gao, Hang Zhang, Dawu Gu, Siqi Ma, Zhiyun Qian, and Juanru Li. Annotating, Tracking, and Protecting Cryptographic Secrets with CryptoMPK. In *Proceedings of the 43rd IEEE Symposium on Security and Privacy (S&P'22)*, 2022.

[74] Spyridoula Gravani, Mohammad Hedayati, John Criswell, and Michael L Scott. Fast Intra-kernel Isolation and Security with IskiOS. In *Proceedings of the 24th International Symposium on Research in Attacks, Intrusions and Defenses (RAID'21)*, 2021.

[75] Michalis Athanasakis, Elias Athanasopoulos, Michalis Polychronakis, Georgios Portokalidis, and Sotiris Ioannidis. The Devil is in the Constants: Bypassing Defenses in Browser JIT Engines. In *Proceedings of the 22nd Annual Network and Distributed System Security Symposium (NDSS'15)*, 2015.

[76] Chengyu Song, Chao Zhang, Tielei Wang, Wenke Lee, and David Melski. Exploiting and Protecting Dynamic Code Generation. In *Proceedings of the 22nd Annual Network and Distributed System Security Symposium (NDSS'15)*, 2015.

[77] Edward J Schwartz, Thanassis Avgerinos, and David Brumley. Q: Exploit hardening made easy. In *Proceedings of the 20th USENIX Conference on Security Symposium (USENIX Security'11)*, 2011.

[78] Andrei Homescu, Michael Stewart, Per Larsen, Stefan Brunthaler, and Michael Franz. Microgadgets: Size Does Matter in Turing-Complete Return-Oriented Programming. In *Proceedings of the 6th USENIX Workshop on Offensive Technologies (WOOT '12)*, 2012.

[79] Hyungjoon Koo, Yaohui Chen, Long Lu, Vasileios P Kemerlis, and Michalis Polychronakis. Compiler-Assisted Code Randomization. In *Proceedings of the 39th IEEE Symposium on Security and Privacy (S&P'18)*, 2018.

[80] The MITRE Corporation. CVE-2013-2028 Detail. https://www.cve.org/CVERecord?id=CVE-2013-2028, [online].

[81] Salman Ahmed. JITROP-Native. https://github.com/salmanyam/jitrop-native, [online].

[82] Rapid 7. Metasploit. https://www.metasploit.com/, [online].

[83] Offensive Security. Exploit Databse. https://www.exploit-db.com/, [online].

[84] The MITRE Corporation. CVE-2020-16040 Detail. https://www.cve.org/CVERecord?id=CVE-2020-16040, [online].

[85] Google. The Chromium Projects. https://www.chromium.org/, [online].

[86] Larry McVoy. lmbench. http://lmbench.sourceforge.net/run_lmbench.html, [online].

[87] LLVM. Welcome to Flang's documentation. https://flang.llvm.org/docs/, [online].

[88] OpenSSL Project. Openssl-speed. https://docs.openssl.org/3.3/man1/openssl-speed/, [online].

[89] Arm Limited. Access Permissions. https://developer.arm.com/documentation/ddi0406/b/System-Level-Architecture/Protected-Memory-System-Architecture--PMSA-/Memory-access-control/Access-permissions?lang=en, [online].

[90] Freedom Metal. Physical Memory Protection. https://sifive.github.io/freedom-metal-docs/devguide/pmps.html, [online].

[91] Michael Larabel. PowerPC Memory Protection Keys In For Linux 4.16, Power Has Meltdown Mitigation In 4.15. https://www.phoronix.com/news/PowerPC-Mem-Protection-Keys, [online].

[92] The Apache Software Foundation. Apache HTTP server benchmarking tool. https://httpd.apache.org/docs/2.4/programs/ab.html, [online].

[93] Oracle. MySQL Benchmark Tool. https://dev.mysql.com/downloads/benchmarks.html, [online].

[94] MongoDB. Performance Tools for Mongodb. https://github.com/mongodb/mongo-perf, [online].

[95] Redis Ltd. How fast is Redis? https://redis.io/topics/benchmarks, [online].

[96] Wikipedia. A* Search Algorithm. https://en.wikipedia.org/wiki/A*_search_algorithm, [online].

[97] corbanbrook. DSP.js. https://github.com/corbanbrook/dsp.js/, [online].

[98] Jacob Seidelin. Pixastic Library. http://www.pixastic.com/, [online].

[99] Mozilla. Sheriffing/TBPL. https://wiki.mozilla.org/Sheriffing/TBPL, [online].

[100] Stanford Security Lab. Stanford Javascript Crypto Library (SJCL). https://wiki.mozilla.org/Sheriffing/TBPL, [online].

[101] Donghyun Kwon, Jangseop Shin, Giyeol Kim, Byoungyoung Lee, Yeongpil Cho, and Yunheung Paek. uXOM: Efficient eXecute-Only Memory on ARM Cortex-M. In *Proceedings of the 28th Conference on USENIX Security Symposium (USENIX Security'19)*, 2019.

[102] Zhuojia Shen, Komail Dharsee, and John Criswell. Fast Execute-Only Memory for Embedded Systems. In *Proceedings of the 2020 IEEE Secure Development (SecDev'20)*, 2020.

[103] Yaohui Chen, Dongli Zhang, Ruowen Wang, Rui Qiao, Ahmed M Azab, Long Lu, Hayawardh Vijayakumar, and Wenbo Shen. NORAX: Enabling Execute-Only Memory for COTS Binaries on AArch64. In *Proceedings of the 38th IEEE Symposium on Security and Privacy (S&P'17)*, 2017.

Appendix

A Kernel Structure Modifications to Support MemoryTrap

Figure A1 is the structures we defined in Linux kernel used to store the basic information about MemoryTrap. Some fields in kernel correspond to the fields in the newly defined ELF format, and some fields are used to store the run-time information for MemoryTrap protection. The *mtrap* structure stores the start and end virtual address of a booby trap in memory. The *mtrap_info_t* structure stores the basic information of a process protected by MemoryTrap: the enable flag, the trap flag, the PKey assigned to enforce execute-only memory, the number of booby traps, and the booby trap list. The mtrap_info_t structure is stored in the *task_struct* structure to save the MemoryTrap information in current process context. Figure A2 is the task_struct structure. We add a mtrap_info_t member at the end of the randomized struct fields, so the mtrap_info_t is compatible with the randomization protection.

```
typedef struct mtrap{
  uint64_t Start; //Start address of booby trap
  uint64_t End; //End address of booby trap
} MTrap;

struct mtrap_info_t{
  int enabled; //MemoryTrap enable flag
  int trap_allowed; //Read operation legality flag
  int pkey; //PKey (in MPK mechanism) of code pages

  size_t mtrap_num; //Number of booby traps
  struct mtrap *mtrap_list; //Booby trap list
};
```

Figure A1: The MemoryTrap structures in Linux Kernel.

```
struct task_struct {
  volatile long          state
  void                   *stack;
  refcount_t             usage;
  unsigned int           flags;
  unsigned int           ptrace;
  ...
  struct mtrap_info_t    mtrap_info ;
  ...
};
```

Figure A2: The task_struct structure in Linux Kernel.

B Performance on Real-World Applications

B.1 Web Servers

We compile and run the standard and protected web server versions, respectively. We use the ApacheBench [92] to simulate 500 clients to send HTTP requests 100,000 times asynchronously, and we record their running time to complete these 100,000 requests. To demonstrate the performance impact of MemoryTrap when requesting different page sizes, we tested five different page sizes: 50KB, 100KB, 200KB, 500KB, and 1MB. Table A1 shows the overhead of MemoryTrap protection on each web server.

Table A1: Overhead of MemoryTrap protected web servers using ApacheBench. The first row presents the request page size, and the data show the overhead for each size.

Web Server	50KB	100KB	200KB	500KB	1024KB	Avg.
Nginx	0.13%	0.54%	0.16%	1.01%	1.71%	0.71%
Apache	1.80%	0.05%	0.72%	0.23%	0.09%	0.58%
Lighttpd	2.11%	0.06%	1.14%	1.21%	0.16%	0.94%
Average	1.35%	0.22%	0.67%	0.82%	0.65%	0.74%

As shown in Table A1, the maximum overhead value is only 2.11%, and the average overhead is 0.74%, and most overhead values are very close to zero. This indicates that the performance impact of MemoryTrap protection on I/O bound web servers is negligible.

Table A2: Performance evaluation result for databases. For each database, the first row is the insertion performance, and the second row is the selection performance. The second column shows the results of the original (unprotected) versions, and the third column shows the results of MemoryTrap-protected version. The unit for the result of MongoDB is second, while the unit of other results is request times per second.

Database	Original	MemoryTrap	Overhead
MySQL	2008.13/s	1969.70/s	1.95%
	20053.05/s	19412.63/s	3.30%
MongoDB	11913.92/s	11905.58/s	0.07%
	14379.11/s	14370.48/s	0.06%
Redis	136138.42/s	134777.04/s	1.01%
	137691.20/s	137484.66/s	0.15%
SQLite	655.14/s	650.75/s	0.67%
	85971.78/s	83298.06/s	3.21%

B.2 Databases & JIT Engine

Databases Performance Results Unlike ApacheBench, there is no such unified performance benchmark for databases. We have to run each database with a customized testing suite.

1. MySQL: We use MySQL's official benchmark *sysbench* [93] to evaluate the insertion and query overhead. We configure sysbench with the *complex* workload to perform insertion and selection for 100,000 rows.

2. MongoDB: We use the official performance benchmark of MongoDB, *mongo-perf* [94], to evaluate the impact

Table A3: Result of Kraken benchmark. "SJCL" is short for Stanford Javascript Crypto Library.

	A*	Audio	Imaging	JSON	SJCL
Original	149.7ms	285.0ms	360.3ms	145.0ms	508.9ms
MemoryTrap	151.6ms	290.7ms	365.1ms	146.1ms	520.8ms
Overhead	1.27%	2.00%	1.33%	0.76%	2.34%

of MemoryTrap on MongoDB. We execute the *simple insert* and *simple query* workload and record how long the workloads can be completed to evaluate the insertion and selection overhead.

3. Redis: For Redis, we also use the official benchmark, *redis-benchmark* [95], to measure the overhead. We execute the *SET* and *GET* operation 100,000 times and record how many requests can be processed per second.

4. SQLite: Since there is no official benchmark for SQLite, we design a custom benchmark for SQLite to simulate other database benchmarks' workloads. We insert 100,000 rows of random data and select the inserted data by their primary key. We record the execution time and calculate how many requests that SQLite can process per second.

Table A2 shows the results of database evaluation. For each database, the first row shows the insertion performance, and the second row shows the selection performance. The last column shows the overhead incurred by MemoryTrap. As shown in Table A2, most of the databases have a very low overhead when enabling MemoryTrap's protection, and the average overhead is only 1.30%. This shows that MemoryTrap has a negligible overhead on protecting databases.

JIT Engine Performance Result We compile the custom V8 engine together with the latest Chromium to get a browser that can generate MemoryTrap protected JIT compiled code and run Kraken Benchmark. The benchmarks in the first row of Table A3 are: 1) A* search algorithm [96]; 2) audio processing using Corban Brook's DSP.js library [97]; 3) image filtering routines [98]; 4) JSON parsing [99]; 5) Stanford Javascript Crypto Library (SJCL) [100]. Table A3 shows the overhead of JIT engine protection using Kraken Benchmark. In all cases, the small overhead caused by MemoryTrap ranges from 0.76% to 2.34%, and the average overhead is only 1.54%.

Table A4: Time for process-related kernel operations in μs. Smaller is better.

Kernel	Null Call	Null I/O	Stat	Open Close	Select TCP	Signal Install	Signal Handle	sh Proc
Standard	0.09	0.14	0.46	1.11	1.60	0.14	1.03	760
MemoryTrap	0.09	0.14	0.47	1.13	1.60	0.14	1.02	764
Overhead	0.00%	0.00%	2.17%	1.80%	0.00%	0.00%	-0.98%	0.53%

C Other Microbenchmark Results

Table A4, Table A5, and Table A6 show the lmbench results for MemoryTrap low-correlation kernel operations.

Table A5: Local communication latencies in μs. Smaller is better.

Kernel	Pipe	AF UNIX	UDP	TCP	TCP Conn
Standard	5.08	4.93	8.09	9.68	11.08
MemoryTrap	5.12	4.97	8.12	9.57	10.93
Overhead	0.79%	0.81%	0.37%	-1.15%	-1.37%

Table A6: File & system latencies in μs. Smaller is better.

Kernel	0K File		10K File		Mmap	100fd
	Create	Delete	Create	Delete	Latency	Select
Standard	7.15	3.85	12.9	6.07	52.2K	0.78
MemoryTrap	7.18	3.90	12.5	6.11	52.6K	0.78
Overhead	0.42%	1.30%	-3.20%	0.66%	0.77%	0.00%

D Applicability to Other Platforms

Windows Platform Support The user-space components of MemoryTrap are OS-independent. However, the Windows kernel currently does not yet provide APIs to support the MPK mechanism. To transplant MemoryTrap to Windows platforms, we can resort to the mature Intel hardware virtualization mechanism EPT instead of MPK to remove the read permission of code pages. Then, we reimplement the exception handler in hypervisor to react to memory disclosure attempts.

Transplant MemoryTrap to ARM A series of papers have transplanted the idea of execute-only memory to various ARM platforms using different hardware features [101–103]. The diversity of ARM specifications offers multiple alternatives to implement MemoryTrap on ARM. All we need to do is to reimplement the fine-grained memory permission control with different hardware features. For example, we can use the unprivileged memory instructions and Memory Protection Unit hardware feature to implement MemoryTrap on Cortex-M [101]. On ARMv7-M and ARMv8-M, we can leverage the hardware-supported debugging facilities [102] to implement MemoryTrap. On AArch64, we can combine the Unprivileged Execute Never bit, Privileged Execute-Never bit, and two Access Permission bits [103], to implement MemoryTrap. Even without any hardware support, we can still implement the software-emulation version of MemoryTrap. For example, we can use the most significant bit to mark booby traps as non-readable memory. Then, we check if the targets of read instructions land in booby trap areas via instrumentation [35].

Table A7: The gadget distance distribution of real-world exploits. The data from Column 2 ∼ Column 6 indicate the count of two adjacent gadgets' distances within that range. "Avg." and "Max." represent "Average Distance" and "Max Distance " of two adjacent gadgets. "Fir.-Las." represents the distance between the First gadget and the Last gadget.

	<1KB	1∼2KB	2∼4KB	4∼8KB	>8KB	Avg. (KB)	Max. (KB)	Fir.-Las. (KB)
CVE-2013-2028	0	0	0	1	3	123.7	193.8	494.7
CVE-2014-2624	0	0	0	0	4	69.8	210.9	279.2
CVE-2014-8322	0	1	0	0	2	31.3	73.3	94.0
CVE-2014-4880	0	0	0	0	3	1494.1	2851.8	4482.2
CVE-2016-2384	0	0	0	0	5	322.4	673.3	1611.9
CVE-2017-6553	0	1	0	2	1	135.0	525.7	539.8
CVE-2017-1000112	1	0	0	2	8	356.1	2937.4	3916.8
CVE-2017-5815	1	0	0	0	15	441.2	2304.8	7060.0
CVE-2017-11176	2	1	1	1	13	1183.2	16127.8	21297.3
EDB-ID-45288	0	0	0	0	7	83.5	234.0	668.4
EDB-ID-44331	0	0	1	0	6	115.1	458.8	920.7
EDB-ID-47482	2	4	2	4	8	11.9	96.4	249.2
EDB-ID-46808	0	0	0	0	5	2291.0	8523.9	13745.9
EDB-ID-47122	0	0	0	0	8	272.3	1177.5	2450.5
CVE-2020-12352	0	0	0	0	5	1133.7	4767.0	5668.6
CVE-2021-22555	1	0	0	0	10	650.6	2105.1	7157.0
Total	7	7	4	10	103	N/A	N/A	N/A

The software-emulation approach is agnostic to the underlying ARM architecture but at the cost of poor performance.

Table A8: Code page traversal results for other vulnerable programs with MemoryTrap enabled. "# of AFOG" is short for "Number of Attempts to Find One Gadget."

CVE Number	Average Bytes Disclosed		# of AFOG	
	Breadth-first	Depth-first	Breadth-first	Depth-first
CVE-2014-8322	687	340	26	9
CVE-2014-1303	670	273	29	15
EDB-ID-45288	536	311	20	10
EDB-ID-44331	501	289	19	13
EDB-ID-47482	499	322	23	15
Average	579	307	23	12

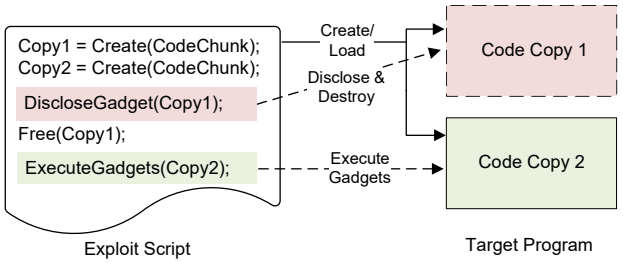


Figure A3: An example of code inference attacks. Code copy 1 is only used for code disclosure, while code copy 2 is only used for gadgets execution.

E Statistical Analysis of Real-world Exploits

We collect publicly available real-world exploits that can reproduce JIT-ROP attacks from Metasploit [82] and Exploit-DB [83]. For every two adjacent gadgets, we count the number of distances within different ranges, from less than 1KB, 1KB∼2KB, 2KB∼4KB, 4KB∼8KB, and greater than 8KB. Table A7 shows the gadget distance distribution data for these exploits. For the exploits without CVE number, we show their Exploit-DB ID (EDB-ID) instead. In Table A7, there are 131 different distance types between two neighboring gadgets, but only 18 of them are less than one memory page's size (i.e., 4KB), accounting for 13.7% of the total. This means that the probability of having two available gadgets in one memory page is less than 13.7%. In Table A7, the average gadget number for an exploit is 8.2, and thus the probability that all gadgets are located in one memory page is at most $0.137^{8.2} \approx 0.000000083\%$, an almost negligible number. In all gadget chains we collected, the minimum distance from the first gadget to the last gadget is 94.0KB, which means the attackers have to disclose at least 94.0KB of code to build the whole payload.

For the exploits we collected from public databases and the vulnerable programs that can also be protected by MemoryTrap, we use the strategies in §6.1 to measure code page traversal results as well. Table A8 reveals a similar pattern to that of CVE-2013-2028: MemoryTrap leaves adversaries with little wiggle room to harvest all of the required gadgets, and only a few attempts get a chance to find **at most one gadget**. The data in Table 2 and Table A8 demonstrate that our booby trap insertion strategy is effective in preventing JIT-ROP gadget search.

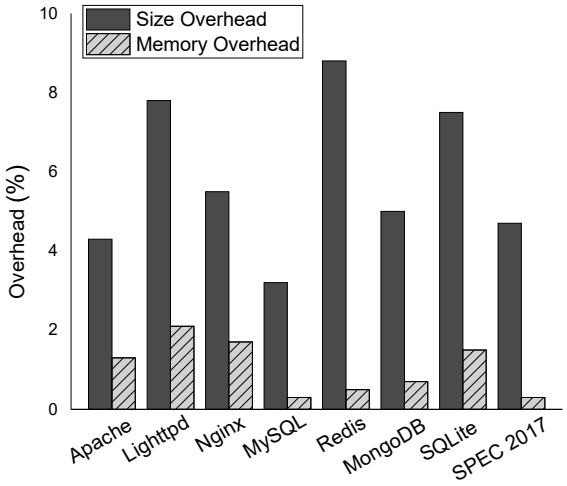


Figure A4: Disk size and memory overhead of MemoryTrap.

F Code Inference Attacks

Snow et al. [57] proposed four types of *code inference attacks*, rendering DCR ineffective. The core principle of code inference attacks is to disclose a segment of code without executing it directly. Instead, another piece of code that is strongly related to the disclosed code will be executed, such as an exact copy of the disclosed code residing in a different memory region, or relevant code that can be inferred based on the disclosed segment. The core idea of code inference attacks is to disclose a piece of code but not to execute it. Instead, another piece of code that is strongly related to it will be executed, such as an exact same copy of the disclosed code in a different memory area, or the relevant code that can be predicted based on the disclosed ones. As shown in Figure A3, adversaries maintain two copies of the same code, and they only disclose one copy to collect gadgets and execute the other copy. The four code inference attacks are as follow:

T1. JavaScript JIT Cloning: Load two copies of the same JIT code to disclose one copy and use the other.

T2. Shared Library Reloading: Disclose a library's code and let it be destructed, and then reload it to use the previously disclosed code.

T3. Process Reloading: Disclose the code of an active process and then reload it to use the refreshed code.

T4. Implicit Reads: Disclose some code to infer the content of another code copy to avoid being destroyed.

G Other Overhead

Since we insert booby traps into the program and attach the ".mtrap" section into the ELF file, the binary size will increase naturally. When executing the protected binary, the size of these contents are fixed after being loaded, and does not change during program execution. We measure the disk size and memory overhead for real-world applications and SPEC CPU 2017 benchmarks that are protected by MemoryTrap. Figure A4 shows the disk size and memory overhead. The disk size overhead rages from 3.2% to 8.8%, and the average disk size overhead is only 5.85%. Memory overhead is smaller than disk size overhead due to the allocation of stack and heap during the program execution. The memory overhead varies between 0.3% and 2.1%, with an average memory overhead that remains negligible at just 1.05%.

When employing MemoryTrap with code randomization, in addition to code pointers, all booby trap pointers also need to be fixed to their shuffled positions, allowing the kernel to recognize the relocated booby trap positions. This introduces overhead during binary loading. To measure this overhead, we collect the booby trap count during compilation with our toolchain and determine the number of pointers requiring fixups using CCR's toolchain [79]. The overhead is calculated by dividing the number of booby traps by the total number of pointers needing fixups. The average overhead for web servers, databases, and SPEC 2017 benchmarks is 5.67%, 2.78%, and 4.01%, respectively, demonstrating a minimal fixup burden. It is noteworthy that this overhead is a one-time cost and does not impact runtime performance.

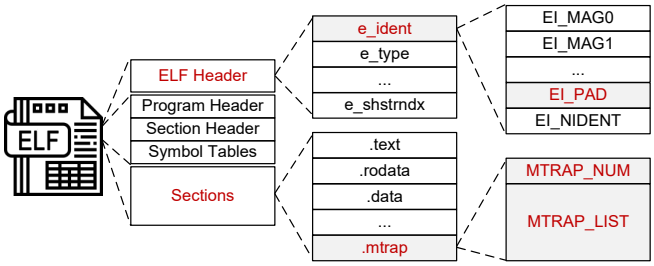


Figure A5: New ELF format to record booby traps metadata.

H New ELF Format

Figure A5 illustrates the new ELF file format we defined.

Separate but Together: Integrating Remote Attestation into TLS

Carsten Weinhold
Barkhausen Institut

Muhammad Usama Sardar
TU Dresden

Ionuț Mihalcea
Arm

Yogesh Deshpande
Arm

Hannes Tschofenig
University of Applied Sciences Bonn-Rhein-Sieg

Yaron Sheffer
Intuit

Thomas Fossati
Linaro

Michael Roitzsch
Barkhausen Institut

Abstract

Confidential computing based on Trusted Execution Environments (TEEs) allows software to run on remote servers without trusting the administrator. Remote attestation offers verifiable proof of the software stack and hardware elements comprising the TEE. However, setting up a secure channel to such a TEE requires a security guarantee that the channel actually terminates inside the TEE. TLS is an existing protocol for secure channel establishment, and in its most common use on the Web, it uses a key pair to assert the server identity encoded in a certificate. Various approaches have been proposed to integrate remote attestation into TLS. Unfortunately, they all have shortcomings. In this paper, we present a protocol that combines the existing certificate-based assurances of TLS with remote attestation-based assurances in a way that they can be *deployed independently* and can *fail independently*. We design these two assurances to be additive without relying on each other, a property that has not been considered by existing approaches.

1 Introduction

Remote attestation, originally developed for Trusted Platform Modules (TPMs) [1], recently experiences a revival due to the availability of Trusted Execution Environments (TEEs) [2] in processors and initiatives towards confidential computing [3]. With confidential computing, a cloud server runs client workloads inside a TEE, with the security guarantee that the code being executed and the data being processed is not accessible even to a malicious system administrator with control over the hypervisor. During execution, TEE state is shielded from software outside the TEE and physically never leaves the processor unencrypted. Remote attestation offers a cryptographic proof to a remote party that the TEE is backed by genuine hardware and asserts which software is running inside it.

However, to communicate with a TEE, the remote party also requires a secure channel into the TEE. Establishing such a channel should be compatible with multiple TEE implementations, and the protocol must guarantee channel termination inside the TEE. Transport Layer Security (TLS) [4] is a widely-deployed and well-analyzed protocol providing secure channels, but needs to be extended to support remote attestation. Such extensions to TLS were explored when TPMs were introduced [5, 6] and more recently with Intel SGX [7, 8] and other TEEs [3]. Such a TLS extension is non-trivial as it must take care of relay attacks [9, 10], where the attestation report of a valid TEE is used by a malicious intermediary to terminate the encrypted connection outside the valid TEE. Therefore, TLS and attestation cannot be used side-by-side, but must be cryptographically linked with each other to guarantee that the encrypted channel terminates at the origin of the attestation report [9].

The existing approaches differ in the way both aspects are linked: One option is to establish a regular TLS connection first and then arrange an attestation exchange through the TLS channel to establish a second, attestation-based communication channel inside the outer TLS channel. However, this construction [8] results in additional handshake round trips and an additional layer of encryption for the inner channel. Another option is to include attestation information in the certificate used to establish the TLS connection, thus linking both aspects statically at deployment [9]. Such linking risks losing the attestation-based assurance in case the long-lived TLS private key is ever leaked, for example through a Heartbleed-like side channel [11].

In this paper, we present TLS+RA, an attestation extension to TLS. To maintain best practices and the resulting trust relationships, TLS+RA keeps the deployment of certification authorities (CAs), which issue end-entity certificates to TLS servers, separate from attestation-related information and vice versa. Therefore, we opt for a late linking of TLS and attestation during the connection handshake, rather than modifying certificates. Using short-lived, per-session handshake secrets rather than long-lived TLS keys, the failure behavior remains independent, resulting in truly additive security assertions. We have not seen this property considered in existing approaches. Two key enablers to revisit this problem now are the compulsory use of ephemeral Diffie-Hellman key agreement (when

used with certificate-based authentication) and the extension mechanisms available in TLS protocol version 1.3 [4].

In summary, TLS+RA combines attestation and TLS in a new way, so that channel termination inside the TEE is guaranteed while both connection aspects are kept separate in terms of deployment and failure behavior. Our contributions are:

1. The introduction of *deployment and failure independence* (Section 2) and a survey of related work (Section 3).
2. The TLS+RA protocol (Section 5), a TEE-agnostic attestation extension to TLS offering deployment and failure independence. While server-side TEEs are our main motivation for use in confidential computing, the protocol supports client-side and mutual attestation as well.
3. An implementation based on OpenSSL (Section 6), which we evaluate regarding handshake latency and channel throughput (Section 7). The code is available as open source [12].

2 Background and Properties

After reiterating the fundamental security properties of TLS and attestation, we explain our desired properties of deployment and failure independence for the combination of both.

2.1 Properties of TLS and Remote Attestation

The TLS protocol establishes a confidential and integrity-protected channel between a client and a server. For a typical use of TLS 1.3 [4] on the Web, the channel is based on an ephemeral shared secret established by Diffie-Hellman key agreement. This key agreement ensures that a passive eavesdropper listening to the exchange cannot derive the shared secret. To prevent active attacks, the server uses an X.509 certificate and key confirmation to authenticate to the client (typically a browser). These server certificates associate an endpoint identifier, typically a Fully Qualified Domain Name (FQDN) [13], with a public key. The certificate is signed by a CA in response to a certificate request from the operator of the server, after the requester has demonstrated that it controls the FQDN. Only the valid endpoint can prove possession of the matching private key, unless this private key is leaked or stolen. To authenticate a server, the TLS client compares the expected FQDN (e.g. the URL entered into the browser) with the FQDN found in the subject name of the certificate to decide whether to trust the server or not [14].

Rather than demonstrating control over a domain name, remote attestation offers an assertion of the hardware and software stack running on the remote machine [1]. A pre-requisite for remote attestation is a set of components on the remote machine that are trusted to reliably ascertain the running software stack and sign a cryptographic report of its state. Collectively, this set of components forms the root-of-trust

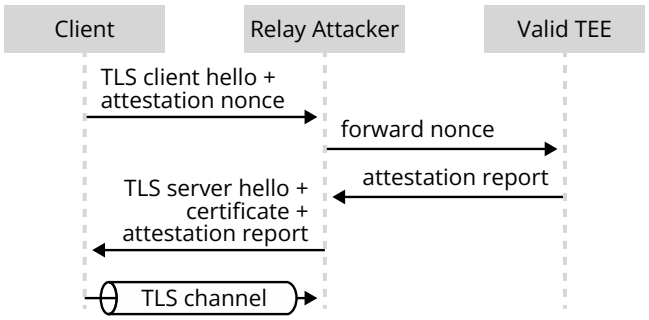


Figure 1: Relay attack when combining TLS and attestation.

(RoT). Each TEE comes with its own RoT implementation, typically involving a secret securely kept inside the device hardware. SGX, for example, implements the RoT as part of the main CPU, including software components performing the report signature. Processor-independent RoTs like TPMs can externally inspect and attest the software being booted on a CPU, effectively treating this boot state as one large execution environment. When connecting to a TEE, the TEE is sent a challenge in the form of a nonce. The TEE asks its RoT to respond with a signed quote, containing the nonce and measurements of the current hardware/software state. The key pair with which this attestation report gets signed is confirmed to be valid by the RoT's manufacturer, usually also in the form of a certificate. As long as the private key pertaining to the RoT is not leaked, only valid hardware with this RoT can create the signature. The connecting party can make a trust decision based on the TEE state encoded in the attestation report.

We believe both trust decisions are useful and orthogonal to each other: While attestation makes an assertion about the internal state of the TEE, the TLS certificate asserts the endpoint identity and administrative control, i.e., factors external to the TEE. As TEEs do not exist in a vacuum, the infrastructure around it may influence the trust decision. Therefore, we intend to combine both assertions into one protocol but keep them independent enough that their properties are additive.

2.2 Combining TLS and Remote Attestation

Past research [9] has shown that combinations of both these protocols can be insecure. Figure 1 illustrates a relay attack. When TLS authentication and attestation are performed next to each other, i.e., without cryptographically interlinking the handshake, the resulting secure channel is not guaranteed to terminate in the TEE issuing the attestation report. In case an attacker obtains the private key of the TLS certificate, a fake endpoint can be set up running an invalid software stack. The attacker forwards the attestation nonce to a correct TEE and thus obtains a valid-looking attestation report, which is forwarded to the client. This attack shows that protocol combinations without adequate cryptographic linking are not independent in their properties: here, the credibility of the

attestation relies on the protection of the TLS private key. If, for example, the TLS private key is handled by cloud provider infrastructure, we suddenly trust the cloud provider as to the validity of attestation reports, undermining the promises of TEEs and confidential computing. Therefore, we postulate five properties a combined TLS+RA protocol should provide:

1. Retain the properties of the secure channel, like confidentiality, integrity, and replay protection, but additionally transmit an attestation report. Currently, there is no Internet protocol standard for establishing a secure channel with remote attestation. Hence, building upon the battle-hardened [15] TLS protocol is natural. The protocol must guarantee that the tunnel terminates in the TEE to prevent relay attacks.

2. Do not add extra network message round trips to the connection handshake. Attestation should not unduly increase the cost of establishing a TLS channel.

3. Once the channel is established, do not add encryption layers. The regular TLS payload encryption should suffice.

4. Independent deployment: To remain compatible with the existing certificate practices (issuance, expiration, revocation, use in transparency logs), the use of certificates and the underlying public-key infrastructure (PKI) should not be changed. Especially, TLS certificates should not contain attestation information, which would complicate load balancing in data centers, where the same domain name may be served by different machines with different attestation technologies. Equally, attestation deployment should only involve the TEE vendor, not a Web PKI CA issuing certificates typical for use on the Web.

5. Independent failure: If either the TLS certificate private key or the attestation RoT are compromised, the assertion of the other should remain intact. The TLS private key can be leaked through side channels like Heartbleed [11]. Then, the connecting party can no longer be sure to connect to the expected machine in the expected administrative environment. But the attestation assertion to the validity of the software stack should remain correct. Vice versa, if the attestation RoT is compromised, the validity of the stack can no longer be assured. However, the TLS certificate should continue to demonstrate that the domain is still under the control of the owner.

With these properties in place, TLS+RA offers additive security properties of TLS and remote attestation.

3 Related Work

While prior work exists that designs such protocols from scratch [10], we focus on combinations of attestation and TLS, because we want to retain and augment the properties of TLS rather than replace it. Points in the solution space differ by the method of combining TLS and attestation, which we cover in the following (Table 1) from coarse-grained linking towards more intricate merging of both. We note that all solutions offer protection against relay attacks if neither the TLS private key nor the attestation RoT private key are leaked. However, only few offer failure independence in case the TLS private key is compromised. The protocol should still ensure that the TLS channel terminates in the TEE that created the attestation report.

Table 1: Related work compared to the TLS+RA properties.

	Prevents Relay Attacks	No added round trips	No added encryption	Independent deployment	Independent failure
HTTPA [8, 16]	✓			✓	✓
Platform Certificate [9]	✓	✓	✓		
RA-TLS [7]	✓	✓	✓		
RA-TLS with CA [7]	✓		✓		
Extending TLS [6]	✓		✓		✓
RATLS [17]	✓	✓	✓	✓	
Trusted Channels [5]	✓	✓	✓	✓	

HTTPA [8, 16] offers such failure independence. It does so by nesting an attestation-based encrypted channel inside the TLS-based encrypted channel, doubling handshake and encryption effort. To avoid this cost, the linking has to happen in the TLS handshake, resulting in a single layer of encryption. Goldman et al. [9] postulate binding the TLS key pair to the attestation RoT key pair by way of a platform certificate, which is signed by a regular CA. If the valid TEE obtains such a certificate and afterwards loses its TLS private key, a relay attack can be launched by presenting this same platform certificate plus the relayed attestation report. Since certificates are connection-independent, they cannot prevent such connection relaying.

Per-connection certificates may offer a way out. RA-TLS [7] creates an ephemeral, self-signed certificate, which contains the attestation report itself. Due to the use of self-signed certificates it does not provide the certificates used in Web PKI environments. The same paper proposes an extension where the ephemeral certificate is signed on-the-fly by a PKI CA. However, this increases connection latency and does not offer failure independence: to automate this per-connection process, a long-lived secret or proof-of-domain-control [13] must be presented to the CA, both of which an attacker could replicate to mount a relay attack without compromising the TEE RoT. Instead of using a TLS certificate as a transport vehicle for the attestation report, Aziz et al. [6] obtain for every connection an ephemeral attestation-key certificate for the RoT and bind this RoT to the TLS handshake nonce. This is enough to prevent

relay attacks even when the TLS key is compromised, but the described approach adds handshake messages and a full round trip for the certificate signing request to the critical path.

Walther et al. [17] and Armknecht et al. [5] describe two protocols very closely integrated with TLS. Both use the existing TLS extension mechanism to add additional data fields to regular TLS handshake messages. TLS certificates remain unmodified. However, both protocols use the TLS private key to link the attestation report to the TLS connection being established. Should the TLS private key be leaked, this linking can be performed outside the TEE on another machine, thus enabling a relay attack.

Finally, co-authors of this paper contribute to an Internet protocol draft [18] that aims to standardize how TLS message extensions carry attestation evidence. The draft seeks interoperability with a related standard proposal [19] and therefore links attestation to the TLS private key. Hence, it offers security guarantees similar to the works of Walther et al. [17] and Armknecht et al. [5] that we already discussed above.

In summary, promising approaches exist, but only designs costly in terms of handshake messages or encryption overhead fulfill our desired property of failure independence.

4 Deployment and Threat Model

We assume that the TLS public-key infrastructure and its CAs are operated independently of the TEE manufacturer and associated remote-attestation infrastructure. However, we do require that the TLS+RA library, the application using it, and any security-critical code they depend on run inside a TEE. This requirement ensures that the code implementing the combined TLS+RA handshake is captured in the attestation report; some TLS+RA state must also be included, as described in the following section. The exact procedure of how to start software in a TEE, what else to include in an attestation report, and how to validate it are orthogonal to the TLS+RA approach. However, we assume that all necessary steps for TEE provisioning have been executed securely.

Under these assumptions, we consider an attacker who can compromise either the TLS private key or the attestation RoT, but not both. A TLS private key or an RoT private key can be leaked (e.g., through a side-channel attack or information-disclosure bug) or a fake key can be certified (in case the responsible CA has been compromised). If an attacker compromises only the RoT key material but *not* the isolation of the associated TEE, then an invalid attestation report can be forged but the TLS private key inside the TEE is not assumed to be compromised.

5 Protocol Design

Although our TLS+RA protocol supports mutual attestation of client and server, let us consider a confidential computing scenario where a client interacts with a workload running inside a TEE hosted by a cloud provider. The client wants to establish a TLS channel with the guarantee that the channel terminates inside the TEE running the expected software. Thus, the server should attest its TEE state to the client. The flow starts with the client checking its attestation verifier for supported attestation evidence formats. This verifier can be a local library or a remote service, but this is orthogonal to the design of TLS+RA. The client then initiates the TLS+RA handshake with the server by sending a client hello message. As part of this message, the client forwards supported attestation evidence formats and an attestation nonce for freshness from the verifier to the server as part of an *attestation request*. If the server supports one of the evidence formats presented to it, it returns a corresponding attestation report as part of its TLS certificate message. If none of the formats match, the handshake terminates.

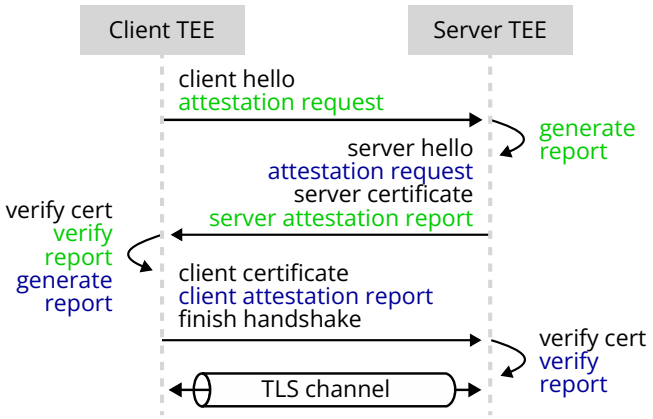


Figure 2: TLS+RA handshake with mutual attestation (server attests to client in light green, client attests to server in dark blue, black is standard TLS).

Building upon TLS In order to ensure deployment independence, we do not want to change how TLS certificates are managed or what information they carry. Thus, attestation data must be added next to the existing TLS data as part of the TLS handshake. As demonstrated by prior work [5,17], it is possible to exchange attestation-related information between the client and the server via *message extensions* available in TLS version 1.3. Figure 2 shows the additional attestation request and report encapsulated in extensions to the hello messages (extended with respective attestation request) and certificate messages (extended with respective attestation report). These extensions are transmitted together with the regular TLS handshake messages and do not incur additional network round trips.

To prevent relay attacks, the TLS+RA session and the attestation report must be cryptographically linked. To construct this link, we exploit another feature of TLS 1.3: the compulsory use (outside of special cases using pre-shared keys-only mode) of Ephemeral Diffie-Hellman (DHE) key agreement. DHE key agreement gives both parties a freshly-generated shared

secret that is unknown to any third party. However, there are no assurances about who the two parties are. Regular TLS solves this problem by linking the shared secret to the TLS certificate. For this, a cryptographic hash called *transcript hash* is computed over a log of the exchanged messages up until this point in the handshake, meaning the client hello message received by the server as well as the server hello response generated by the server. This hash therefore subsumes TLS nonces and the public parts of the DHE key material that each peer must use to compute the shared secret. The transcript hash is then signed using the private key belonging to the TLS certificate. This signature offers cryptographic proof of possession of the TLS private key and thus establishes a link between the TLS session and the endpoint identity (typically the domain name).

Linking TLS and Attestation Handshakes To this existing link in standard TLS, TLS+RA adds a second, independent link: We generate a *linking hash* and forward it as the challenge to the attestation RoT. As the key difference compared to the transcript hash, this linking hash is computed over the same TLS handshake messages, but also includes the DHE-derived shared secret. Without this shared secret, the linking hash would only cover publicly visible information of handshake messages, therefore opening the possibility of a relay attack. Including the shared secret, which is only known to the connecting parties, prevents this attack. The verifying peer can compute the same hash, because it actively participated in the DHE key exchange. Note that the linking hash also ensures freshness by including the attestation nonce, which is part of the attestation request transmitted via a message extension. In summary, this hash links the attestation report to the TLS session in the same way the TLS certificate is linked to it. The secure tunnel is therefore guaranteed to terminate where the TLS private key is kept *and* where the attestation report was generated.

Additive Security in TLS+RA As shown in Figure 3, this *double linking* causes both links to be symmetric (they are derived from the same message log) and independent from each other (they use different signing keys and deployment infrastructure). If the attestation RoT fails, but TEE isolation remains unaffected as described in our threat model (Section 4), the TLS assurances remain intact. If the TLS private key is compromised, the client can still trust and verify the attestation evidence. The attacker can only set up a rogue server with the same TEE hardware and the same software the client expects; otherwise the client will reject the attestation evidence.

Furthermore, double linking prevents relay or other intercept attacks [20]. TLS+RA inherits the property of *forward secrecy* from TLS 1.3: A stolen TLS private key cannot be used to decrypt previously intercepted traffic, because message payloads are instead encrypted using a key derived from the randomly-generated, per-session DHE shared secret. However, unlike the standard protocol, TLS+RA also prevents the attacker from using the TLS private key to mount *future* relay attacks. If the attacker sets up a fake server running malicious software using

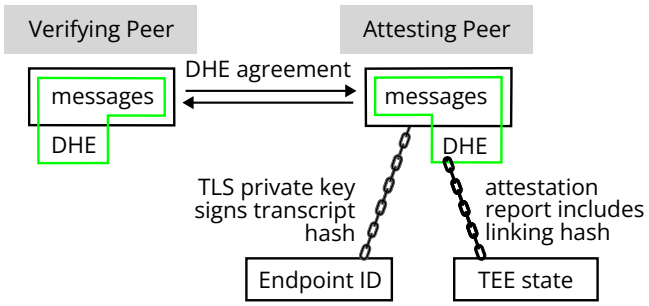


Figure 3: Double linking of TLS+RA handshake. While the TLS private key signs the message transcript, the linking hash in the attestation report also includes the shared DHE secret.

the stolen key, this fake server cannot attest itself and must therefore establish separate channels with the client and the original server. Since the attacker cannot predict the randomness used by the other two machines, both TLS+RA channels will use different DHE secrets resulting in different linking hashes. Hence, the client will detect the mismatch between the linking hash based on its own DHE computation and the one that has been attested by the original server; it will then abort the handshake.

Other protocols lack this property, because they link the attestation with the TLS certificate private key [7, 9], thus not forming two independent links. Or, they link it only with the TLS nonce [5, 17] but not with the DHE key material, which would additionally protect against relay attacks. The attestation linking we propose matches the protocol that Stumpf et al. designed from scratch without TLS integration [10]. TLS+RA thus integrates remote attestation with the state-of-the-art and widely deployed TLS 1.3 protocol, benefitting from its known security properties.

6 Implementation

Similar to the work done by Armknecht et. al. [5] and Walther et. al. [17], we built our TLS+RA prototype on top of OpenSSL. With `SSL_CTX_add_custom_ext`, this TLS library offers a public API for adding callbacks that can create or inspect message extensions during the TLS handshake. Another callback registered with `SSL_CTX_set_verify` is invoked, when the attestation report has been received as an extension to a certificate message. Using `SSL_CTX_set_ex_data`, we embed user-defined data in the TLS session context. Using these APIs, we implement a remote attestation state machine as a set of callback functions that runs in parallel to the TLS handshake.

However, there is one modification we have to make to the OpenSSL codebase: to link the attestation report to the TLS session, we must compute the linking hash over the log of handshake messages and the DHE-derived shared secret. This information is readily available internally in OpenSSL, but not accessible through an API. As part of the TLS specification,

RFC 5705 [21] defines so-called *exporters*, which do give applications access to TLS keying material. However, the corresponding APIs can only be called after the completion of the handshake. This is too late for our approach. We therefore add an additional, non-standard exporter function that generates the linking hash during the handshake, right after OpenSSL has computed the shared secret using DHE key agreement. Note that this extension does not change the TLS wire protocol, but merely derives the linking hash from internal state of the TLS implementation to facilitate the cryptographic link between the TLS session and the attestation report.

The way in which TLS+RA extends standard TLS is agnostic to the TEE in which it is running, the underlying attestation RoT, and the data formats used in attestation reports. TLS+RA treats these reports as opaque data when shipped inside TLS message extensions, so any data format can be carried. To demonstrate the separation between TLS+RA as a protocol and the underlying attestation RoT, we integrate a simple API for platform-specific TLS+RA plugins. At the core, plugins for this API must implement two functions: `remote_attest` and `check_report`, which request and validate attestation reports, respectively. Two of these plugins, one for industry-standard TPMs and one for AMD's Secure Encrypted Virtualization with Secure Nested Paging (SEV-SNP), will be evaluated quantitatively in the next section. Another plugin targeting Arm Confidential Compute Architecture (Arm CCA) has been developed and tested using an emulator. This plugin will not be used in benchmarks, because Arm CCA-enabled hardware is not available to the public yet.

7 Evaluation

With our experiments, we want to demonstrate the following properties of TLS+RA:

1. Qualitatively show interoperability with different TEE implementations.
2. Quantify the overhead of TLS+RA over standard TLS to demonstrate the feasibility of TLS+RA.
3. Compare TLS+RA against approaches that layer attestation on top of TLS to demonstrate the benefit of merging TLS and attestation into a combined protocol.

We evaluate TLS+RA on AMD SEV-SNP (representing a cloud server TEE), firmware TPMs (representing end user devices), and hardware TPMs (representing Internet-of-things devices). For AMD SEV-SNP, we use an Amazon c6a.large instance running Amazon Linux 2023 with Linux kernel 6.1. The firmware TPM (fTPM) implements the TPM 2.0 specification and is provided by a 13th Gen Intel Core i5-13400 desktop machine running Ubuntu Linux kernel 6.5. The discrete hardware TPM (dTPM) is an Infineon Optiga SLB 9670, also implementing TPM 2.0. It is connected to a Raspberry Pi 4 Model B Rev 1.4 with four Cortex-A72 cores and running kernel 5.15. For Arm CCA, we use an emulator by Arm which allows to test functionality, but is not suitable for performance measurements. To qualitatively test the interoperability of TLS+RA, we successfully executed combined TLS and remote attestation flows with these four technologies. TLS+RA supports server-to-client, client-to-server, and mutual attestation.

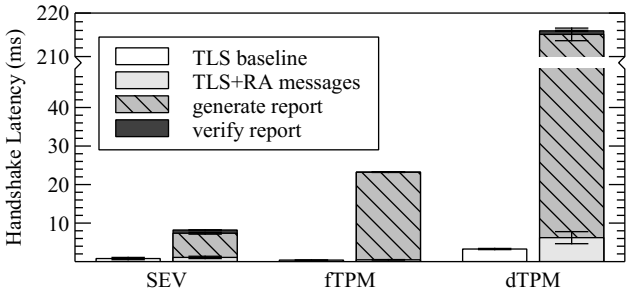


Figure 4: Handshake latency for TLS+RA compared to standard TLS. Note: The y-axis skips the range from 50 to 210 ms.

Attestation Overhead TLS+RA does not add any network round trips compared to standard TLS. Thus, to measure the worst-case processing overhead of the combined protocol, we co-locate both client and server on different cores of the same machine and let them communicate over the loopback interface. We disable idle sleep states on these cores to reduce measurement noise. Figure 4 compares a standard TLS baseline against TLS+RA for each attestation technology. Each measurement is repeated 100 times after two warmup runs; mean and standard deviation (some error bars are too small to be visible) are plotted. The overall handshake duration for TLS+RA is broken down into the time required to generate the attestation report on the server, the time required to verify the report on the client, and the processing and transmission of all messages by the OpenSSL library.

We see that TLS+RA adds overhead over standard TLS, which is dominated by the generation and verification of the attestation report. The remaining gap between TLS+RA message processing and baseline TLS is caused by increased computation and system-call overhead, as TLS+RA handshake messages are larger due to the additional report data. While baseline TLS sends approximately 1.5 KiB of data, TLS+RA for the fTPM technology has to transfer 6.4 KiB. The increase in data volume is dominated by the attestation report (called a *quote*) and the public part of the TPM's *attestation identity key*. The prototype plugin for TPMs serializes this information into JSON format. The SEV plugin forwards the original binary representation of both the roughly 1 KiB attestation report and the key certificate of the host platform. All reports are coarse-grained, describing the software stack as a whole.

Network Latency To quantify the end-to-end overhead of TLS+RA across various network distances, we plot in

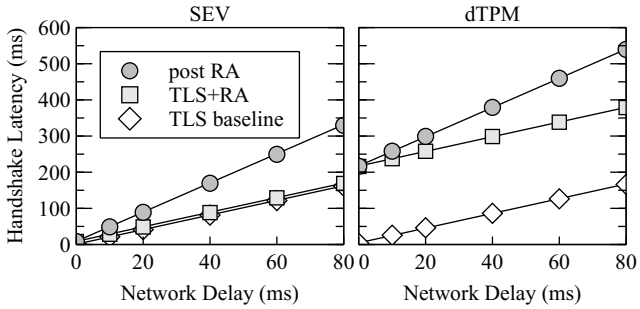


Figure 5: Handshake latency for TLS+RA compared to standard TLS and TLS with post-handshake attestation for various network delays. Lines between measurement points are linear fits to help with readability.

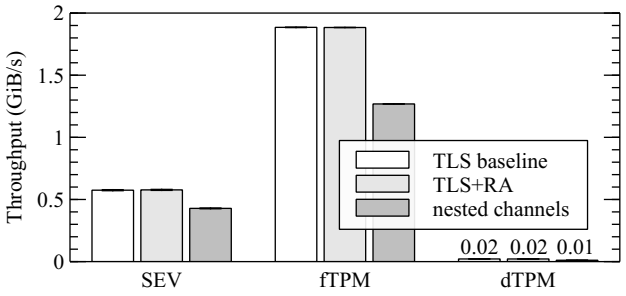


Figure 6: Channel throughput for TLS+RA compared to standard TLS and nested channels.

Figure 5 the handshake duration including network latency. We use Linux' *netem* facility to simulate network delays of up to 80 ms per direction, which is representative of client and server being on different continents. Again, the mean of one hundred runs is presented, after two warmup rounds. For SEV, which takes only 6 ms to generate the report, the relative overhead of TLS+RA shrinks to 5 percent at 80 ms network latency. With dTPM, report generation needs 210 ms, resulting in a 1.3x handshake overhead at 80 ms network distance.

Comparison to Related Work To compare TLS+RA not only against standard TLS as a baseline, we re-implemented the composition approaches of related works, while keeping the TLS protocol version, its implementation, and the hardware the same. The lines labeled "post RA" in Figure 5 show the performance impact of extra network round trips, when attestation is performed *after* the TLS handshake. For this experiment, we re-implemented the approach used by related work solutions such as HTTPA [8] and others, which require twice as many network round trips between client and server. In the case of SEV, for 10 ms network delay and greater, performing the TLS and attestation handshakes sequentially incurs more than 100% overhead compared to baseline TLS.

Solutions like HTTPA also nest two independent channels, requiring redundant encryption and integrity protection of all message payload. To quantify the associated costs, we compare throughput of standard TLS, TLS+RA, and a re-implementation of the nested channel solution. The nested channel is simulated using a second layer of AES-GCM authenticated encryption for sending 1 GiB of payload from server to client. Client and server again run on the same machine, so throughput is limited by cryptographic operations protecting the payload. The mean of 10 runs after two warmup runs is plotted. Figure 6 illustrates that TLS and TLS+RA perform similarly after the initial connection handshake, because they perform exactly the same cryptographic operations. Adding remote attestation introduces no overhead post-handshake. Throughput is therefore not determined by the TEE technology, but by the processor performing cryptographic operations. A nested channel solution achieves a lower overall throughput because of the additional cryptographic overhead.

Similar to TLS+RA, related works such as [5, 7, 17] avoid the cost of additional network round trips and nested channels, too. However, they offer weaker security properties by lacking failure independence (Table 1).

8 Conclusion

We have presented TLS+RA, a network protocol combination of TLS and remote attestation. TLS establishes an encrypted and integrity-protected channel to a specific endpoint, whereas remote attestation cryptographically verifies the hardware and software state of that endpoint. Both protocols are combined at the handshake level using standard TLS message extensions. The novel properties of deployment and failure independence set TLS+RA apart from related work. During the handshake, double linking of both the TLS certificate and the attestation report against the secret established during handshake is performed. Thereby, TLS+RA guarantees that the encrypted channel terminates at the attested machine even if the TLS certificate private key is compromised. By adding this security assertion to standard TLS, we hope to make attestation more accessible and thus more pervasively used in today's connected world.

Acknowledgements

We thank the anonymous reviewers of the USENIX ATC'24, ATC'25, and ACM/IFIP Middleware'24 program committees for their valuable feedback, which helped us to improve this paper. We thank our shepherd Joel Wolfrath for his feedback on drafts of the final version. This research is funded by the European Union's Horizon Europe research and innovation program under grant agreement No. 957216 (iNGENIOUS), No. 101094218 (CYMEDSEC), No. 101092598 (COREnext), and via German Research Foundation (DFG) grant No. 389792660 as part of TRR 248 (CPEC). It is also financed on the basis of the budget passed by the Saxon State Parliament in Germany.

References

[1] Boris Balacheff, Liqun Chen, Siani Pearson, David Plaquin, and Graeme Proudler. *Trusted Computing Platforms: TCPA Technology in Context*. Prentice Hall, 2003.

[2] Patrick Jauernig, Ahmad-Reza Sadeghi, and Emmanuel Stapf. Trusted Execution Environments: Properties, Applications, and Challenges. *IEEE Security & Privacy*, 18(2):56–60, March 2020.

[3] Muhammad Usama Sardar, Thomas Fossati, and Simon Frost. SoK: Attestation in Confidential Computing. *ResearchGate pre-print*, January 2023.

[4] Eric Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446, August 2018.

[5] Frederik Armknecht, Yacine Gasmi, Ahmad-Reza Sadeghi, Patrick Stewin, Martin Unger, Gianluca Ramunno, and Davide Vernizzi. An Efficient Implementation of Trusted Channels based on OpenSSL. In *3rd ACM workshop on Scalable Trusted Computing (STC)*, pages 41–50, October 2008.

[6] Norazah Abd Aziz, Nur Izura Udzir, and Ramlan Mahmod. Extending TLS with Mutual Attestation for Platform Integrity Assurance. *Journal of Communications*, 9(1):63–72, January 2014.

[7] Thomas Knauth, Michael Steiner, Somnath Chakrabarti, Li Lei, Cedric Xing, and Mona Vij. Integrating Remote Attestation with Transport Layer Security. Technical report, Intel Labs, January 2018.

[8] Gordon King and Hans Wang. HTTPA: HTTPS Attestable Protocol. In *Future of Information and Communication Conference (FICC)*, pages 811–823. Springer, March 2023.

[9] Kenneth Goldman, Ronald Perez, and Reiner Sailer. Linking Remote Attestation to Secure Tunnel Endpoints. In *1st ACM workshop on Scalable Trusted Computing (STC)*, pages 21–24, November 2006.

[10] Frederic Stumpf, Omid Tafreschi, Patrick Röder, and Claudia Eckert. A Robust Integrity Reporting Protocol for Remote Attestation. In *Workshop on Advances in Trusted Computing (WATC)*, page 65, November 2006.

[11] Common Vulnerabilities and Exposures: CVE-2014-0160 ("Heartbleed bug"), April 2014.

[12] TLS+RA Source Code on GitHub. https://github.com/Barkhausen-Institut/ratls, May 2025.

[13] Richard Barnes, Jacob Hoffman-Andrews, Daniel McCarney, and James Kasten. Automatic Certificate Management Environment (ACME). RFC 8555, March 2019.

[14] Peter Saint-Andre and Rich Salz. Service Identity in TLS. RFC 9525, November 2023.

[15] Yaron Sheffer, Ralph Holz, and Peter Saint-Andre. Summarizing Known Attacks on Transport Layer Security (TLS) and Datagram TLS (DTLS). RFC 7457, February 2015.

[16] Gordon King and Hans Wang. HTTPA/2: a Trusted End-to-End Protocol for Web Services. In *Future of Information and Communication Conference (FICC)*, pages 824–848. Springer, March 2023.

[17] Robert Walther, Carsten Weinhold, and Michael Roitzsch. RATLS: Integrating Transport Layer Security with Remote Attestation. In *4th Workshop on Cloud Security and Privacy (Cloud S&P)*, pages 361–379. Springer Nature, June 2022.

[18] Hannes Tschofenig, Yaron Sheffer, Paul Howard, Ionuț Mihalcea, Yogesh Deshpande, Arto Niemi, and Thomas Fossati. Using Attestation in Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS), Internet-Draft. https://datatracker.ietf.org/doc/draft-fossati-tls-attestation/09/, April 2025.

[19] Henk Birkholz, Dave Thaler, Michael Richardson, Ned Smith, and Wei Pan. Remote ATtestation procedureS (RATS) Architecture. RFC 9334, January 2023.

[20] N. Asokan, Valtteri Niemi, and Kaisa Nyberg. Man-in-the-Middle in Tunnelled Authentication Protocols. In *11th International Workshop on Security Protocols*, pages 28–41. Springer, April 2003.

[21] Eric Rescorla. Keying Material Exporters for Transport Layer Security (TLS). RFC 5705, March 2010.

DDLUMOS: Understanding and Detecting Atomic DDL Bugs in DBMSs

Zhiyong Wu
KLISS, BNRist, School of Software
Tsinghua University, China

Jie Liang *
School of Software
Beihang University, China

Jingzhou Fu
KLISS, BNRist, School of Software
Tsinghua University, China

Wenqian Deng
KLISS, BNRist, School of Software
Tsinghua University, China

Yu Jiang *
KLISS, BNRist, School of Software
Tsinghua University, China

Abstract

Atomic Data Definition Language (Atomic DDL) is fundamental in DBMSs, ensuring that schema modifications are executed completely or not at all, preserving database integrity. Despite their critical importance, bugs persist in the Atomic DDL, leading to severe consequences such as data corruption and system inconsistencies. However, there is a limited understanding of the characteristics and root causes of these bugs. Furthermore, existing testing methods often fail to effectively identify Atomic DDL bugs, particularly under conditions of high concurrency and unexpected system failures.

This paper presents a comprehensive study of 207 Atomic DDL bugs across three widely used DBMSs. It reveals that Atomic DDL bugs primarily manifest as incorrect results, post-recovery data inconsistency, and system unavailability, which are mainly triggered by metadata conflicts between DDL statements. Based on these findings, we developed DDLUMOS, a testing tool that detects Atomic DDL bugs with metadata conflict-guided DDL synthesis and graph-based consistency analysis. We applied DDLUMOS to six popular DBMSs (e.g., PostgreSQL and MySQL) and found 73 previously unknown Atomic DDL bugs. DBMS vendors responded promptly, fixing 14 issues, highlighting the effectiveness of DDLUMOS in improving the reliability of DBMSs.

1 Introduction

Atomic Data Definition Language (Atomic DDL) is foundational in Database Management Systems (DBMSs), ensuring that schema changes (e.g., creating, altering tables) are executed entirely or not at all, thereby preserving database integrity [22, 23]. Traditional DDLs modify the data structure immediately without rollback mechanisms, so any interruption or failure can leave the DBMS in an inconsistent state.

To address this flaw, modern DBMSs such as MySQL and PostgreSQL introduce Atomic DDL mechanism, ensuring that schema modifications either fully succeed or leave the DBMS in its original state, thus preventing partial updates that could cause inconsistencies or corruption.

```
CREATE TABLE t1 (c1 int,);
CREATE TABLE t2 (c2 int,  FORIGEN KEY(c2) REFERENCE c1);
INSERT INTO t1(···);
INSERT INTO t2(···);
ALTER TABLE t1 DROP COLUMN c2, ADD COLUMN c3 INT;
-- Execute failed with [ERROR 1553 (HY000)]: Cannot drop
column 'c1': needed in a foreign key constraint
-- Check table t1
-- Expected: t1 only have 1 column c1 due to execution failure
-- Actual: t1 has two columns c1 and c3
```

Figure 1: An Atomic DDL bug in MySQL8.0.

The implementation of Atomic DDL is complex due to the involvement of multiple components(e.g., rollback mechanisms, metadata synchronization, and concurrency control), all of which must work together to ensure atomicity and consistency during schema changes. Consequently, implementation errors in the Atomic DDL are difficult to avoid. We define *Atomic DDL bugs* (**ADBs**) as issues that arise during the execution of Atomic DDL operations, which can lead to partial updates, data corruption, or inconsistencies within the DBMSs. For example, Figure 1 illustrates an Atomic DDL bug in MySQL 8.0, which is caused by the implementation errors in rollback mechanisms. The `ALTER TABLE` statement tries to drop column `c1` and add a new column `c3` on table `t1`. However, it fails due to the foreign key constraint. Therefore, table `t1` would still have one column `c1`. However, the `ALTER TABLE` statement is not correctly rolled back, resulting in the `t1` table having two columns (i.e., `c1` and `c3`), which is inconsistent with the expected state. ADBs have distinct characteristics that make them particularly problematic:

(1)*Widespread Impact:* ADBs have a widespread impact due to their role in maintaining the consistency and integrity of the database schema. When these bugs occur, they can propagate inconsistencies across multiple layers of the DBMS. In a typical scenario, a partially executed `CREATE TABLE` operation caused schema discrepancies that disrupted not only the

*Jie Liang and Yu Jiang are the corresponding authors.

current statement but also subsequent DDL or DML operations. For instance, among the recently reported 119 Atomic DDL bugs in MariaDB, 48% of them affected subsequent operations beyond the initial DDL statement, leading to cascading failures in query execution and schema management.

(2)*High Severity:* ADBs pose significant threats to DBMS stability, often leading to severe consequences such as system crashes, data corruption, and even permanent data loss. For example, among the recent 50 CVEs related to Atomic DDLs in MySQL [30], 29 CVEs scored over 7.0. Among them, 28% ADBs caused system crashes, while 33% ADBs resulted in data corruption or data loss. In 15% of cases, the damage was irreversible, causing prolonged downtime and necessitating database restoration from backups.

(3)*Complex and Hard to Detect:* The complexity of ADBs lies in the intricate interactions between different schema elements during DDL operations. Additionally, many ADBs can only be triggered under specific conditions (e.g., specific sequences of DDL operations), which are not easily replicated during standard testing. This complexity makes them particularly challenging to isolate and resolve. Moreover, these bugs are difficult to detect because they might not cause immediate or obvious failures; instead, they may lead to latent inconsistencies that only become apparent under certain query patterns or after further schema modifications.

Despite the substantial damage ADBs can cause, there is a lack of testing methods for ADBs. Current DBMS testing approaches, such as fuzzing and random query generation, primarily target functional correctness [35–37], performance bugs [15], and memory safety [16, 39, 46]. They mainly focus on generating complex SQL syntax structures or sequences of SQL operations to trigger more behaviors in DBMS components. However, due to the limited understanding of the specific manifestations and trigger conditions of ADBs, existing works struggle to identify them.

To understand and detect ADBs, we conducted a comprehensive study of 207 ADBs across three widely used DBMSs, namely PostgreSQL, MySQL, and MariaDB. Our research focused on identifying the manifestations of ADBs, the root causes of the ADBs, and the conditions that trigger these bugs. Our investigation revealed that approximately 44% of ADB lead to incorrect results, 32% cause post-recovery data inconsistency, and 24% result in system unavailability. We found that about 32%, 22%, 18%, and 22% of ADBs are caused by the implementation errors in rollback mechanisms, metadata synchronization, concurrency control subsystem, and error handling mechanisms, respectively. More importantly, we found that 94% of ADBs are triggered by the metadata conflicts between DDL statements.

Building on these findings, we developed DDLUMOS, a testing tool that detects ADBs with metadata conflict-guided DDL synthesis and graph-based consistency analysis. DDLUMOS was applied to six DBMSs, uncovering 73 ADBs. DBMS vendors responded promptly, fixing 14 bugs. Meanwhile, DDLUMOS can rediscover 94.7% studied bugs in one week. Moreover, we compare DDLUMOS with state-of-the-art DBMS testing tools. The 48-hour results show that DDLUMOS detects 27, 31, 32, and 26 more bugs than SQLancer, SQLsmith, SQUIRREL, and TXCHECK, respectively. In summary, this paper makes following contributions:

1. We conduct a comprehensive analysis of 207 ADB across three widely used DBMSs, revealing that ADBs manifest as incorrect results, post-recovery data inconsistency, and system unavailability.

2. We identify the root cause and trigger conditions that lead to ADBs, particularly arising from the metadata conflicts of DDL operations. Based on the findings, we developed DDLUMOS to detect the ADBs.

3. DDLUMOS uncovered 73 previously unknown ADBs in six popular DBMSs. 20 bugs have been fixed, demonstrating the effectiveness of the tool in improving the reliability and consistency of the DBMS.

2 Preliminaries of Atomic DDL

Definitions. Database Management Systems (DBMSs) are software that allows users to define, create, maintain, and control access to databases [1]. *Structured Query Language (SQL)* is a standard language for interacting with DBMS [41], used to create, modify, and query relational databases. *Data Definition Language (DDL)* is a subset of SQL that is used to define and manage the structure of database objects like tables and indexes. It plays a key role in schema design and maintenance. For example, the *CREATE TABLE* statement in DDL is used to define a table and its columns, specifying data types and constraints Atomic DDL arises from the risk of partial updates during schema modifications, which can cause data corruption and system inconsistencies. Traditional DDL operations may fail midway, leaving the database in an inconsistent state. Atomic DDL addresses this by ensuring that schema changes either fully succeed or fail without residual effects, preserving database stability and reliability.

There are two main implementations for Atomic DDL: *Atomic DDL Statement*, where individual DDL statements are treated as atomic transactions by the DBMS, ensuring either full success or full rollback (e.g., MySQL); and *Atomic DDL Transaction*, which allows grouping a series of DDL statements into a single atomic transaction, enabling multiple DDL statements to commit or rollback together (e.g., PostgreSQL).

Key Features. Atomic DDLs are designed to ensure that schema changes are executed in a reliable, consistent, and robust manner. They have the following features:

Comprehensive Rollback Mechanism. Atomic DDL ensures that any changes made during a schema modification are either fully applied or fully rolled back. This all-or-nothing approach prevents partial updates that could otherwise leave

the database in an inconsistent state. For instance, if a table alteration operation fails midway due to a system error, Atomic DDL automatically rolls back the changes, restoring the DBMS to its original state before the operation began. This comprehensive rollback mechanism is crucial for safeguarding data integrity and consistency, especially in complex production environments.

Concurrency Control. In environments where multiple users may attempt to modify the schema simultaneously, Atomic DDL plays a critical role in managing concurrency. It uses locking mechanisms to ensure that only one operation can alter a specific part of the schema at any given time, preventing conflicts and potential corruption. This control is vital for avoiding race conditions and ensuring that all schema changes are applied in a controlled and predictable manner.

Failure Handling and Recovery. One of the key strengths of Atomic DDL is its robust failure handling and recovery mechanism. In case of failure, it ensures no partial changes remain by using detailed logging and transactional controls to track schema modifications. This enables precise rollbacks or retries, protecting the database from corruption while streamlining the recovery process, reducing downtime, and ensuring operational continuity.

3 Methodology

To better understand the characteristics of Atomic DDL Bugs (ADBs), we investigate ADBs across three widely-used DBMSs: PostgreSQL, MySQL, and MariaDB from these DBMSs' issue tracker systems and commit histories [2, 3, 6, 7, 11]. The number of ADB issues we collected is summarized in Table 1.

Table 1: The numbers of collected ADBs in selected DBMSs.

DBMSs	PostgreSQL	MySQL	MariaDB	Total
Studied Issues	351	301	2103	2755
Unique Bugs	38	50	119	207

These DBMSs, developed over long periods (e.g., 27 years for MySQL), often contain a large number of issues (e.g., over 109k issues in MySQL). Manually inspecting all these issues to identify ADBs is time-consuming and challenging. Therefore, we apply filtering rules to identify relevant issues in the recent 10 years. Since DBMS developers typically do not label issues as related to ADBs, we use keywords like "atomic", "DDL", "CREATE", "Atomic DDL", and their variations to retrieve potentially relevant issues. We then manually review the bug descriptions and developer comments, excluding those that do not involve Atomic DDL mechanisms. As a result, we find 2755 related issues. Then, we filter these issues into two steps. First, we exclude bugs labeled as "duplicate" to refine our dataset. Second, we manually reproduce each bug by extracting the provided SQL statements and transaction sequences, ensuring that only distinct issues are included in our study. Through this process, we identified 38, 50, and 119 ADBs in PostgreSQL, MySQL, and MariaDB, respectively.

Threats to Validity. Similar to other studies, our research has several limitations that should be considered.

Invisibility of high-severity vulnerabilities. High-severity vulnerabilities, such as those leading to data breaches or unauthorized access, are often not disclosed publicly by security teams. For example, severe security issues in MySQL and PostgreSQL are typically reported directly to their security teams and are not made available in public bug trackers. Consequently, our study may miss some of the most critical vulnerabilities, potentially limiting the completeness of our findings. However, the focus on transaction-related bugs still provides valuable insights into detecting atomicity issues.

Representativeness of selected DBMSs. Our study focused on three popular relational DBMSs: PostgreSQL, MySQL, and MariaDB. While these systems represent a significant portion of the market, our findings may not fully apply to other databases, such as NoSQL or graph databases, with different transaction models and atomicity guarantees. Nevertheless, the insights gained could still provide useful guidance for understanding atomicity in various DBMS.

4 General Findings of ADBs

In this Section, we analyze 207 collected unique ADBs to identify their common characteristics, focusing on the manifestations, root cause, and trigger conditions of ADBs.

4.1 Manifestations of ADBs

To understand the potential damage of ADBs, we examined the manifestations of the collected bugs by manually reviewing the bug descriptions and reproducing them. Regarding the manifestations of ADBs, we have the following findings:

Finding 1. *Among the studied bugs, about 44% (91/207) of them manifest as incorrect results, about 32% (66/207) of them manifest as post-recovery data inconsistencies, and about 24% (50/207) of them manifest as system unavailability.*

Incorrect Result. Our investigation indicates that approximately 44% of the ADBs result in incorrect outcomes, undermining the accuracy and reliability of data returned to users. These incorrect results contain various forms, such as schema errors, metadata mismatches, trigger errors, function errors, and constraint violations. Such manifestations arise when a DDL operation (e.g., `ALTER TABLE` or `CREATE INDEX`) aborts but fails to roll back all of its catalog mutations, leaving the database in a partially updated state. As a result, query results may be incorrect due to a misalignment between the database schema and the intended structure. For instance, the `ALTER TABLE` statement in Figure 1 fails to execute but still modifies the metadata of table `t1`, resulting in the incorrect result.

Post-Recovery Data Inconsistency. Post-recovery data inconsistency is a prominent issue in ADB, potentially leading to long-term corruption or loss of critical information, occurring in 32% (66/207) studied cases. This problem typically arises during system recovery when a DDL operation is interrupted by a fault (e.g., power outage), and the DBMS attempts to restore the previous state upon restart. If the atomicity of the DDL operation is not properly maintained, the recovery process may fail to fully restore the schema, leading to the loss of recently added or modified schema objects. Figure 2 illustrates an example of post-recovery data inconsistency in MariaDB. In this case, table `t1` is modified through several DDLs, such as adding and dropping columns, when the server process is unexpectedly terminated. Upon restart, MariaDB shows that table `t1` does not exist, indicating data loss.

```
(1) Create initial add and add constraints
CREATE TABLE t1 (col1 INT, col2 INT, col3 INT,
col4 TEXT) ENGINE = InnoDB ;
ALTER TABLE t1 ADD PRIMARY KEY(col4(10),col2);
INSERT INTO t1 (col2,col4) VALUES(2, 'data');
(2) Runs random alter statements in the same session
ALTER TABLE t1 ADD COLUMN extra INT;
ALTER TABLE t1 DROP COLUMN col3 ;
ALTER TABLE t1 CHANGE COLUMN extra col3 INT ;
(3) Kill the server process at some point of time
(4) Restart the server and show tables;
SHOW KEYS FROM `test`.`t1` harvests;
⚠ ERROR[1932]: Table 'test.t1' doesn't exist
```

Figure 2: A post-recovery data inconsistency in MariaDB.

System Unavailability. The complete system unavailability is a severe manifestation of ADBs, causing significant downtime and affecting the accessibility of the DBMS, accounting for 24% (50/207) studied cases. This includes issues such as system crashes, assertion failures, memory errors, and system hangs, where the entire system becomes unresponsive or fails to recover, causing prolonged downtime and potential data loss. This typically occurs when the DBMS engine encounters an unhandled exception or fatal error during the execution of a DDL operation. The instability introduced by these bugs makes them particularly dangerous, as they can compromise the availability and reliability of DBMSs.

4.2 Root Cause of ADBs

We manually checked each bug report and further identified several underlying causes that directly contribute to the manifestations described earlier. We discovered that all the ADBs are caused by the implementation errors of Atomic DDL, which are inherently complex due to the intricacies involved in ensuring atomicity, consistency, and rollback mechanisms during schema modifications. We categorize the root causes into four categories, and we have the following findings:

Finding 2. *Among the studied bugs, four main causes of ADBs were identified, with the following distribution: incomplete rollback mechanisms accounted for 32%, improper synchronization for 22%, inadequately implemented concurrency control for 18%, and incorrect error handling for 22%.*

The four categories collectively account for 94% of the studied ADBs. Figure 3 illustrates how frequently each root cause appears in ADBs manifesting as incorrect results, data loss, or system crashes. Inappropriate metadata synchronization is the most common trigger for incorrect results, while inadequate rollback mechanisms frequently lead to data loss, and faulty error handling is often responsible for system crashes. Besides them, other ADBs (6%) are caused by implementation errors in various auxiliary mechanisms, such as improper handling of system-level configurations, and rare edge cases in DBMS extensions or plugins.

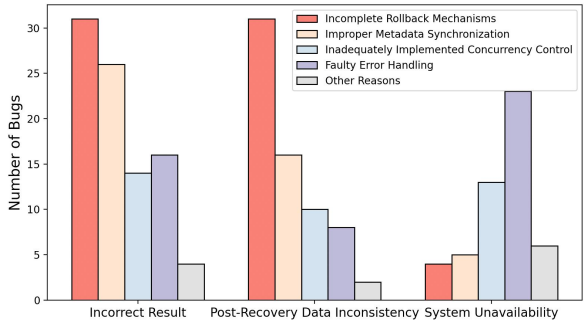


Figure 3: Statistic of bugs and root causes.

Incomplete Rollback Mechanisms. The implementation error in the rollback mechanisms is a prevalent root cause for ADBs, responsible for 65 bugs, including 31 incorrect results and 31 post-recovery data inconsistencies. These bugs arise when a failed Atomic DDL operation leaves the schema in an inconsistent state due to incomplete rollbacks. For example, in MDEV-25506 of MariaDB, the server is forcibly crashed while executing `CREATE TABLE IF NOT EXISTS tt11 AS SELECT...` statement. After crash recovery, both `DROP TABLE tt11` statement and a new `CREATE TABLE tt11` statement fail to execute successfully with error messages. The drop statement returns “Unknown table tt11” while the create statement complains that the "tablespace tt11 already exists". The main reason is that InnoDB discovers both tt11.frm and tt11.ibd on disk, yet the internal data dictionary believes the table is already being dropped and raises an error that test.tt11 does not exist [24]. The root cause of the bug is a flaw in the rollback mechanism of Atomic DDL recovery: the DDL log rolled back the dictionary-level create operation but failed to purge the tablespace, leaving the recovery state inconsistent.

Improper Metadata Synchronization. The implementation errors in synchronization between schema metadata and physical data structures cause 47 ADBs, including 26

incorrect result cases. This issue arises when metadata is updated to reflect a schema change, but the corresponding data structures remain unchanged due to an interruption in Atomic DDL operations. For example, in bug#18570 of PostgreSQL, when an `DROP EVENT TRIGGER` statement appears to succeed in detecting the trigger, while the event trigger remains active in memory. After the command, catalog queries such as `information_schema.triggers` show no trigger remains, yet every subsequent DDL statement still invokes the trigger's function; meanwhile, attempting to drop that function fails because the system claims the (supposedly deleted) trigger depends on it [33]. The mismatch between system catalogs and the in-memory trigger cache creates a "ghost trigger" that blocks normal maintenance until the function is forcibly dropped, breaking Atomic DDL semantics.

Inadequately Implemented Concurrency Control. The implementation errors in the concurrency control subsystem represent another major root cause, accounting for 37 ADBs. These issues typically arise when multiple DDL operations or concurrent transactions contend for the same resources, such as locks on tables or indexes. For example, in BUG#116502 of MySQL, during an online build of a unique index that includes a `NULL-able` column, MySQL 's Atomic DDL mishandles a concurrent sequence of `DELETE → ROLLBACK` combined with an `INSERT`. The engine incorrectly treats rows containing NULL as duplicates, so the rollback cannot re-insert the original row, and the entire DDL aborts—violating its all-or-nothing guarantee [28].

Faulty Error Handling. Faulty error handling during DDL operations is another root cause for ADBs, accounting for 18% of incorrect result cases and 46% of system unavailability. This issue occurs when the DBMS fails to properly handle errors or exceptions that arise during DDL operations. For instance, in MDEV-13205 of MariaDB, an unexpected error happens during the executing the `ALTER TABLE` statements to add unique key and foreign key [25]. The `ALTER TABLE t2 ADD UNIQUE(c)` is aborted due to a duplicate key error, but the DBMS fails to remove the partially created index stub from the data dictionary. Because that orphaned metadata is left intact, executing `ALTER TABLE t3 ADD FOREIGN KEY (c) REFERENCES t2(c)` statement will consult the corrupted data dictionary, trigger the assertion !dict_index_is_online_ddl(index), and finally crash the server.

4.3 Trigger Conditions of ADBs

Following the identification of root causes, we analyze the specific conditions that trigger ADBs.

Finding 3. *Most ADBs (approximately 93%) are triggered in scenarios involving data conflicts between DDL statements, where multiple DDL statements operate or interact with the same metadata elements in the test case.*

We define **metadata conflict points** as the specific locations in test cases where multiple DDL statements operate or interact with the same metadata elements, such as a table, column, index, or constraint. In other words, the conflict points reflect the data dependency between different statements, which can help trigger more code behaviors of Atomic DDL for detecting ADBs. Figure 4 illustrates an example of metadata conflict points. The two `CREATE TABLE` statements establishes two tables(`t1` and `t2`) with three columns and one column, respectively. The subsequent `ALTER TABLE` statement adds two indexes on table `t1`, which will change the three metadata elements `t1`, `t1.c1`, and `t1.c2`. Here, we identify `t1`, `t1.c1`, and `t1.c2` as the 3 metadata conflict points in the test cases. The second `CREATE TABLE` statement does not make any metadata conflict points due to it does not change any existing metadata elements.

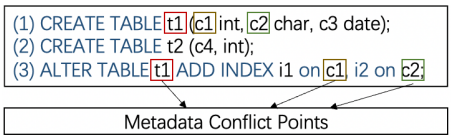


Figure 4: An example of metadata conflict points.

In our study, 193 (93% of 207) studied bugs contain metadata conflict points, and 174 (84% of 207) cases contain over 4 metadata conflict points. Among the 193 studied bugs containing metadata conflict points, 94% (181/193) were directly triggered by metadata conflict points with the corresponding statements. The remaining 6% (12/193) were not directly triggered by conflict points, but the bug-triggering statements have dependency relationships with those conflict points. Our investigation results demonstrate that metadata conflict points are essential to trigger ADBs.

Finding 4. *Incorrect result, post-recovery data inconsistency, and system unavailability have different trigger scenarios and test oracle.*

In our study, incorrect results and system unavailability can occur during DDL execution. Incorrect results are identified by comparing the execution output and metadata. For example, the `ALTER TABLE` statement in Figure 1 should fail without modifying any metadata, which was confirmed by detecting unexpected changes in metadata. System unavailability is detected by checking the server's response after executing the test cases. If the server disconnects or becomes unresponsive, it is confirmed as a bug. In contrast to incorrect results and system unavailability, post-recovery data inconsistency occurs when the server is restarted mid-Atomic DDL execution. If the recovery metadata differs from the expected state after rollback, developers confirm it as a bug.

4.4 Limitations of Existing Methods

According to the conditions outlined in Section 4.3, detecting ADBs requires generating DDL statements that involve more

metadata conflict points. Moreover, these test cases must be examined in various scenarios to detect the three manifestations: incorrect results, system crashes, and post-recovery data inconsistency. However, existing DBMS testing tools are limited in both aspects.

First, most existing tools focus on generating DQL (Data Query Language) statements, with minimal attention paid to DDLs. Even when DDLs are included, they are generated with few metadata conflict points. For example, SQLsmith [39] and EET [14] focus on generating `SELECT` queries, and the DDLs they produce are typically simple, making it difficult to simulate the complex interactions between different DDL operations. Similarly, SQLancer [37] generates equivalent queries to detect logic bugs by transforming SQL expressions, which will overlook potential conflicts between DDLs.

Moreover, existing testing tools struggle to detect ADB manifestations, particularly those resulting in inconsistencies. For instance, when a DDL operation is interrupted mid-execution, the DBMS may enter an inconsistent state with only partial changes applied—a scenario that current tools fail to detect. Furthermore, these tools also struggle to identify data inconsistency after recovery, as this requires precise tracking of Atomic DDL execution and metadata modifications, capabilities that are often lacking in existing approaches.

Following the findings, we design DDLUMOS which targets ADBs with metadata conflict-guided DDL synthesis and graph-based consistency analysis. The main idea is to analyze DDL dependencies to generate more metadata conflict points, while also designing detection methods for three manifestations. Specifically, DDLUMOS generates more metadata conflict points by tracking schema elements and guiding test case generation through a metadata table. On the other hand, it constructs a metadata graph to analyze schema inconsistencies and identify ADBs based on predefined detection scenarios.

5 Design of DDLUMOS

Figure 5 illustrates the workflow of DDLUMOS, which consists of two key modules: an *Test Case Generator* for synthesizing high-quality DDL sequences and a *Metadata Consistency Analyzer* for detecting ADBs. The process follows four main steps: In Step ①, DDLUMOS initializes a metadata table to record schema elements used during test case generation, as well as the metadata conflict points associated with each DDL operation. In Step ②, DDLUMOS alternates between generating DDL statements and other SQL statements. After each DDL operation, DDLUMOS analyzes the resulting metadata conflict points and updates the metadata table accordingly, guiding subsequent DDL generation to maximize conflict scenarios. In Step ③, DDLUMOS constructs a metadata graph for the generated test cases and sends them to the target DBMS. The metadata graph captures the relationships between schema elements before and after DDLs, providing a structural foundation for analyzing schema consistency and correctness. In Step ④, DDLUMOS uses the metadata graph to detect ADBs by comparing the execution results with predefined detection scenarios for three bug manifestations.

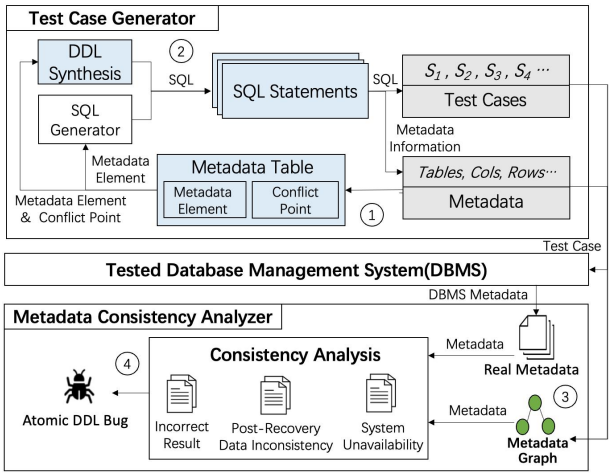


Figure 5: The workflow of DDLUMOS. It includes two main components: (1) Test Case Generator for generating test cases with metadata conflict guided DDL synthesis. (2) Metadata Consistency Analyzer for detecting atomic DDL bugs with graph-based consistency analysis.

5.1 Metadata Conflict-Guided DDL Synthesis

Metadata Conflict-Guided DDL Synthesis generates test cases that maximize metadata conflict points to trigger ADBs effectively. This approach consists of two components: Conflict Point Tracking and Test Case Synthesis, which work together to trigger deep behaviors of Atomic DDL.

Conflict Point Tracking. Based on the findings in Section 4.3, incorporating *metadata conflict points* into DDL test cases significantly increases the likelihood of exposing ADBs. To achieve this, DDLUMOS continuously collects and updates *Metadata Information* during test case generation, using these details to guide the synthesis of subsequent DDL operations. Specifically, DDLUMOS maintains a metadata table that records critical information about schema elements relevant to conflict scenarios, including the DDL identifier (e.g., operation type), table name, column, index, and constraint data. After each DDL statement executes, DDLUMOS updates the corresponding entries in the metadata table based on the statement type (create, alter, or drop) and the targeted objects (tables, columns, indexes, or constraints).

For example, as shown in Figure 6, when processing the 5th `ALTER TABLE` statement, DDLUMOS identifies the affected table (i.e., t2, t1) in the metadata table and increments its "conflict point" field by the value introduced by that operation (e.g., t1(1)). Likewise, if new columns or indexes are added or removed(e.g., 6th statement), DDLUMOS revises the corresponding fields to reflect the latest state and constraints.

Synthesized Test Case

(1). CREATE TABLE t1 (c0 INT, c1 FLOAT);
(2). INSERT INTO t1 () VALUES (1) ,(2);
(3). CREATE TABLE t2 (c3 CHAR, c4 INT);
(4). CREATE INDEX i1 on t1(c0);
(5). ALTER TABLE t2 ADD FOREIGN KEY(c4) REFRENCE t1(c0);
(6). ALTER TABLE t2 ADD COLUMN c5 INT, ADD COLUMN c6 DATE;

⇩ Metadata Information Collection

	Type	Metadata	Conflict Points
(1)	CREATE TABLE	t1 | t1.c0 | t1.c1	NULL
(2)	NOT-DDL	t1.c0 | t1.c1	NULL
(3)	CREATE TABLE	t2 | t2.c3 | t2.c4	NULL
(4)	CREATE INDEX	t1.i1	t1(1), t1.c0(1)
(5)	ALTER TABLE	t2.c4 | t1.c0	t1(1),t2(1),t2.c4(1),t1.c0(1)
(6)	ALTER TABLE	t2.c5 | t2.c6	t2(2)

Figure 6: An example of metadata information collection.

This metadata-driven update process accurately captures how the schema evolves over time and provides precise conflict guidance for generating subsequent test cases.

Test Case Synthesis. By tracking conflict points in the metadata table, DDLUMOS synthesizes test cases that strategically exploit these conflict points to maximize the likelihood of uncovering atomic DDL bugs. During test-case generation, DDLUMOS interweaves DDL statements with DML (e.g., `INSERT`, `UPDATE`) and DQL (e.g., `SELECT`) statements, creating comprehensive and high-coverage input sequences. DDLUMOS generates statements of each test case iteratively: First, it determines the statement type. Second, it builds the skeleton of the statement based on its type and then populates objects into the skeleton with the metadata table.

Algorithm 1: Test Case Synthesis with Metadata Table

Input : L: The length of test cases
MT: The metadata table
Output : T: The test cases

```
1  T = initalTestCase ();
2  while L > 0 do
3      M_e, C_p = getMetadataAndConlictPoint(MT);
4      S_type = pickStatementType(T);
5      if S_type = DDL then
6          M_u = sortMetadataWithConflictPoint(M_e);
7          S = synthesizeDDL(M_u, C_p);
8      end
9      else
10         S = synthesizeDMLorDQL(M_e);
11     end
12     MT = updateMetadataTable(MT, S);
13     T = addToTestCase(T, S);
14     L = L − 1;
15 end
```

Algorithm 1 details the process of test case synthesis with the metadata table. DDLUMOS first initializes an empty test case(T) and then iterates to synthesize the statement until the desired length of the test case (L) is reached (Line 1-2). The length of the test case(L) is critical for the performance of DDLUMOS. We will give the specific length of the test case(L) in Section 6. In each iteration, DDLUMOS first retrieves the current metadata state (M_e) and conflict point information (C_p) from the metadata table (MT) (Line 3). Next, it determines the type of the next SQL statement to be synthesized (S_{type}) (Line 4). Specifically, DDLUMOS uses a predefined ratio (80% for DDL, 20% for other types in our implementation) to determine whether to generate DDL statements. If true, DDLUMOS then randomly selects the least frequently used DDL statement types (e.g., ALTER TABLE, CREATE INDEX) to synthesize, ensuring diversity in the generated statements. Otherwise, DDLUMOS similarly randomly chooses from the available types of non-DDL statements, maintaining variability in the types of statements generated. If the statement type is DDL, DDLUMOS sorts the existing metadata by conflict point frequency and synthesizes the DDL with stored metadata(M_u) and corresponding conflict points(Lines 5-8). Otherwise, if the selected statement type is DML or DQL, DDLUMOS uses the existing metadata(M_e) to synthesize to ensure semantic correctness (Lines 9-11). The implementation details of the DDL synthesis process can be found in Section 6. Once the statement (S) is generated, DDLUMOS updates the metadata table (MT) to reflect changes introduced by the new statement and adds it to the test case (T) (Lines 12-13).

5.2 Graph-Based Consistency Analysis

After generating test cases using Metadata Conflict Guided DDL Synthesis, DDLUMOS distributes the test cases across multiple client threads, while controlling concurrency based on the sequence of these statements (Details can be found in Section 6). Besides, DDLUMOS employs *Graph-Based Consistency Analysis* to verify the correctness of test case execution results. This analysis involves two critical steps. First, for a test case, DDLUMOS constructs a metadata graph that represents the schema elements and their relationships before and after executing each DDL statement. Then, based on the metadata graph and DDL execution scenario, DDLUMOS analyzes the metadata state resulting from the DDL operations. It compares the expected metadata derived from the graph with the actual metadata in the database after execution. Any discrepancies between the expected and actual metadata indicate a potential atomic DDL bug.

Metadata Graph Construction. For each test case, DDLUMOS constructs a metadata graph that represents the metadata state of the DBMS both before and after executing each SQL statement in the test cases. Unlike a full dataset dump, this metadata graph focuses on essential metadata information, which can be influenced by DDL operations, including tables, columns, constraints, triggers, row counts, indexes, views, as well as their relationships. Each element of the metadata information is modeled as a node in the graph, while edges represent dependencies or hierarchical relationships.

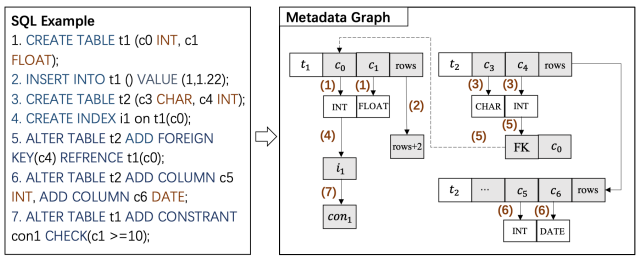


Figure 7: An example of metadata graph construction.

For instance, Figure 7 demonstrates the construction process of the metadata data graph with a test case in MySQL. The left side of the figure shows the SQL statements executed sequentially in the order(e.g., `(1)->(2)->(3)->...`). The right side illustrates the effects of these statements on the graph structure (such as the data structure, dependencies, etc.). The numbers(e.g., (1), (2)) on the arrows indicate which SQL statement generated each corresponding effect.

For statement (1), DDLUMOS first creates a node for table `t1` along with child nodes for each column (`c0` and `c1`), connecting the parent table node to its columns. In addition, DDLUMOS initializes a `rows` node that records the row count of `t1`. When statement (2) inserts records into `t1`, DDLUMOS updates the `rows` node accordingly. Similarly, statement (3) adds another table node (`t2`) and its corresponding column nodes (`c3` and `c4`). Attributes such as data types, constraint types, and dependency properties are also stored within nodes and edges, ensuring comprehensive tracking of schema changes. For example, when statement (4) creates index `i1` on `c0` of `t1`, DDLUMOS adds an `i1` node to the metadata graph and draws a direct edge from the existing `c0` node to this new `i1` node. When statements (4), (5), and (7) further alter the metadata of `t1` and `t2`, DDLUMOS adds or updates the corresponding nodes and edges to capture the new information and relationships and reflect these changes in the metadata graph. Throughout the entire test execution process, the metadata graph is dynamically updated to provide an accurate and consistent representation of the schema's state at each stage.

Metadata Consistency Analysis. After constructing the metadata graph, DDLUMOS evaluates the correctness of DDL operations by comparing it with the actual metadata state after each operation. Specifically, the metadata information(e.g., schema information, tables, columns, row counts, triggers, and indexes) stored in the metadata graph should align precisely with the metadata in the real databases. For each DDL statement, DDLUMOS identifies affected metadata objects and checks for inconsistencies between the metadata graph and their actual counterparts in the DBMS. Discrepancies are flagged as potential bugs.

For example, the sixth `ALTER TABLE` statement in Figure 7,modifies table `t2` along with columns `c5` and `c6`. DDLUMOS first determines that this operation could affect the metadata for `t2, c5`, and `c6`, and updates the metadata graph accordingly. More concretely, upon executing the sixth statement, the metadata graph adds two child nodes `c5` and `c6` under the node `t2`, including their data types. Once the database finishes processing this statement, DDLUMOS retrieves the relevant metadata from the graph (namely the node information for `t2, c5`, and `c6`), and compares it with the actual metadata stored in the DBMS. Any mismatch discovered in this process is reported as a potential bug.

By comparing the metadata graph with the actual metadata stored in the database, DDLUMOS can evaluate the correctness of DDL operations. However, as highlighted in Section 4.1, ADBs manifest in three distinct forms: incorrect results, system unavailability, and post-recovery data inconsistencies, each arising from different triggering scenarios. To comprehensively detect ADBs, DDLUMOS extends its graph-based consistency analysis to incorporate the detection scenarios described in 4.3.

Algorithm 2: Detect ADBs with Consistency Analysis

Input : T: The test case to be executed
K: The kill signal to simulate the crash
G: Initial metadata graph
Output : The ADBs detected by DDLUMOS

```
1  S ← splitToStatements(T)
2  foreach s ∈ S do
3      G ← constructGraph(G, s)
4      M ← executeToGetMetadata(s)
5      if isSimulateCrash(K) = false then
6          if connectionCheck () = false then
7              return System Unavailability
8          end
9          if !ConsistencyAnalysis(M, G) then
10             return Incorrect Result
11         end
12     end
       // asynchronously trigger crash
13     spawn(AsyncCrashRecovery(K, G))
14 end
15 procedure AsyncCrashRecovery(K, G):
16     sendKillSignal(K)
17     M_n ← RebootandGetMetadata()
18     if !ConsistencyAnalysis(M_n, G) then
19         return Post-Recovery Data Inconsistency
20     end
```

Algorithm 2 details the process of identifying ADBs with these three manifestations. Given a test case generated with metadata conflict-guided DDL synthesis, DDLUMOS first splits it into individual SQL statements (Line 1). For every statement, DDLUMOS constructs and updates the metadata graph to represent the expected metadata state after the execution. DDLUMOS will also send the statement for execution and get real metadata information from the DBMS (Lines 3-4). Following the SQL execution, DDLUMOS systematically evaluates the three types of Atomic DDL bugs: system un-

availability, incorrect results, and post-recovery data inconsistencies. To detect system unavailability, DDLUMOS performs a connection check to assess the server's responsiveness. If the system becomes unresponsive during execution, DDLUMOS identifies this as a system unavailability issue (Lines 6-8). For detecting incorrect results, DDLUMOS compares the actual metadata state (M) retrieved from the database with the expected metadata graph (G) following the consistency analysis. Any discrepancies between the actual metadata and the expected graph indicate an incorrect result (Lines 9-11). To identify post-recovery data inconsistencies, DDLUMOS asynchronously simulates a server crash by sending a kill signal to abruptly terminate the database process. After rebooting the server, DDLUMOS retrieves the post-recovery metadata and compares it with the metadata graph. Any mismatches between the recovered metadata and the expected state are flagged as post-recovery data inconsistencies, highlighting failures in the database's recovery mechanism (Lines 13-20). Note that the process of killing the server to simulate the crash is independent and asynchronous from the process of executing the SQL statements and identifying the system unavailability and incorrect results.

6 Implementation

Based on our approach, we realized DDLUMOS. The overall codebase consists of 10k lines of C++ code, 4k lines of Bison and Flex code, and 1k lines of Python code. Then, we explain some other implementation details, which we consider significant for the outcome.

Effort to Adaption. Adapting DDLUMOS to a new DBMS requires minimal effort and involves two primary steps. First, we need to write the metadata information query statements retrieve schema elements such as tables, columns, indexes, and constraints. This step typically requires no more than 10 lines of SQL, tailored to the specific metadata structures of the DBMS under test. Second, we find the SQL grammar file (e.g., Yacc files) of the target DBMS for grammar adaption. DDLUMOS can leverage these grammar files to automatically adapt to the SQL operators of the target DBMS with a grammar adaptor(implemented with python and bison code).

Length of Test Case Setting. The length of test cases generated by DDLUMOS is critical to the testing performance. Overly long test cases will increase the storage and computational burden of maintaining and analyzing the metadata graph, thereby reducing testing efficiency. Conversely, overly short test cases may fail to adequately trigger the deeper logic associated with Atomic DDL operations. We analyze the test cases of 207 studied bugs and we observed that 98% of them contained no more than 15 SQL statements. Consequently, we set the length of test cases to 15 SQL statements in our testing framework to balance efficiency and effectiveness.

Details of DDL Synthesis. Synthesizing DDLs with the picked statement type and metadata table contains two steps. (1)DDLUMOS first constructs a DDL skeleton based on the chosen statement type (e.g., `ALTER TABLE [`*table_name*`] ADD COLUMN [`*column*1 *datatype*`], ADD COLUMN [`*column*2 *datatype*`]...)`. The clause of this skeleton is randomly generated(e.g., `ADD COLUMN [`*column*1 *datatype*`]`. To increase potential conflict points, each skeleton contains at least two objects (named metadata elements such as *table_name*). (2)DDLUMOS populates the skeleton objects (e.g., *table_name*) with entries (e.g., `t1, t1.c1`) from the metadata table, which tracks existing metadata elements along with their corresponding conflict point counts. During the population, entries with fewer conflict points are selected more frequently, and these chosen entries must conform to the relevant data type semantics. If DDLUMOS cannot find suitable entries in the metadata table, it generates new entries to fill the skeleton objects.

Distributing Test Cases with Multiple Clients. As the study shows, about 18% of ADBs arise from those implementation errors, and they still manifest as incorrect results, post-recovery data inconsistency, and system unavailability. To cover the codes of the concurrency control mechanism, DDLUMOS constructs concurrent scenarios by leveraging multiple clients to distribute SQL statements, while controlling concurrency based on the sequencing of these statements. Specifically, DDLUMOS tags every SQL statement with a sequence identifier, then launches N client worker threads (typically 2–5) and assigns the statements to them in that predefined order. Note that each client issues its own statements sequentially so that DDLUMOS can reconstruct an accurate metadata graph of the execution. To trigger more behaviors of DBMS's concurrency control mechanism, DDLUMOS occasionally sends a subsequent statement before the completed execution of the previous one, thereby intentionally triggering concurrency control behaviors to detect ADBs in them.

7 Evaluation

To evaluate the effectiveness of DDLUMOS, we conduct experiments to address the following research questions:

- **RQ1:** Is DDLUMOS effective in finding ADBs in real-world DBMSs?
- **RQ2:** How does DDLUMOS's performance compared with state-of-the-art methods?
- **RQ3:** How many bugs collected in the study can be rediscovered by DDLUMOS?

7.1 Evalution Setup

Tested DBMSs. To evaluate the generality and efficiency of DDLUMOS, we select six popular open-source and well-tested DBMSs, namely MySQL [29,43], MariaDB [4,26], Percona [5], PolarDB [32], GreatSQL [10], and PostgreSQL [27, 34]. These DBMSs are also widely used in industry.

Table 2: DDLUMOS discovered 73 bugs in six DBMSs within two weeks. (MySQL: 14, MariaDB: 15, Percona: 14, PolarDB: 11, GreatSQL: 15, PostgreSQL: 4). [**IRM**: Incomplete Rollback Mechanisms, **IMSYN**: Improper Metadata Synchronization, **IICC**: Inadequately Implemented Concurrency Control, **FEH**: Faulty Error Handling, **Others**: Other Root Causes for Bugs.

DBMS	Bug Type and Number	Bug Status	Root Cause
MySQL	Incorrect Result(7)	Confirmed(7), Fixed (1)	IRM(2), IMSYN(1), IICC(1), FEH(1), Others(2)
MySQL	System Unavailability(4)	Confirmed(4), Fixed(1)	FEH(3), IICC(1)
MySQL	Post-Recovery Data Inconsistency(3)	Confirmed(3)	IRM(1), IMSYN(1), IICC(1)
MariaDB	Incorrect Result(8)	Confirmed(8), Fixed(1)	IRM(3), IMSYN(2), FEH(2), Others(1)
MariaDB	System Unavailability(3)	Confirmed(3), Fixed(1)	FEH(1), IICC(1), IRM(1),
MariaDB	Post-Recovery Data Inconsistency(4)	Confirmed(4)	IMSYN(2), IRM(1), FEH(1)
Percona	Incorrect Result(4)	Confirmed(4), Fixed(1)	IRM(1), IMSYN(2), FTH(1)
Percona	System Unavailability(4)	Confirmed(4)	Others(2), FEH(1), IICC(1)
Percona	Post-Recovery Data Inconsistency(6)	Confirmed(6), Fixed(2)	IMSYN(3), IRM(2), FEH(1)
PolarDB	Incorrect Result(4)	Confirmed(4)	IICC(1), FEH(1), IRM(2),
PolarDB	System Unavailability(4)	Confirmed(4), Fixed(1)	FEH(3), Others(1)
PolarDB	Post-Recovery Data Inconsistency(3)	Confirmed(3), Fixed(1)	IRM(3)
GreatSQL	Incorrect Result(7)	Confirmed(7)	IMSYN(3), IRM(3), IICC(1)
GreatSQL	System Unavailability(3)	Confirmed(3), Fixed(1)	FEH(1), IICC(2)
GreatSQL	Post-Recovery Data Inconsistency(5)	Confirmed(5), Fixed(1)	IRM(1), IICC(2), FEH(1), Others(1)
PostgreSQL	Incorrect Result(1)	Confirmed(1)	IMSYN(1)
PostgreSQL	System Unavailability(2)	Confirmed(2), Fixed(2)	FEH(2)
PostgreSQL	Post-Recovery Data Inconsistency(1)	Confirmed(1), Fixed(1)	IRM(1)
Total	**73 ADBs**	**73 confirmed, 14 fixed**	

Basic Setup. The experiments were conducted on a machine running 64-bit Ubuntu 20.04 with an AMD EPYC 7742 Processor @ 2.25 GHz, 128 cores, and 504 GiB of main memory. All DBMSs were tested using docker containers that were downloaded directly from their websites, each with 5 CPU cores and 50 GiB of RAM.

Compared Techniques. We compare DDLUMOS with four state-of-the-art DBMS testing tools, including SQLancer [37], SQLsmith [39], SQUIRREL [46], and TXCHECK [13], which are widely used in industry and detect amounts of bugs SQLancer and TXCHECK detect the logic and transactional bugs, respectively, while SQLsmith and SQUIRREL focus on detecting crashes in DBMSs.

7.2 ADBs in Real-World DBMS

We apply DDLUMOS to six DBMS under test for atomic DDL bug detection evaluation. DDLUMOS successfully detected 73 ADBs in two weeks. Table 2 shows DDLUMOS discovered 14, 15, 14, 11, 15, and 4 ADBs in MySQL, MariaDB, Percona, PolarDB, GreatSQL, and PostgreSQL, respectively. 14 ADBs of them have been fixed due to the high severity. As described in Section 4.1, the 73 ADBs contain three types, 31, 20, and 22 bugs corresponding to incorrect results, system unavailability, and post-recovery data inconsistency, respectively.

Root Cause and Severity. We analyze the root causes of discovered bugs based on the response of the vendor. The right part of Table 2 shows the statistics of ADBs' root causes in six DBMSs. Specifically, DDLUMOS detected 21, 15, 11, and 19 ADBs caused by incomplete rollback mechanisms, improper metadata synchronization, inadequately implemented concurrency control, and faulty error handling, respectively. Besides, 7 bugs are caused by other root causes (e.g., untracked schema element orphaning, errors in parsing DDLs).

Feedback from DBMS Developers. We have actively reported all 73 bugs to the corresponding DBMS vendors and received their confirmation feedback. At the time of the paper writing, 73 ADBs have already been confirmed, and 14 ADBs have been fixed. Besides, 9 bugs have been assigned with CVE IDs due to their severity. More importantly, the detected bugs have garnered the attention of the DBMS developers. For example, when we were reporting bugs to PolarDB, the developers responded for them immediately.

Case Studies. We give the following case studies to demonstrate the effectiveness of DDLUMOS.

Case Study 1: No trigger exists after CREATE OR REPLACE TRiGGER in MySQL. Figure 8 illustrates a case of incorrect result in MySQL. This bug occurs when attempting to create a trigger on one table (`t1`) and then replace it with another table (`t2`) using the `CREATE OR REPLACE TRIGGER` statement. Instead of successfully replacing the original trigger on `t1` with a new trigger on table `t2`, an error is thrown and both triggers disappear finally in DBMS. The root cause is the implementation errors in metadata synchronization, where the new trigger can not be saved correctly due to the metadata inconsistency and return error messages.

```
CREATE TABLE t1 (c1 int );
CREATE TABLE t2 (c1 int );
CREATE OR RPLACE trigger tr0 AFTER INSERT on t1 for
each row set @X =1;
CREATE OR RPLACE trigger tr0 AFTER INSERT on t2 for
each row set @X =1;
-- ERROR 1360 ( HY000 ): Trigger tr0 does not exist
```

Figure 8: A case of incorrect result in MySQL.

Case Study 2: Tablespace id Confilt and Data Loss upon Crash Recovery. Figure 9 illustrates a case of post-recovery

data inconsistency in GreatSQL. This bug occurs when attempting to execute an `ALTER TABLE` statement to modify a table (`t0`) by changing its storage engine. During the execution of this statement, the server is forcibly killed. Upon restarting the server, an error is thrown, indicating a tablespace mismatch. Additionally, table `t0` is found to be empty after the restart, resulting in data loss. The root cause lies in the implementation errors in rollback mechanisms. The modification to table `t0` affects the underlying tablespace information, which does not properly rollback after the crash. As a result, after the server recovery, the data in table `t0` is lost.

```
CREATE TABLE t0(c1 int, c2 char);
INSERT INTO t0(·····);
-- Kill the server when executing below statement
ALTER TABLE t0 ENGINE = InnoDB , ADD COLUMN IF NOT
EXISTS c2 MEDIUMINT UNSIGNED ZEROFILL DEFAULT 2,
LOCK = SHARED
-- Reboot the server
-- [ ERROR ] InnoDB : Expected tablespace id 43
but found 0 in the file ./db/ t2 . ibd
-- Table t0 is empty
```

Figure 9: A post-recovery inconsistency case in GreatSQL.

7.3 Comparison with State of the Art

To evaluate the effectiveness of our approach, we compared DDLUMOS with four state-of-the-art DBMS testing tools: SQUIRREL, SQLancer, SQLsmith, and TXCHECK, all of which are widely recognized in the industry. The evaluation focused on three key metrics: (i) the number of code branches covered in DDL operation modules, (ii) the average number of metadata conflict points generated per DDL statement within 48 hours, and (iii) the number of unique ADBs identified by each tool during the same 48-hour period. For a fair comparison, after each testing session, we collected all generated queries and reran them to unify branch measurements. Each experiments were run for 5 times.

Table 3: Number of related branches covered by each tool.

DBMS	SQLancer	SQLsmith	SQUIRREL	TXCHECK	DDLUMOS
MySQL	1083	401	954	571	3184
MariaDB	1,206	593	1103	452	2994
Percona	1,120	374	853	501	3201
PolarDB	1075	524	812	532	2814
GreatSQL	938	412	1063	447	2597
PostgreSQL	2041	953	1443	1202	3405
Total	7463	3257	6228	3705	18195
Variance	2304	8049	15293	9984	8852

Related Code Branches. Tables 3 show the average number of covered code branches in DDL operations modules by those DBMS testing tools in 48 hours within 5 runs. The results indicate that DDLUMOS outperformed other DBMS testing tools in triggering DDL behaviors. Specifically, DDLUMOS covered 10732, 14938, 11976, and 14490 more branches in Atomic DDL modules than SQUIRREL, SQLancer, SQLsmith, and, TXCHECK, respectively. The variance shows that different tools exhibit variations during testing.

Average Metadata Conflict Points. Table 4 shows that DDLUMOS generates more metadata conflict points for each DDL statement on average than other tools within 5 runs. Specifically, each DDL statement generated by DDLUMOS contains an average of 1.24 metadata conflict points, which is 0.93, 1.21, 0.17, and 0.76 more than the average metadata conflict points for each DDL statement generated by SQUIRREL, SQLancer, SQLsmith, and TXCHECK, respectively.

The improvement achieved by DDLUMOS is primarily due to its metadata conflict-guided DDL synthesis. This approach enables DDLUMOS to systematically generate test cases that specifically stress the atomicity of DDL operations by creating scenarios that involve metadata conflicts. These targeted test cases not only enhance coverage but also uncover deep logical flaws in the DBMS's handling of concurrent and interrupted DDL operations. In contrast, other tools lack specialized mechanisms for generating Atomic DDL scenarios. For example, SQLancer primarily generates random DDL operations without focusing on metadata conflicts, which limits its effectiveness in testing atomicity. Similarly, SQUIRREL and SQLsmith prioritize syntax correctness over conflict-driven test case design, resulting in lower coverage of Atomic DDL components in DBMSs.

Table 4: Number of average metadata conflict points per DDL.

DBMS	SQLancer	SQLsmith	SQUIRREL	TXCHECK	DDLUMOS
MySQL	0.33	0.03	0.15	0.43	1.28
MariaDB	0.27	0.03	0.17	0.46	1.31
Percona	0.29	0.02	0.16	0.44	1.26
PolarDB	0.29	0.04	0.17	0.5	1.25
GreatSQL	0.34	0.04	0.22	0.48	1.18
PostgreSQL	0.28	0.03	0.23	0.39	1.16
Average	0.3	0.03	0.18	0.45	1.24
Variances	0.00067	0.00047	0.0092	0.00127	0.00283

Triggered ADBs. Table 5 displays the average number of bugs detected by each tool in 48 hours within 5 runs. It shows that DDLUMOS outperformed the other tools, uncovering 27, 31, 32, and 26 more bugs compared to SQLancer, SQLsmith, SQUIRREL, and TXCHECK, respectively.

Table 5: Number of bugs detected by each tool in 48 hours.

DBMS	SQLancer	SQLsmith	SQUIRREL	TXCHECK	DDLUMOS
MySQL	2	1	2	3	8
MariaDB	1	0	0	1	9
Percona	2	1	1	2	5
PolarDB	2	2	0	1	4
GreatSQL	1	1	1	2	8
PostgreSQL	1	0	0	1	2
Total	9	5	4	10	36
Variances	2	0.8	0.4	2	6

To better understand the relationships between the bugs detected by the different tools, we analyzed their bug reports. Figure 10 illustrates the overlap between bugs detected by DDLUMOS and the other tools. DDLUMOS uniquely identified 36 bugs, demonstrating its ability to uncover a broad range of ADBs. Among these, 2 bugs and 3 bugs are overlapped with SQLancer and TXCHECK, respectively, while no overlap was observed with SQLsmith or SQUIRREL. The overlapping bugs detected by DDLUMOS, SQLancer, and TXCHECK were primarily related to simple schema modifications involving basic DDL operations, which both tools focus on. However, DDLUMOS significantly outperformed SQLancer in detecting bugs involving concurrent schema modifications and recovery scenarios, as SQLancer does not target these conditions.

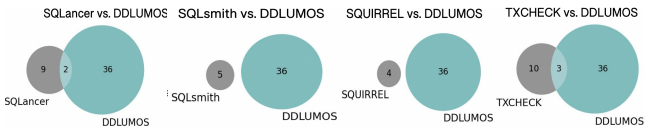


Figure 10: Venn diagram for bugs found by different tools.

The bugs detected by DDLUMOS and SQLsmith were entirely orthogonal. SQLsmith primarily focuses on generating complex `SELECT` statements to detect crash bugs, producing limited DDL statements that do not involve metadata modifications. In contrast, DDLUMOS targets metadata conflicts, incomplete rollbacks, and concurrency issues during DDL operations, making it uniquely suited for detecting ADBs. Similarly, the bugs detected by DDLUMOS and SQUIRREL were also orthogonal. SQUIRREL explores a broader range of DBMS behaviors with coverage guidance to detect memory safety bugs and crashes. The four bugs detected by SQUIRREL are unrelated to DDL statements.

7.4 Rediscovery of Surveyed ADBs

To further evaluate the effectiveness of DDLUMOS, we examined whether DDLUMOS could rediscover 207 known Atomic DDL bugs previously collected in our research. Our goal was to evaluate whether DDLUMOS could systematically rediscover these ADBs in realistic settings. Specifically, we applied DDLUMOS to the versions of PostgreSQL, MySQL, and MariaDB corresponding to each bug report in the dataset. For each DBMS version, DDLUMOS ran for one week to detect ADBs. Whenever a bug was detected, we compared its triggers and observed symptoms with those in the surveyed dataset. A bug was considered rediscovered if it manifested in the same way as originally reported, such as incorrect results, system crashes, or data loss.

Figure 11 presents the cumulative number of Atomic DDL bugs rediscovered by DDLUMOS in PostgreSQL, MySQL, and MariaDB in one week. Within the first two days of testing, DDLUMOS successfully rediscovered 86.5% of the bugs (179/207), demonstrating its efficiency in quickly exposing a significant portion of critical DDL-related issues. By extending the test period to one week, DDLUMOS increased the total number of rediscovered bugs to 196, representing approximately 94.7% of the 207 reported Atomic DDL bugs. This includes 34 out of 38 bugs in PostgreSQL, 48 out of 50 bugs in MySQL, and 114 out of 119 bugs in MariaDB.

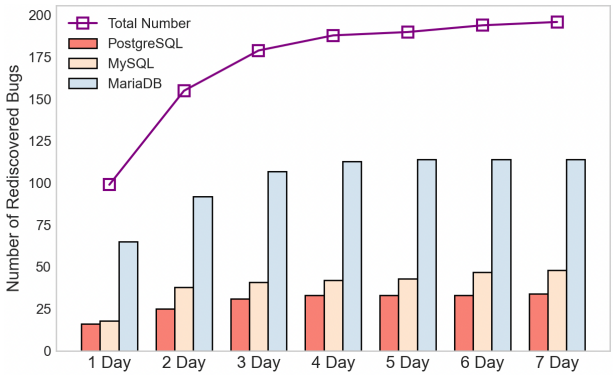


Figure 11: Number of rediscovered bugs in one week.

The high rediscovery rate primarily stems from two key design features in DDLUMOS. First, its *Metadata Conflict Guided DDL Synthesis* systematically constructs test scenarios that faithfully reflect real-world DBMS usage, such as concurrent schema modifications and transaction rollbacks. As a result, DDLUMOS rapidly surfaces problems that can lead to data corruption, server crashes, or anomalies during recovery. Second, the *Graph-Based Metadata Consistency Analysis* enhances DDLUMOS's effectiveness by identifying subtle inconsistencies in the database schema and system that may not be immediately obvious. Through a graph-based representation of schema states, DDLUMOS can efficiently explore diverse execution paths and detect bugs that would remain hidden under conventional testing.

8 Discussion

Missed Rediscoveries. Despite the bugs being rediscovered, there are still 11 bugs that did not resurface in the one-week experiments. A closer inspection revealed two main reasons for these missed rediscovered. First, 8 out of the 11 unrediscovered bugs relied on specialized plugins or configuration features. Since DDLUMOS does not modify system environments during testing (e.g., it does not install plugins or enable specific configurations), the required conditions for these bugs to occur were not present. These environment-dependent issues represent a small fraction (8 out of 207) of the total. Second, 3 bugs required specific execution paths that DDLUMOS's current generation strategy did not systematically explore within one week. When the testing period was extended to two weeks, DDLUMOS was able to rediscover

these three bugs, demonstrating its ability to detect rare or timing-sensitive defects when given sufficient time.

Extend Existing DBMS Testing Tools with DDLUMOS. While DDLUMOS primarily focuses on detecting atomic DDL bugs, its metadata conflict tracking and DDL synthesis patterns can also aid in discovering other types of DBMS bugs. First, DDLUMOS can be integrated into existing grammar-based DBMS testing frameworks to enhance their bug detection capabilities. For example, grammar-based tools typically focus on generating syntactically correct SQL queries, but they may not always consider how DDL operations interact with schema elements. By incorporating DDLUMOS, these existing frameworks can trigger scenarios that may expose rare or hard-to-find bugs in DBMS schema handling. Furthermore, DDLUMOS can be used in conjunction with other DBMS testing tools, such as those focused on transaction-level bugs or logical errors, to cover a broader range of potential issues.

9 Related Work

DBMS Testing. DBMS testing has evolved significantly over the years, with efforts primarily focusing on crash bug detection [12, 16, 39, 46], correctness verification [14, 17, 35, 37, 40, 45], and performance evaluation [15, 18].

Crash bug testing aims to identify scenarios where the DBMS may fail or crash under specific conditions. Tools like SQLsmith [39] generate a variety of SQL queries to stress test the DBMS, revealing potential crash points. SQUIRREL [46] and RATEL [42] introduce coverage-feedback into the query mutation to cover more branches. LEGO [16] enhances code coverage with sequence-oriented SQL mutation. LEGO [16] proposes sequence-oriented mutation to improve the code coverage by combining different SQL Type Sequences of the statements. Griffin [9] designs grammar-free mutation methods to detecting crash bugs in DBMS. Correctness verification is another crucial aspect of DBMS testing, focused on ensuring the accuracy of query execution. Techniques such as metamorphic testing [14, 17, 35–37] and differential testing [40, 45] have gained widespread adoption. For example, SQLancer uses oracles [35, 37, 37] to detect logic bugs by comparing the outcomes of different queries that should theoretically yield the same result. Additionally, RAGS [40] and Grand [45] utilize differential testing to uncover logic discrepancies by comparing the results of similar queries across different execution paths. Performance evaluation tests how efficiently a DBMS handles diverse workloads. Tools like APOLLO [15] conduct performance regression testing to identify degradation over time. Meanwhile, AMOEBA [18] assesses the performance consistency of semantically equivalent queries to uncover unexpected performance bottlenecks. Puppy [44] detects performance degradation bugs with limited-optimization plan construction.

While these approaches have significantly advanced DBMS testing, they primarily focus on query execution and general DBMS stability, often overlooking the specific challenges associated with Data Definition Language (DDL) operations. With the growing complexity of DDL operations, particularly in high-concurrency environments, there is an urgent need for testing frameworks that focus specifically on ensuring atomicity and consistency in these operations.

Atomicity Bug Detection. Atomicity bugs pose significant challenges to DBMS reliability. These bugs occur when a sequence of SQL operations, expected to be atomic, fails to execute completely, leaving the DBMS in an inconsistent state. Existing work related to atomicity has largely focused on transaction management [8, 13, 19–21, 31, 38]. For example, techniques like transactional memory testing [19, 21] and atomicity violation detection in multithreaded programs [20, 31, 38] can also be applied to detect atomicity issues. In the context of DDL operations, Atomic DDL bugs are often triggered by metadata conflicts, insufficient rollback mechanisms, and concurrency issues. Tools specifically designed to detect such Atomic DDL bugs are still scarce. While some research has explored the use of fuzzing techniques to generate DDL operations for testing purposes [8, 13], these efforts are still in their infancy and lack the sophistication needed to fully capture the complex interactions that can lead to atomicity violations in real-world DBMSs.

DDLUMOS addresses this gap by focusing specifically on detecting atomicity bugs in DDL operations. By generating DDL sequences that intentionally create metadata conflicts and simulating high-concurrency environments, DDLUMOS is able to uncover vulnerabilities that are typically missed by conventional testing approaches. This focus on DDL atomicity represents a significant advancement in DBMS testing, providing a new avenue to improve the reliability of schema modifications in modern databases.

10 Conclusion

In this paper, we have presented an in-depth analysis of 207 ADBs across three widely used industry DBMSs. Our investigation reveals that ADBs primarily manifest as incorrect results, data loss during recovery, and system crashes, which are mainly triggered by the metadata conflicts between DDL statements. To address these issues, we developed DDLUMOS, a testing tool that detects ADBs with metadata conflict-based DDL generation. By employing DDLUMOS across widely used DBMSs (e.g., MySQL and MariaDB), we discovered 73 previously unknown bugs, all of which have been confirmed.

Acknowledgements

We thank the shepherd and reviewers for their valuable comments. This research is partly sponsored by the National Key Research and Development Project (No. 2022YFB3104000), NSFC Program (No. 62302256, 92167101, 62021002, U2441238), and CCF-NSFOCUS Program (No.2023001).

References

[1] databases. https://en.wikipedia.org/wiki/Database. Accessed: June 3, 2025.

[2] Mariadb github. https://github.com/MariaDB/server. Accessed: June 3, 2025.

[3] Postgresql github. https://github.com/postgres/postgres. Accessed: June 3, 2025.

[4] BARTHOLOMEW, D. *MariaDB cookbook*. Packt Publishing Ltd, 2014.

[5] BLÅUDD, A. M. Percona website. https://www.percona.com/. Accessed: June 3, 2025.

[6] CORPORATION, M. Mariadb's jira. https://jira.mariadb.org, 2024. Accessed: June 3, 2025.

[7] CORPORATION, O. Mysql bug system. https://bugs.mysql.com/, 2024. Accessed: June 3, 2025.

[8] CUI, Z., DOU, W., DAI, Q., SONG, J., WANG, W., WEI, J., AND YE, D. Differentially testing database transactions for fun and profit. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering* (2022), pp. 1–12.

[9] FU, J., LIANG, J., WU, Z., WANG, M., AND JIANG, Y. Griffin: Grammar-free dbms fuzzing. In *Conference on Automated Software Engineering (ASE'22)* (2022).

[10] GREATDB. Greatsql github. https://github.com/GreatSQL/GreatSQL. Accessed: June 3, 2025.

[11] GROUP, T. P. G. D. Postgresql bug report mailing list. https://www.postgresql.org/list/pgsql-bugs/, 2024. Accessed: June 3, 2025.

[12] JIANG, Z.-M., BAI, J.-J., AND SU, Z. {DynSQL}: Stateful fuzzing for database management systems with complex and valid {SQL} query generation. In *32nd USENIX Security Symposium (USENIX Security 23)* (2023), pp. 4949–4965.

[13] JIANG, Z.-M., LIU, S., RIGGER, M., AND SU, Z. Detecting transactional bugs in database engines via Graph-Based oracle construction. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)* (Boston, MA, July 2023), USENIX Association, pp. 397–417.

[14] JIANG, Z.-M., AND SU, Z. Detecting logic bugs in database engines via equivalent expression transformation. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)* (Santa Clara, CA, July 2024), USENIX Association, pp. 821–835.

[15] JUNG, J., HU, H., ARULRAJ, J., KIM, T., AND KANG, W. APOLLO: Automatic Detection and Diagnosis of Performance Regressions in Database Systems (to appear). In *Proceedings of the 46th International Conference on Very Large Data Bases (VLDB)* (Tokyo, Japan, Aug. 2020).

[16] LIANG, J., CHEN, Y., WU, Z., FU, J., WANG, M., JIANG, Y., HUANG, X., CHEN, T., WANG, J., AND LI, J. Sequence-oriented dbms fuzzing. In *2023 IEEE International Conference on Data Engineering (ICDE)*, IEEE.

[17] LIANG, J., WU, Z., FU, J., WANG, M., SUN, C., AND JIANG, Y. Mozi: Discovering DBMS bugs via configuration-based equivalent transformation. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024* (2024), ACM, pp. 135:1–135:12.

[18] LIU, X., ZHOU, Q., ARULRAJ, J., AND ORSO, A. Automatic detection of performance bugs in database systems using equivalent queries. In *Proceedings of the 44th International Conference on Software Engineering* (2022), pp. 225–236.

[19] LOURENÇO, J., AND CUNHA, G. Testing patterns for software transactional memory engines. In *Proceedings of the 2007 ACM workshop on Parallel and distributed systems: testing and debugging* (2007), pp. 36–42.

[20] LU, S., TUCEK, J., QIN, F., AND ZHOU, Y. Avio: detecting atomicity violations via access interleaving invariants. *ACM SIGOPS Operating Systems Review 40*, 5 (2006), 37–48.

[21] MANOVIT, C., HANGAL, S., CHAFI, H., MCDONALD, A., KOZYRAKIS, C., AND OLUKOTUN, K. Testing implementations of transactional memory. In *Proceedings of the 15th international conference on Parallel architectures and compilation techniques* (2006), pp. 134–143.

[22] MARIADB. Aotmic ddl of mairadb. https://mariadb.com/kb/en/atomic-ddl/. Accessed: June 3, 2025.

[23] MARIADB. Aotmic ddl of mysql. https://dev.mysql.com/doc/refman/8.4/en/atomic-ddl.html. Accessed: June 3, 2025.

[24] MARIADB. Atomic ddl: .frm file is removed and orphan innodb tablespace is left behind upon crash recovery. https://jira.mariadb.org/browse/MDEV-25506. Accessed: June 3, 2025.

[25] MARIADB. Innodb: Failing assertion: !dict_index_is_online_ddl(index) upon alter table. https://jira.mariadb.org/browse/MDEV-13205. Accessed: June 3, 2025.

[26] Mariadb. https://mariadb.org/, 1 2024. Accessed: June 3, 2025.

[27] MOMJIAN, B. *PostgreSQL: introduction and concepts*, vol. 192. Addison-Wesley New York, 2001.

[28] MYSQL. Duplicate detection flaw for null values in uk during ddl rollback operations. https://bugs.mysql.com/bug.php?id=116502. Accessed: June 3, 2025.

[29] Mysql. https://www.mysql.com/, 1 2024. Accessed: June 3, 2025.

[30] ORACLE. Oracle security website. https://www.oracle.com/security-alerts/. Accessed: June 3, 2025.

[31] PARK, C.-S., AND SEN, K. Randomized active atomicity violation detection in concurrent programs. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering* (2008), pp. 135–145.

[32] Polardb. https://www.alibabacloud.com/product/polardb, 1 2024. Accessed: June 3, 2025.

[33] POSTGRESQL. Drop event trigger for ddl finishes successfully but trigger still executed on ddl. https://www.postgresql.org/message-id/18570-43bc1b85bebd2572%40postgresql.org. Accessed: June 3, 2025.

[34] Postgresql. https://www.postgresql.org/, 1 2024. Accessed: June 3, 2025.

[35] RIGGER, M., AND SU, Z. Detecting optimization bugs in database engines via non-optimizing reference engine construction. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (2020), pp. 1140–1152.

[36] RIGGER, M., AND SU, Z. Finding bugs in database systems via query partitioning. *Proceedings of the ACM on Programming Languages 4*, OOPSLA (2020), 1–30.

[37] RIGGER, M., AND SU, Z. Testing database engines via pivoted query synthesis. In *14th USENIX Symposium on Operating Systems Design and Implementation OSDI 20)* (2020), pp. 667–682.

[38] SAMAK, M., AND RAMANATHAN, M. K. Synthesizing tests for detecting atomicity violations. In *Proceedings of the 2015 10th Joint Meeting on foundations of software engineering* (2015), pp. 131–142.

[39] SELTENREICH, A., TANG, B., AND MULLENDER, S. Sqlsmith: a random sql query generator.

[40] SLUTZ, D. R. Massive stochastic testing of SQL. In *VLDB'98, Proceedings of 24rd International Conference on Very Large Data Bases, New York, USA*, Morgan Kaufmann, pp. 618–622.

[41] VAN DER LANS, R. F. *The SQL standard: a complete guide reference*. Prentice Hall International (UK) Ltd., 1989.

[42] WANG, M., WU, Z., XU, X., LIANG, J., ZHOU, C., ZHANG, H., AND JIANG, Y. Industry practice of coverage-guided enterprise-level dbms fuzzing. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)* (2021), IEEE, pp. 328–337.

[43] WIDENIUS, M., AXMARK, D., AND ARNO, K. *MySQL reference manual: documentation from the source*. " O'Reilly Media, Inc.", 2002.

[44] WU, Z., LIANG, J., FU, J., WANG, M., AND JIANG, Y. Puppy: Finding performance degradation bugs in dbmss via limited-optimization plan construction. In *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering* (2025), pp. 1–12.

[45] ZHENG, Y., DOU, W., WANG, Y., QIN, Z., TANG, L., GAO, Y., WANG, D., WANG, W., AND WEI, J. Finding bugs in gremlin-based graph database systems via randomized differential testing. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis* (2022), pp. 302–313.

[46] ZHONG, R., CHEN, Y., HU, H., ZHANG, H., LEE, W., AND WU, D. Squirrel: Testing database management systems with language validity and coverage feedback. In *The ACM Conference on Computer and Communications Security (CCS), 2020* (2020).

SpaceExit: Enabling Efficient Adaptive Computing in Space with Early Exits

Jiacheng Liu[1,2,*] Xiaozhi Zhu[1,*] Tongqiao Xu[1] Xiaofeng Hou[1,†] Chao Li[1,†]
[1]*Shanghai Jiao Tong University*
[2]*The Chinese University of Hong Kong*

Abstract

Advances in satellite technology and reduced launch costs have led to a proliferation of Earth observation (EO) satellites in low-Earth orbit (LEO). These satellites generate massive high-resolution imagery, creating a significant downlink bottleneck due to limited satellite-to-ground communication bandwidth. While orbit edge computing (OEC) can reduce data volume, existing static approaches fail to adapt to the varying complexity of satellite imagery, resulting in limited system performance and inefficient resource utilization.

We therefore propose *SpaceExit*, an integrated system for efficient adaptive computing on satellites. *SpaceExit* introduces three key components: (1) a geospatial-contextual adaptive detector that leverages both visual semantics and geospatial context to adjust processing complexity for each image, (2) a complexity-driven adaptive task scheduler that partitions images into tiles and allocates inference tasks across onboard devices based on content complexity and device capabilities, and (3) a satellite resource adaptive controller that ensures safe and efficient execution under changing conditions. Evaluations of diverse satellite settings and hardware platforms demonstrate that *SpaceExit* increases the performance by 5.2%-37.6% compared with the SoTA design.

1 Introduction

Recent advances in satellite technology and reduced launch costs have led to a significant increase in Earth observation (EO) satellites in low-Earth orbit (LEO) [37]. These advanced satellites, equipped with high-resolution cameras, can generate terabytes of imagery data per day, providing valuable insights for applications such as precision agriculture [4, 29], disaster response [33, 35], and environmental monitoring [28]. However, this massive data generation creates unprecedented challenges for satellite-based computing systems that differ fundamentally from terrestrial computing environments in three critical aspects.

First, *satellite computing occurs in an extremely constrained and hostile environment*. Satellites face severe thermal variations (e.g., -22°C to +77°C [34]) as they move between sunlight and Earth's shadow during their orbital periods. Such extreme conditions can significantly impact onboard chips and other equipment. As a result, effective satellite computing requires careful thermal management strategies that extend beyond conventional resource optimization. For instance, existing analysis of operational LEO satellites shows that thermal constraints alone can reduce computation performance by up to 10% [39].

Second, *satellite resources exhibit unique dynamic characteristics tied to orbital mechanics*. As micro-datacenters in space, satellites face network and power constraints that differ fundamentally from terrestrial systems. Satellite-to-ground communication bandwidth varies dramatically (e.g., from 0 to 220 Mbps) based on orbital position and ground station visibility [1]. Similarly, available power fluctuates with solar panel orientation and eclipse periods. Traditional "bent pipe" approaches that simply downlink raw imagery become impractical under these constraints [10].

Third, *satellite observation tasks present distinct workload patterns that challenge conventional computing paradigms*. Continuous Earth monitoring generates spatially and temporally correlated image sequences with varying complexity from simple ocean scenes to intricate urban landscapes [37]. These images often contain redundant information across overlapping observation swaths, yet may also capture rapidly evolving phenomena requiring immediate analysis.

While recent orbital edge computing (OEC) approaches [37] attempt to address these by performing onboard analysis, existing solutions rely largely on static processing pipelines that fail to adapt to the dynamic nature of both satellite operations and EO data [9–11]. They typically attempt to reduce computational overhead by selectively discarding imagery or employing varying models for different scenes [9, 10]. Despite these methods leveraging

*Equal contributions.
†Corresponding authors.

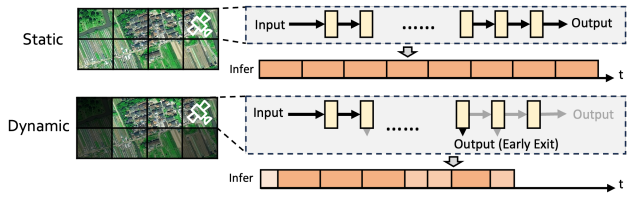


Figure 1: Comparison between the existing static OEC method and the proposed adaptive OEC method (*SpaceExit*). Unlike static approaches that process all imagery indiscriminately, *SpaceExit* adapts its processing to image complexity, performing early exiting for simple images and full processing for complex ones.

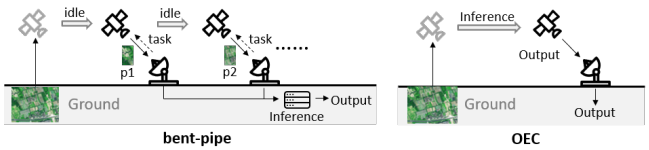


Figure 2: Comparison of classic bent-pipe with OEC.

prior knowledge about images to implement differentiated processing, they ultimately rely on static models. This inherent rigidity presents significant limitations in EO contexts, where scene complexity, illumination conditions, and orbital parameters undergo constant flux. Moreover, these systems cannot adaptively respond to runtime variations in both computational resources (like chips) and environmental factors which is a critical requirement for resource-constrained satellite platforms. As illustrated in Figure 1, an adaptive processing pipeline that *adaptively modulates computational intensity based on real-time scene complexity and resource* can achieve superior efficiency by enabling early termination for simpler inputs while maintaining high-fidelity processing for challenging scenarios.

We therefore propose *SpaceExit*, an integrated system that addresses all three satellite-specific characteristics through algorithm-system co-design. Unlike previous methods that merely simulate adaptive behavior through predefined rules, *SpaceExit* continuously adjusts its processing based on current input and system conditions, offering a truly flexible and efficient solution to the growing demands of EO missions.

At its core, *SpaceExit* features a geospatial-contextual adaptive detector that serves as the foundation for adaptive computation. This module incorporates a multi-scale feature extraction pipeline with an innovative exiting mechanism, adaptively adjusting processing complexity for each image region based on visual semantics and geospatial context. The system's architecture enables efficient handling of both fine-grained details and large-scale patterns while allowing early termination for easily recognizable images. To enhance adaptability, a geospatial context component integrates auxiliary information such as geographical location, time of day, and historical data to guide exit decisions and optimize resource allocation. For example, it can prioritize more complex processing in urban areas while enabling quicker exits in open-water regions, thus optimizing resource allocation based on contextual information.

Complementing this adaptive processing capability, *SpaceExit* features a complexity-driven adaptive task scheduler designed to optimize onboard computing resource utilization, which is crucial for long-term stable execution of in-orbit computing tasks [39]. It employs an adaptive tiling strategy, partitioning input images into regions of varying sizes based on their estimated complexity. Then it implements an adaptive allocation scheme that distributes these tiles across available onboard devices, considering factors such as tile complexity, device capabilities, and current workload. Additionally, to ensure robust operation in the challenging space environment, *SpaceExit* incorporates a satellite resource adaptive controller. This component continuously monitors system parameters including computing load, bandwidth utilization, temperature, and energy consumption. It adaptively adjusts scheduling and exiting strategies to maintain safe operating conditions while maximizing processing throughput.

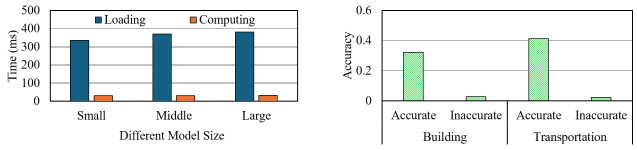
Our comprehensive experiments simulate various scenarios and hardware configurations with real-world datasets. The results demonstrate that *SpaceExit* consistently outperforms existing static detection pipelines, increasing the performance by 24.3% on average compared with the SoTA design.

The contributions of this work are multifaceted. To the best of our knowledge, *SpaceExit* represents the first integrated system for adaptive onboard earth observation that is optimized across both algorithm and system levels for satellite deployment. The novel geospatial-contextual adaptive detector pushes the boundaries of efficient inference in resource-constrained environments. The complexity-driven adaptive task scheduler offers a new paradigm for managing heterogeneous computing resources in space, while the satellite resource adaptive controller ensures reliable operation under the unique constraints of orbital platforms.

2 Background and Challenges

2.1 Static OEC Systems

To address the limitations of classic "bent-pipe" approaches, OEC is proposed to process data onboard satellites before transmission (Figure 2). Existing OEC systems mainly rely on static methods for data processing. TargetFuse [40] deploys a DNN model serving as a counter, while Kodan [9] provides several context-based models to yield higher accuracy in different scenes. These static approaches also adopt selective discarding techniques or data deduplication, achieving significant data reduction. However, these methods struggle to



(a) Model loading overhead of different models.

(b) Inaccurate priors significantly impact performance.

Figure 3: Limitation of static methods.

adapt to the diverse and dynamic nature of Earth observation data, often leading to suboptimal results when confronted with unexpected or complex scenes.

A major limitation of multi-model systems for satellite-based image processing is the substantial overhead associated with model loading and storage. Satellites have limited onboard storage capacity, constraining the number and complexity of models that can be deployed [3]. Our analysis, as shown in Figure 3a, reveals that the time spent loading different object detection models is often much longer than their computation time. This overhead significantly impacts system efficiency and responsiveness. Furthermore, the typical approach of applying simple models first and escalating to more complex ones if necessary can result in significant delays and computational waste. If a simple model proves inadequate for a given scene, the resources expended on its application and the time spent loading subsequent models are essentially lost. These limitations force a trade-off between model diversity and processing capability, highlighting the need for a more efficient and adaptive onboard processing approach.

Another critical drawback of static methods is their reliance on prior knowledge, which may be inaccurate or unavailable in the dynamic domain of Earth observation [7]. Pre-defined rules for scene classification or object detection may fail when confronted with unexpected phenomena or rapidly changing environments. Our analysis, illustrated in Figure 3b, demonstrates this vulnerability. We trained two models specifically designed for Building and Transportation detection. When the prior knowledge aligns with the input data, these models achieve high performance. However, when confronted with unexpected scenes or objects, their performance drops significantly. This inflexibility leads to suboptimal resource utilization across heterogeneous scenes, as the system cannot adaptively adjust its processing strategy based on the current input and operational context.

2.2 Challenges of Adaptive OEC

While implementing a truly adaptive method for Earth observation processing holds great potential, implementing such a system presents several significant challenges:

Challenge-1: Adapting multi-scale information and geospatial context for object detection in OEC. Dynamic object detectors have shown great promise in non-satellite scenarios, where they can adjust computation based on the complexity of the input data. However, applying these methods to satellite imagery introduces additional challenges. Satellite imagery requires the processing of multi-scale image information and incorporation of geospatial context, which is not typically considered in traditional methods. Furthermore, the constraints of the satellite environment, such as limited computational resources and the need for real-time processing, make dynamic methods in satellites more energy-sensitive compared to ground-based systems. Developing an adaptive approach that can adjust computation cost without sacrificing accuracy while leveraging both visual and geospatial information is a complex task requiring careful algorithm design.

Challenge-2: Scheduling dynamic task across heterogeneous onboard devices. Unlike static methods where processing requirements can be predicted in advance, adaptive approaches must continuously adjust resource allocation based on current input and system state. This is particularly challenging in the satellite environment where multiple heterogeneous computing devices with varying capabilities are often deployed. Designing a scheduling algorithm that can efficiently distribute workload across these devices while adapting to dynamic processing requirements is non-trivial.

Challenge-3: Runtime resource management under varying onboard constraints. The ever-changing space environment and varying mission requirements pose unique difficulties in runtime management. Satellites face severe constraints such as bandwidth, power, and thermal budget, which fluctuate based on orbital position and mission phase. Additionally, onboard resources like power and memory are limited and must be carefully managed. Prior work [39] has shown that these constraints can significantly impact the stablitity and performance of onboard processing systems. Ensuring safe and efficient execution under these dynamic conditions requires careful design of the resource management system.

3 *SpaceExit* System Design

The *SpaceExit* system architecture is designed to address the key challenges of onboard object detection on satellites holistically. As shown in Figure 4, *SpaceExit* takes the high-resolution satellite images as input and processes them with an efficient adaptive object detector on heterogeneous onboard computing devices.

Specifically, *SpaceExit* contains three main modules that synergistically optimize the detection pipeline across various dimensions of algorithm-system co-design:

- **Geospatial-Contextual Adaptive Detector (GCAD, Section 3.1)**: This module forms the core of *SpaceExit*'s adaptive processing capability to address *Challenge-1*. It employs a multi-scale feature backbone with a novel exiting mechanism that leverages both visual semantics and geospatial context. The GCAD can adaptively adjust

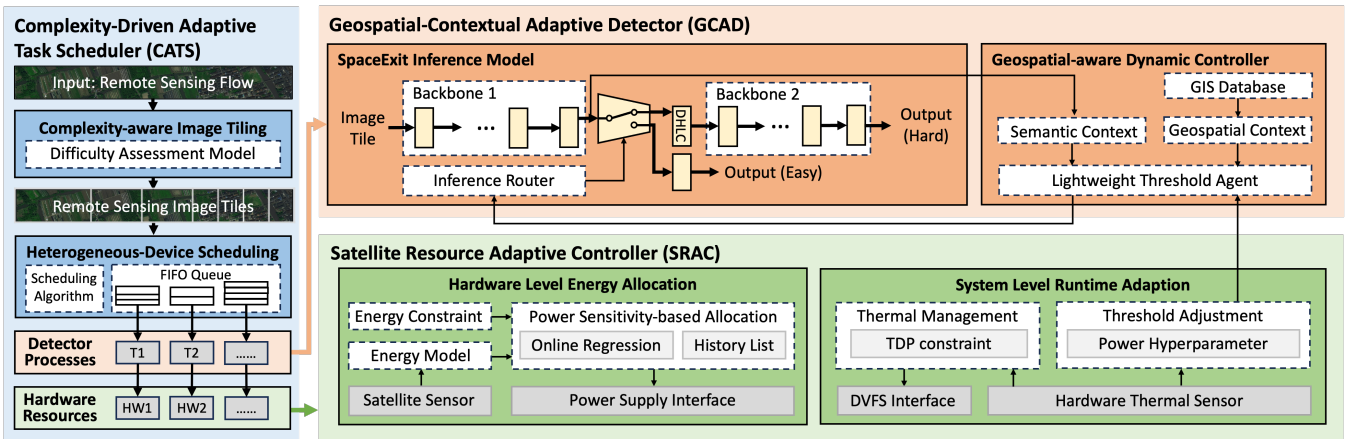


Figure 4: Overview of the *SpaceExit*.

the processing complexity for each tile, allowing for efficient early exits on simpler areas while dedicating more resources to complex regions. This approach enables *SpaceExit* to balance detection accuracy and computational efficiency across diverse satellite imagery.

- **Complexity-Driven Adaptive Task Scheduler (CATS, Section 3.2)**: The CATS optimizes the utilization of onboard computing resources across multiple devices to address *Challenge-2*. It implements a complexity-aware scheduling algorithm that partitions input images into tiles based on their estimated complexity. These tiles are then allocated across available onboard devices, considering factors such as tile complexity, device capabilities, and current workload. This adaptive approach ensures efficient load balancing and maximizes overall system throughput.

- **Satellite Resource Adaptive Controller (SRAC, Section 3.3)**: The SRAC is responsible for maintaining safe and efficient system operation under the dynamic constraints of the space environment to address *Challenge-3*. It continuously monitors system parameters including computing load, temperature, and energy consumption. Based on these inputs, the SRAC adaptively adjusts system configurations and exiting strategies to optimize resource utilization while ensuring the system operates within safe limits.

These three modules work together to enable *SpaceExit*'s efficient and adaptive approach to onboard object detection. The GCAD provides content-aware processing, the CATS ensures optimal resource utilization across devices, and the SRAC maintains system stability and efficiency in changing environmental conditions.

3.1 Geospatial-Contextual Adaptive Detector

The GCAD forms the core of *SpaceExit*'s adaptive processing capability, addressing the challenge of efficiently processing satellite imagery with varying complexity. It consists of two key components: a multi-exit network architecture and a geospatial-aware threshold adjustment mechanism.

3.1.1 Multi-exit network architecture

GCAD employs a multi-scale feature backbone with multiple exit points, allowing for adaptive adjustment of processing complexity based on image content. This backbone is designed to capture features at various scales, which is essential for detecting both large-scale structures and small details in satellite images. Specifically, a L layer feature extraction process can be defined as,

$$f_{\mathbf{x}}^{L} = \mathcal{F}_L(\cdots \mathcal{F}_2(\mathcal{F}_1(\mathbf{x}))), \tag{1}$$

where $\mathbf{x}$ is the input image and $\mathcal{F}_i$ represents the feature extractor at the i-th scale. Each $\mathcal{F}_i$ is designed to extract features at a scale lower than $\mathcal{F}_{i+1}$ and higher than $\mathcal{F}_{i-1}$, allowing for a comprehensive multi-scale representation of the image. The overall detector design builds upon the concept of progressive feature refinement [23], it consists of multiple connected networks: a primary feature extractor and multiple enhancement networks. The primary extractor processes the input image to generate an initial set of multi-scale features, while the enhancement network further refines these features to capture more complex patterns and relationships.

To implement adaptive computing for different images, the central question is how to estimate the uncertainty score of the model's prediction on different images. Existing methods may use confidence-based estimation to measure uncertainty. However, the neural network is known to be over-confident [2]. Thus, we introduce an additional router that takes multiple features as input and outputs the estimated uncertainty of

the model's prediction on this image. Simialr to the squeeze-and-excitation (SE) module [20], we first pool the multi-scale features F independently and concatenate them all as,

$$\widetilde{F} = \|_{i=1}^{L} \mathcal{P}(f^i), \tag{2}$$

where $\mathcal{P}$ denotes global average pooling, $\|$ denotes channel-wise concatenation, and f^i represents the features from the i-th scale of the feature. This operation compresses the multi-scale features into a compact representation $\widetilde{F}$, which is then used by the router to make its decision. Specifically, the uncertainty score can be calculated as,

$$u = \sigma(W_2(\delta(W_1\widetilde{F} + b_1)) + b_2), \tag{3}$$

where δ, σ denote the ReLU and Sigmoid activation functions respectively, and W_1, W_2, b_1, b_2 are learnable parameters.

After each processing stage, the router will be activated to judge whether the model should processed to the next block or go to the head to get the prediction and exit. Specifically, for images deemed "easy" (i.e., those with an uncertainty score below a certain threshold), the system proceeds directly to the detection head. Conversely, for "hard" images, the system activates the enhancement networks for more sophisticated processing until it gets enough certainty score or reaches the end of the detector.

This adaptive routing allows GCAD to allocate computational resources efficiently, dedicating more processing power to complex images while quickly handling simpler ones. Such an approach is particularly valuable in satellite imaging applications, where images can vary greatly in complexity and where processing resources may be limited.

Training a adaptive detector like GCAD presents unique challenges. We employ an adaptive offset-based training method [25] to balance the performance of the different processing paths. This method introduces adaptive offsets to the loss function, encouraging the network to utilize both the simple and complex processing paths effectively.

3.1.2 Geospatial-aware adaptive inference

With the multi-scale features and early exit branches, the next question is how to adaptively decide which scale to extract and use for each image tile. Considering both the semantic and geospatial context in satellite computing is essential. However, interweaving complex multi-exit network training with geographic information and resource margin will greatly increase the difficulty and cost of model training. To this end, we propose a two-stage approach, which introduces a lightweight auxiliary model based on the adaptive model to fuse geospatial information.

Firstly, we adjust the trained adaptive neural network based on the difficulty score which is irrelevant to the geospatial information. This ensures that we can use vast amounts of satellite images without geospatial information to train our model. Specifically, consider a validation set $\mathcal{S}_{val}$, the threshold can be estimated as,

$$\tau_{val} = \mathcal{P}(\mathcal{S}_{val}, k), \tag{4}$$

where $\mathcal{P}(\cdot, k)$ means the k-th quantile of $\mathcal{S}_{val}$.

Then, we utilize a small dataset with geospatial information to train a lightweight agent to adaptively adjust the threshold. Specifically, the agent takes two inputs: (1) The semantic features r from each exit scale, which encodes the semantic information. (2) The geospatial embedding g, which encodes the geographic context like terrain type, land cover, POI density, etc. The geospatial priors are pre-computed offline and stored in a GIS database. At runtime, they are efficiently queried based on the region coordinates and resolutions. The priors are used as soft guidance and can be overridden by the visual patterns if there is a discrepancy. This adaptive fusion of visual and geospatial information improves the detection accuracy and efficiency.

Based on the above strategy, *SpaceExit* can adaptively select the most appropriate scale and exit point for each image tile, optimizing both accuracy and computational efficiency. By integrating geospatial information and image features, *SpaceExit* ensures that the inference process is not only context-aware but also adaptive to varying operational conditions. This results in improved performance in terms of both speed and accuracy, making *SpaceExit* highly suitable for real-time on-board satellite image analysis.

3.2 Complexity-Driven Adaptive Scheduler

CATS optimizes the utilization of onboard computing resources across multiple devices. It implements a complexity-aware scheduling algorithm that partitions input images into tiles based on their estimated complexity and allocates them across available onboard devices.

3.2.1 Complexity-aware image titling

CATS first uses a difficulty assessment model that estimates the processing complexity d_t of each image. This model can leverage simple heuristics (e.g., color variation) or train a simple low-complexity model.

Based on the difficulty assessment, CATS employs an adaptive tiling strategy to partition input images. This approach is motivated by the observation that satellite images often contain regions of varying complexity, and uniform tiling can lead to inefficient resource utilization.

Assume there are N onboard devices, where each device i has a processing rate v_i (in pixels/s) and memory capacity m_i. Given a raw input image I with dimensions $W \times H$, CATS partitions it into a grid of N_r tiles $\{r_1, \ldots, r_{N_r}\}$, where each tile has dimensions $w_i \times h_i$.

The tile size is chosen to satisfy two key constraints:

1. Memory constraint: $wh < \min_i m_i$, ensuring that a tile can fully fit into any device's memory.

2. Accuracy-complexity balance: The tile size should be large enough to guarantee fast processing, but small enough to capture meaningful context for object detection [40].

The actual tiling process is adaptive, with tile sizes varying based on the difficulty scores. More complex regions (higher d_t) are divided into smaller tiles for more focused processing, while simpler regions use larger tiles to reduce overhead:

$$w_t = h_t = \frac{w_{base}}{\sqrt{d_t}}, \tag{5}$$

where w_{base} is a base tile width chosen to satisfy the above constraints.

3.2.2 Heterogeneous-Device Scheduling

The core of CATS is its multi-device scheduling algorithm, which allocates tiles to available onboard devices. This allocation considers three primary factors as: tile complexity (from the difficulty assessment), device capabilities (processing rate, memory capacity) and Current device workload.

CATS maintains a FIFO queue Q_i for each device i to manage pending image tiles. The scheduling process for each tile r involves assigning it to a device queue:

$$Q_{\sigma(i)} = Q_{\sigma(i)} \cup \{r\}, \tag{6}$$

where the device selection function $\sigma(i)$ is defined as:

$$\sigma(i) = \arg\min_j \left\{ \frac{n_j}{v_j} \middle| j \in [1,N], |Q_j| < K_j \right\}. \tag{7}$$

Here, n_j is the current unprocessed workload on device j, v_j is its processing rate, and K_j is the maximum queue size for device j. This policy aims to equalize processing time across devices while avoiding queue overflow.

Unlike static task where workload depends solely on the number of tasks, our system must account for both quantity and complexity. Thus, we modify the workload measure n_j to incorporate the difficulty scores:

$$n_j = \sum_{r \in Q_j} d_r \cdot |r|, \tag{8}$$

where $|r|$ is the size of tile r in pixels.

CATS updates these workload counters n_j adaptively as new tiles are enqueued or processed tiles are dequeued. This process repeats for each raw image until all tiles are distributed, providing global load balancing across the heterogeneous onboard devices.

Furthermore, CATS implements a priority system within each device queue. Each tile r is assigned a priority score based on its deadline urgency and complexity:

$$s(r) = \eta \cdot \frac{1}{d(r) - t} + \lambda \cdot d_r, \tag{9}$$

where $d(r)$ is the tile's deadline, t is the current time, d_r is the tile's difficulty score, and η, λ are weighting factors that can be tuned based on mission requirements. This dual consideration of deadline urgency and processing complexity addresses the unique requirements of OEC tasks. The urgency factor ensures time-critical observations aren't missed, which is essential for applications like disaster monitoring. The complexity factor ensures that challenging tasks start processing earlier, reducing the risk of missing deadlines.

Tiles are processed in descending order of these scores: $r^* = \arg\max_r s(r)$. After each tile is processed, CATS updates the resource usage and checks if it exceeds the current budget allocated by the SRAC module. If so, it pauses scheduling until more resources become available in the next cycle.

This sophisticated, multi-factor scheduling approach ensures efficient utilization of limited onboard resources, adapting to both the varying complexity of input data and the heterogeneous capabilities of onboard devices.

3.3 Satellite Resource Adaptive Controller

The SRAC module is responsible for maintaining safe and efficient system operation under the dynamic constraints of the space environment. It continuously monitors system parameters including computing load, temperature, and energy consumption, and adaptively adjusts system configurations to optimize resource utilization while ensuring the system operates within safe limits.

3.3.1 Energy Allocation

A primary function of SRAC is the adaptive allocation of energy resources among onboard computing devices. This is particularly challenging in the satellite environment due to the highly variable nature of solar power availability and the critical importance of maintaining overall system stability.

Let $I(t)$ denote the total available power at time t, which varies based on factors such as solar panel orientation, eclipse periods, and battery state of charge. SRAC must allocate this power among N heterogeneous computing devices. Let $P_i(t)$ denote the power budget for device i at time t. The allocation must satisfy:

$$\sum_{i=1}^{N} P_i(t) \leq I(t). \tag{10}$$

SRAC employs an adaptive allocation policy $\pi : I(t) \rightarrow \{P_i(t)\}_{i=1}^N$ designed to maximize overall system throughput. This policy operates in two stages:

1. Reserve a base power P_{base} for critical satellite systems (e.g., attitude control, communication).

2. Allocate remaining power $I(t) - P_{base}$ among computing devices based on their performance sensitivity to power.

Formally, the allocation for each device is computed as:

$$P_i(t) = P_{base,i} + (I(t) - P_{base}) \cdot \frac{\partial X_i / \partial P_i}{\sum_{j=1}^{N} \partial X_j / \partial P_j}, \quad (11)$$

where X_i is the measured throughput of device i, and $\partial X_i / \partial P_i$ represents its performance sensitivity to power. This approach allocates more power to devices that can translate it into higher throughput gains.

The performance-power relationship $\partial X_i / \partial P_i$ is learned and continuously updated during system operation using online regression techniques. This allows the system to adapt to changes in device characteristics over time, which is crucial for long-duration space missions.

SRAC updates this allocation whenever there is a significant change in available power ($\Delta I > I_{th}$) or periodically (e.g., every T_{update} seconds) to account for gradual changes in system state.

3.3.2 DVFS and Threshold Adjustment

Given the power budget $P_i(t)$ for each device, SRAC employs fine-grained control to optimize performance within these constraints. This is achieved through a combination of Dynamic Voltage and Frequency Scaling (DVFS) and adaptive adjustment of processing thresholds.

Specifically, SRAC adjusts frequencies f_j based on measured utilization u_j:

$$f_j = f_{min} + (f_{max} - f_{min}) \cdot \frac{u_j - u_{min}}{u_{max} - u_{min}}, \quad (12)$$

subject to the Thermal Design Power (TDP) constraint [13]:

$$\sum_j A_j C_j V_j^2 f_j \leq P_i, \quad (13)$$

where A_j is the switching activity level, C_j is the capacitance of the circuit, and V_j is the voltage of core j. This approach allows higher frequencies (and thus higher performance) when utilization is high, while saving power during periods of low utilization.

Thermal management is a critical aspect of SRAC. It continuously monitors chip temperatures T_i and proactively adjusts power states when temperatures approach the limit T_{max}. SRAC employs a learned thermal-power model:

$$T_i(t+1) - T_i(t) = \alpha[P_i(t) - \beta(T_i(t)^4 - T_{amb}^4)], \quad (14)$$

where α and β are thermal parameters and T_{amb} is the ambient temperature. Using this model, SRAC estimates the power adjustment ΔP needed to maintain safe temperatures:

$$\Delta P = \frac{1}{\alpha}[T_{max} - T_i(t)] - [P_i(t) - \beta(T_i^4(t) - T_{amb}^4)]. \quad (15)$$

This allows for smooth throttling of DVFS settings to maintain safe operation without abrupt shutdowns.

Finally, SRAC adaptively adjusts the processing thresholds used in the GCAD module based on current resource availability. The exit threshold τ in GCAD is modulated as:

$$\tau = \tau_{base} \cdot \frac{\gamma P_{current}}{P_{nominal}}, \quad (16)$$

where $P_{current}$ is the current power availability, $P_{nominal}$ is the nominal power level and γ is the hyperparameter modeling the influence of threshold in power usage. This allows the system to become more conservative (exiting earlier) when resources are constrained, and more thorough when resources are plentiful.

3.3.3 Bandwidth Allocation

SRAC implements a priority-based bandwidth allocation strategy to maximize the value of limited satellite-to-ground communication opportunities. The allocation strategy operates on two timescales: inter-pass accumulation and intra-pass transmission. During inter-pass accumulation, SRAC accumulates data between ground station passes into priority queues. The intra-pass transmission phase occurs during a ground station pass, where SRAC allocates bandwidth based on the priority.

SRAC maintains three priority queues: Q_1 for onboard calculation results (highest priority), Q_2 for routine telemetry and housekeeping data (second highest priority), and Q_3 for uncertainty-scored image data. This ordering ensures that critical operational data and high-value scientific results are always transmitted before any additional image data.

During a ground station pass, SRAC follows a strict priority-based transmission protocol. The system first transmits all data from Q_1, ensuring that all onboard calculation results are downlinked. Once Q_1 is empty, it moves on to Q_2, transmitting all routine telemetry and housekeeping data. Only after both Q_1 and Q_2 are fully transmitted does the system begin to downlink data from Q_3, which contains the uncertainty-scored image data.

This priority-based allocation strategy ensures that SpaceExit always transmits critical operational data and high-value scientific results first. By using a simple, deterministic protocol for the highest-priority queues and a lightweight heuristic for managing lower-priority data, SRAC achieves robust and efficient bandwidth utilization while keeping computational overhead low.

4 Experimental Setup

4.1 Computational Hardware

We design and implement a heterogeneous computing testbed that faithfully emulates the resource-constrained environment of 3U CubeSat onboard computers. Our experimental platform incorporates two primary computing units that represent typical space-grade hardware configurations, carefully selected to balance performance and power constraints.

Firstly, we employ the NVIDIA Jetson Nano, which is equipped with a Quad-core ARM Cortex-A57 MPCore processor clocked at 1.43 GHz, paired with a 128-core Maxwell GPU that delivers 472 GFLOPS of compute performance. The system is supported by 4 GB of LPDDR4x memory, offering 25.6 GB/s bandwidth. Operating in a 10W TDP configuration, the Jetson Nano supports both FP16 and FP32 precision computations, making it suitable for efficient neural network inference within a compact 70 mm × 45 mm form factor [27]. Additionally, we utilize the NVIDIA Jetson Xavier NX, which offers higher performance in a similar size to the Jetson Nano. It features a 6-core NVIDIA Carmel ARMv8.2 64-bit CPU, a 384-core Volta GPU with 48 Tensor Cores, and 8 GB of LPDDR4x memory with 59.7 GB/s bandwidth. It delivers up to 21 TOPS of AI performance and supports TDP configurations of 10W, 15W and 20W, making it ideal for demanding AI applications in space-constrained environments.

To emulate a 3U CubeSat onboard computer with heterogeneous payloads, we setup an NVIDIA Jetson Nano with a 10W power mode connected to NVIDIA Jetson Xavier NX with 20W power mode. This configuration provides substantial AI computing power within the 4kg mass and 30W power budgets of a typical 3U bus [11], making it faithfully model the real onboard setup.

4.2 Satellite dataset

Following previous research [40], we evaluate *SpaceExit* using the DOTA (Dataset of Object deTection in Aerial images) dataset [38], a comprehensive satellite imagery dataset designed specifically for object detection tasks. The dataset contains 403k annotated instances across 15 object categories, including various vehicles, ships and infrastructures, with objects captured at arbitrary orientations to reflect real-world conditions. We preprocess to resemble the conditions encountered in EO applications closely.

4.3 *SpaceExit* Implementation

In *SpaceExit*, the adaptive router takes the terrain-type embedding (one-hot vector) and four visual embeddings as input. The selected scale then passes through a feature fusion module to combine the coarse semantics and fine details. The terrain type is predicted by a pre-trained MobileNet classifier fine-tuned on the DOTA images with ground-truth land cover labels. We use 2 categories: land and ocean, which are visually distinctive. This router requires only about 130KB of memory (approximately 2.4% of the full model's memory footprint), which is negligible.

We construct our geospatial embeddings by concatenating terrain type, land cover, and pre-processed POI data. Each geospatial embedding requires at most 32B of storage, resulting in a total database size under 100MB to cover the entire Earth. For single-orbit operations, this requirement decreases significantly to less than 100 KB. Given the inherent spatial locality of satellite imagery, our system retrieves 50 geospatial embeddings simultaneously. This efficient batching strategy means database access is required only once per minute, rendering the access time negligible in our overall processing pipeline. As for the complexity estimation model, we experimented with both a simple two-layer CNN and a color variation-based method. While both performed well, the latter proved significantly more efficient, leading us to adopt it as our default model.

The model is trained end-to-end with Adam optimizer at 1e-4 learning rate for 100 epochs. We use 0.1 weight decay and a cosine learning rate schedule with 5 epochs of warmup. The input size is 2048×2048 pixels, randomly cropped from the full images. The batch size is 8 per GPU (32 total) with synchronized BN. For augmentation, we apply HSV jittering, flipping, translation, and multi-scale training.

4.4 Baselines

We evaluate *SpaceExit* against four satellite image processing systems that represent the state-of-the-art system in OEC:

- BentPipe [11] implements the traditional ground-centric approach, where satellites downlink raw imagery to ground stations for processing. This system maximizes processing capability but is severely limited by communication bandwidth constraints.
- SpaceOnly [11] performs all object detection computations onboard using deployed neural networks, transmitting only the detection results to ground stations. While this minimizes bandwidth usage, it faces significant onboard computational resource constraints.
- Kodan [9] leverages specialized machine learning models selected based on prior knowledge about observation targets. This approach improves efficiency through targeted processing but is heavily dependent on the accuracy of prior information.
- TargetFuse [40] employs a combination of adaptive tiling, object clustering, and bandwidth-aware transmission scheduling to minimize detection errors under resource constraints. This hybrid approach balances onboard processing with efficient data transmission.

These baselines, ranging from fully ground-based to completely onboard processing, provide comprehensive reference points for evaluating *SpaceExit* 's performance.

4.5 Metrics

For detection performance, we adopt the standard MS COCO evaluation protocol, reporting mean Average Precision (mAP) at different Intersections over Union (IoU) thresholds. Specifically, we report the standard MS COCO metrics [24], including AP@IoU=0.5 and AP@IoU=0.5:0.95 for each category.

For efficiency, we measure the end-to-end processing time and total charge on the satellite system. This includes the time taken for taking photos, inference, and the communication of information from the satellite to the ground station. We use the total amount of information transmitted from the satellite to the ground station as the benchmark for evaluating the system's processing speed.

To evaluate the overall performance of our system, we use goodput which combines both accuracy and throughput. The goodput of a system is defined as the correct information produced per unit time. Goodput takes into account not only the amount of data processed but also the quality and relevance of the processed data, ensuring that our system meets the stringent requirements of satellite-based Earth observation.

5 Results Analysis

5.1 *SpaceExit* Consistently Outperforms SoTA OEC Systems

To compare the overall performance of *SpaceExit* with the existing systems, we systematically evaluated the system using three distinct hardware configurations: HW Set-1 (heterogeneous deployment with Jetson Xavier NX and Jetson Nano), HW Set-2 (homogeneous deployment with two Jetson Nanos), and HW Set-3 (high-performance setup with two Xavier NXs). Each configuration was tested under three network bandwidth conditions: 100Mbps, 50Mbps, and 10Mbps, providing insights into system behavior under varying resource constraints.

The results in Figure 5 reveal that *SpaceExit* consistently outperforms baseline approaches across all scenarios, with performance improvements varying based on hardware capabilities and network conditions. In the heterogeneous setup (HW Set-1), *SpaceExit* achieves remarkable improvements of 27.1%, 29.5%, and 29.4% for 100Mbps, 50Mbps, and 10Mbps bandwidths respectively. This demonstrates the system's ability to effectively leverage diverse computational resources. The performance gains are more modest in HW Set-2, ranging from 5.2% to 8.4%, indicating that computational constraints become a limiting factor when using lower-powered devices. However, HW Set-3 showcases *SpaceExit*'s full potential, with substantial improvements of 36.9-37.6%, highlighting the system's capability to efficiently utilize powerful computation resources.

The impact of network bandwidth on system performance presents interesting insights, particularly when examining BentPipe's behavior. Under high bandwidth conditions (100Mbps), BentPipe maintains competitive performance relative to other baselines, though still significantly below *SpaceExit*. However, its performance deteriorates dramatically under constrained bandwidth (10Mbps), achieving only approximately 5% normalized goodput. This stark contrast underscores the importance of bandwidth-aware design in distributed inference systems. *SpaceExit*, in contrast, maintains robust performance across all bandwidth conditions, demonstrating its resilience to network constraints.

Among the baseline approaches, Kodan emerges as the strongest competitor, particularly in computationally constrained scenarios like HW Set-2, where its lightweight detectors provide a strategic advantage. TargetFuse and SpaceOnly exhibit similar performance trajectories but consistently underperform compared to *SpaceExit*. The performance gap between *SpaceExit* and these baselines becomes particularly pronounced in scenarios with powerful hardware and adequate bandwidth, suggesting effective resource utilization by our proposed system.

To further analyze these methods, we measure the throughput as shown in Figure 6. Here, the Kodan outperforms other systems in most cases with the help of specialized light detectors. However, this may result in a significant performance drop when the prior knowledge is not accurate (the goodput of Kodan is lower as shown in Figure 5). What's more, as the computational capabilities of the devices increase, Kodan's advantage in throughput gradually diminishes. When using higher-performance hardware, specifically in the most powerful HW Set-3, Kodan is eventually surpassed by *SpaceExit*.

These results collectively demonstrate *SpaceExit*'s superior performance across diverse deployment scenarios. The system's ability to maintain significant improvements under various hardware configurations and network conditions validates its design principles and resource management strategies.

5.2 GCAD Enables Adaptive Computing

We evaluate the effectiveness of GCAD's adaptive processing capability by comparing its performance to that of static detectors on the DOTA test set. Our implementation is based on the YOLOv7 framework, where the static detector is configured with a fixed computational load of 3.4 GFLOPS, while GCAD adapts its computational load between 2.0 and 3.5 GFLOPS, depending on the complexity of the input image.

Figure 7 illustrates an example of how GCAD adapts its processing strategy based on image complexity. In Figure 7(a), we show a relatively simple scene containing well-separated ships on a uniform water background. For such images, GCAD successfully achieves early exit after the first

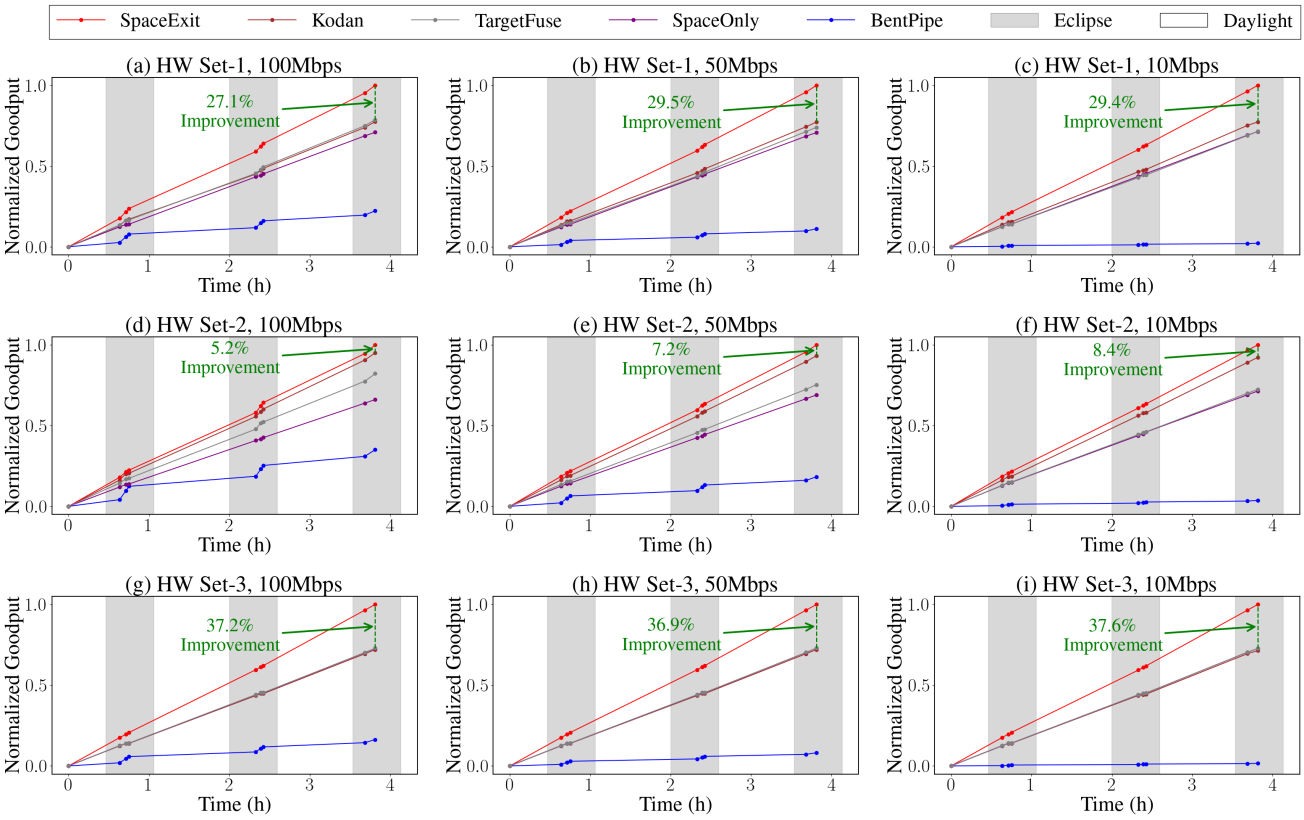


Figure 5: Performance comparison of *SpaceExit* against baselines across varied hardware configurations and network bandwidths.

backbone, maintaining detection accuracy while reducing computational cost by approximately 40%. In contrast, Figure 7(b) presents a complex urban scene with densely packed buildings and various objects at multiple scales. Here, GCAD correctly identifies the need for full processing through both backbones to maintain detection accuracy. This adaptive behavior demonstrates GCAD's ability to intelligently allocate computational resources based on scene complexity.

The quantitative results in Table 1 validate that our adaptive approach maintains or improves detection accuracy while reducing the average computational cost. The adaptive detector achieves an overall mAP@0.5 of 0.64 and mAP@0.5:0.95 of 0.404, outperforming the static detector's scores of 0.635 and 0.394, respectively. Performance improvements are particularly notable for challenging object categories - for instance, Bridge detection improves from 0.159 to 0.192 AP@0.5:0.95, and Road Aircraft (RA) detection shows a significant gain from 0.595 to 0.607 AP@0.5.

The class-wise analysis reveals that GCAD maintains robust performance across object categories with varying complexity. For simpler objects like planes and ships, which often appear against uniform backgrounds, the adaptive detector achieves comparable accuracy to the static detector while frequently enabling early exits. For complex categories like small vehicles (SV) and Harbor, where more detailed analysis is often necessary, GCAD appropriately opts for full processing to maintain accuracy. This balanced approach demonstrates that our adaptive detection strategy successfully adapts to both image-level and category-level complexity variations.

To further illustrate the performance of our adaptive model, we present a comparison between the static and adaptive models in terms of mean mAP across all categories, as well as mAP for selected object categories. Additionally, we show how the mAP of the adaptive model varies with computational load (measured in GFLOPS). Figure 8 illustrates the comparative performance of adaptive and static models. The plot depicts the mean mAP for adaptive models across all categories, as well as the mAP for specific categories such as Harbor, Large Vehicle (LV), and Roundabout (RA). The static models' mAP values are indicated by the triangular markers. This visualization underscores the adaptive model's ability to maintain high accuracy while adapting computational load based on image complexity.

5.3 CATS Improves Detection Throughput

To evaluate the effectiveness of CATS in improving system throughput, we conduct experiments comparing system per-

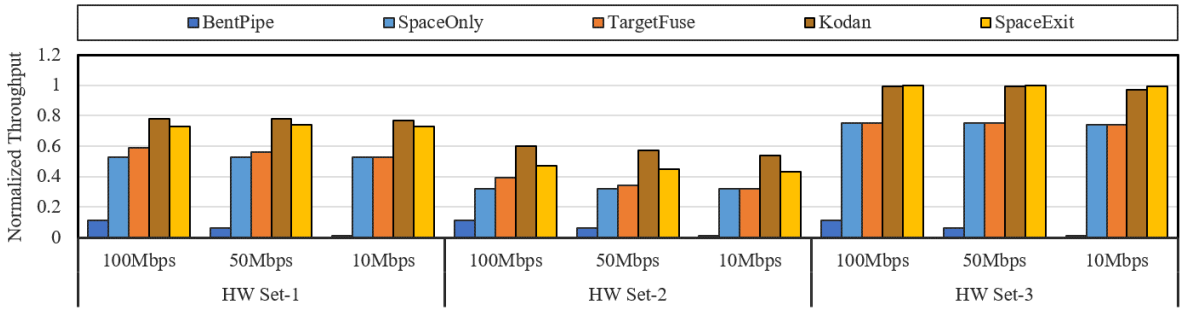
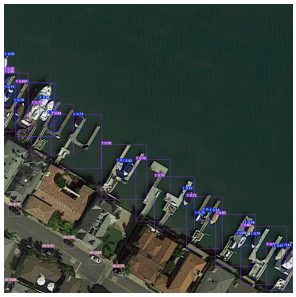


Figure 6: Normalized transmitted tile numbers of existing systems.



(a) Simple image that only requires partial execution.

(b) Complex image that requires full execution.

Figure 7: Examples of images used to illustrate the effectiveness of the GCAD's early exit decision.

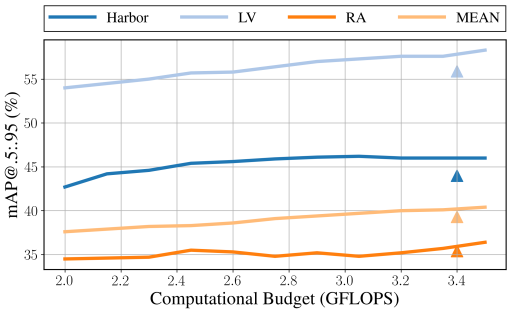


Figure 8: Comparison between GCAD (lines) and static detectors (triangles) on the DOTA test set across different computational budgets.

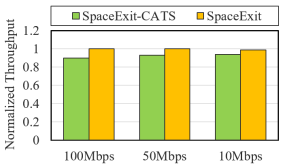


Figure 9: CATS improves the processing throughput.

Figure 10: SRAC improves the processing throughput.

formance with and without our adaptive task scheduler. We test it with HW Set-1 under varying communication bandwidths to assess scheduler performance across diverse operational scenarios.

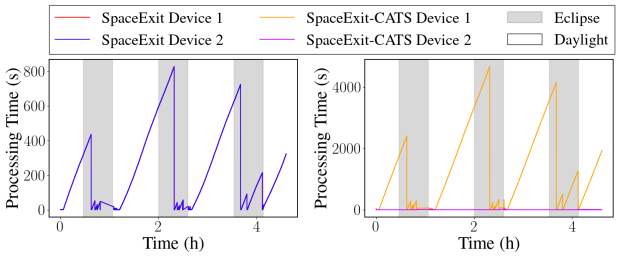
Figure 9 demonstrates that CATS significantly improves system throughput across all configurations. At 100 Mbps bandwidth, CATS achieves 11% improvement in throughput compared to the baseline without scheduling. This improvement is attributed to CATS's ability to effectively distribute workloads based on device capabilities and current utilization levels. The performance gain is particularly pronounced in the heterogeneous configuration, where intelligent task allocation can better leverage the complementary capabilities of different devices.

Figure 11 shows the estimated processing time for each device to complete all queued tasks at 100 Mbps bandwidth. With CATS, the processing time is balanced across devices, whereas without scheduling, the more powerful device finishes quickly, leaving the less powerful device with a backlog. From the data in the figure, the estimated processing time without scheduling is approximately five times higher than with scheduling. This imbalance highlights the inefficiency of not using CATS, as it leads to underutilization of the more capable device and overloading of the less capable one. Additionally, the figure shows a sharp decline in processing time at certain points, which corresponds to the moments when the satellite begins communication with the ground station. These moments act as deadlines for task processing, preventing storage from being overwhelmed by pending tasks.

These results demonstrate that CATS successfully addresses the challenge of efficient resource utilization in heterogeneous satellite computing environments. The scheduler's ability to maintain high throughput while reducing waiting times is particularly valuable for time-sensitive Earth observation applications, where processing delays can significantly impact the utility of collected data. The consistent performance improvements across different hardware configurations and bandwidth conditions underscore the robustness of our scheduling approach.

Table 1: Comparison of static method and GCAD.

Class	Static		GCAD	
	mAP@.5	mAP@.5:.95	mAP@.5	mAP@.5:.95
Plane	0.911	0.643	0.918	0.666
Ship	0.866	0.577	0.872	0.598
ST	0.732	0.427	0.764	0.464
BD	0.721	0.395	0.723	0.405
TC	0.937	0.831	0.932	0.833
BC	0.661	0.480	0.608	0.434
GTF	0.468	0.266	0.470	0.253
Harbor	0.815	0.440	0.836	0.460
Bridge	0.399	0.159	0.430	0.192
LV	0.800	0.559	0.814	0.583
SV	0.582	0.288	0.594	0.297
HC	0.423	0.188	0.355	0.186
RA	0.595	0.354	0.607	0.364
SBF	0.564	0.399	0.551	0.397
SP	0.688	0.289	0.693	0.312
MEAN	**0.635**	**0.394**	**0.64**	**0.404**



(a) Processing time without CATS, imbalanced task distribution.
(b) Processing time with CATS, balanced task distribution.

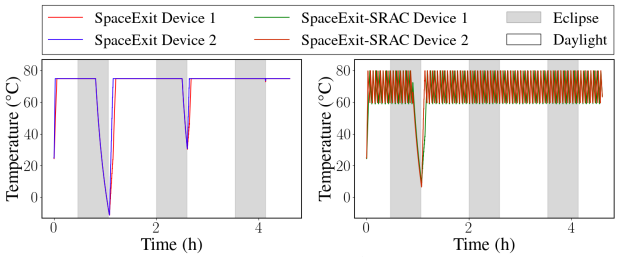
Figure 11: CATS reduces the expected processing time.

5.4 SRAC Guarantees Safe Execution In Space

We evaluate SRAC's effectiveness in maintaining safe and efficient system operation across different hardware configurations under realistic space constraints. Using a bandwidth of 100 Mbps, we tested HW Set-1, HW Set-2 and HW Set-3. Our experiments demonstrate SRAC's ability to balance processing throughput with thermal and power constraints.

Figure 10 illustrates the normalized throughput across different hardware configurations, comparing performance with and without SRAC. With SRAC enabled, the system achieves consistently higher throughput while operating within safe thermal limits. This improvement stems from SRAC's ability to adaptively adjust workload distribution based on real-time temperature and power measurements.

The thermal management capabilities of SRAC are clearly demonstrated in Figure 12. With SRAC enabled (Figure 12 (a)), device temperatures are maintained below the critical threshold of 80°C through proactive DVFS adjustments. The temperature profile shows stable thermal management with minimal fluctuations, maintaining a consistent range within



(a) Temperature of computing devices with SRAC.
(b) Temperature of computing devices without SRAC.

Figure 12: Temperature profiles of computing devices with and without SRAC.

70-80°C during normal operation. In contrast, systems without SRAC (Figure 12 (b)) frequently experience temperature spikes above 80°C, triggering emergency thermal throttling that significantly impacts performance.

These results highlight the crucial role of SRAC in satellite operations, where thermal management directly impacts system reliability and longevity. The system's ability to proactively adjust processing loads and power states ensures consistent performance without compromising hardware safety, making it well-suited for the challenging thermal environment of space operations.

6 Related Work

6.1 Orbitital Edge Computing

Orbital Edge Computing (OEC) has emerged as a promising approach to address the growing challenges of Earth observation data processing and transmission [9, 11, 37, 39]. The foundational concept and architecture of OEC were introduced by Denby and Lucia, who demonstrated that processing remote sensing images onboard satellites can significantly alleviate the communication bottleneck between space and ground segments [11]. Subsequent research has expanded the scope of OEC, with numerous systems built to process Earth observation tasks [8, 9, 40]. For instance, Kodan maximizes the utility of saturated satellite downlinks while mitigating the computational bottleneck [9], while EagleEye proposes a mixed-resolution, leader-follower constellation design [8]. Recent proposals have even discussed the possibility of building microdatacenters with 4KW power in space [5], further expanding the potential capabilities of OEC.

While these OEC approaches have made significant progress in enabling onboard data processing and reducing transmission volumes, they generally rely on fixed processing pipelines or pre-defined decision criteria. This limits their adaptability to the diverse and dynamic EO tasks.

6.2 Resource Management in Satellite Systems

Resource management in satellite systems, particularly in OEC contexts, faces fundamental challenges in balancing limited bandwidth, computation, and energy resources [7, 12]. Research has explored various approaches to optimize computation offloading and resource allocation, including deep reinforcement learning and game theory [14,41]. Energy management remains critical for solar-powered satellite operations [42], with optimization frameworks employing techniques like mixed-integer programming to balance latency, energy consumption, and computational throughput [6].

Despite these advancements, current resource management strategies for satellite systems often lack close integration with the specific requirements of dynamic onboard image processing and object detection tasks.

6.3 Dynamic Neural Network Inference

Dynamic neural network inference has emerged as a promising approach to balance accuracy and efficiency in various applications [15, 18, 30, 31]. Early exit networks, which dynamically terminate inference at earlier layers, represent a significant advancement in this field [22]. Several methods have been proposed in this direction, including BranchyNet [32], SkipNet [36], MSDNet [21], LoRAExit [26] and MMexit [17]. These approaches typically add early exit points to neural networks, enabling dynamic inference times for classification tasks and reducing average computation time for easily classifiable inputs. Few works have extended dynamic inference to object detection. For instance, DynamicDet employs an adaptive router to analyze multi-scale information and automatically determine the inference route [25]. The computational efficiency of these methods has led to their adoption in resource-constrained environments [16, 19].

Despite these advancements, current dynamic inference methods primarily target classification tasks and general-purpose computing environments. They have yet to fully address the unique challenges posed by object detection in satellite imagery or the specific constraints of orbital edge computing systems.

7 Conclusion

In this paper, we presented *SpaceExit*, an integrated system that enables efficient adaptive Earth observation on satellites. By leveraging a novel multi-scale feature backbone with adaptive exiting, a unified adaptive scheduling framework, and a runtime resource management module, SpaceExit addresses the unique challenges of orbital edge computing. Our comprehensive evaluations across diverse satellite settings and hardware platforms demonstrate that SpaceExit significantly outperforms existing methods. By enabling adaptive, efficient onboard processing, SpaceExit paves the way for a new generation of autonomous and capable Earth observation satellites. It demonstrates the potential to maximize the value of limited downlink bandwidth, opening new possibilities for space-based data analytics and real-time global monitoring.

Acknowledgment

We sincerely thank all the anonymous reviewers and our shepherd Divya Mahajan for their valuable comments and feedback. This work is supported by the National Natural Science Foundation of China (No. U23A6007), and also supported in part by the SJTU-SAST Joint Research Grant (USCAST2022-16). Corresponding authors are Xiaofeng Hou and Chao Li.

References

[1] Starlink specifications - starlink. [Online; accessed 2025-01-13].

[2] Moloud Abdar, Farhad Pourpanah, Sadiq Hussain, Dana Rezazadegan, Li Liu, Mohammad Ghavamzadeh, Paul Fieguth, Xiaochun Cao, Abbas Khosravi, U Rajendra Acharya, et al. A review of uncertainty quantification in deep learning: Techniques, applications and challenges. *Information fusion*, 76:243–297, 2021.

[3] David J Barnhart, Tanya Vladimirova, and Martin N Sweeting. Very-small-satellite design for distributed space missions. *Journal of Spacecraft and Rockets*, 44(6):1294–1306, 2007.

[4] Elinor Benami, Zhenong Jin, Michael R Carter, Aniruddha Ghosh, Robert J Hijmans, Andrew Hobbs, Benson Kenduiywo, and David B Lobell. Uniting remote sensing, crop modelling and economics for agricultural risk management. *Nature Reviews Earth & Environment*, 2(2):140–159, 2021.

[5] Nathaniel Bleier, Muhammad Husnain Mubarik, Gary R Swenson, and Rakesh Kumar. Space microdatacenters. *2023 56th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 900–915, 2023.

[6] Marco Centenaro, Cristina E Costa, Fabrizio Granelli, Claudio Sacchi, and Lorenzo Vangelista. A survey on technologies, standards and open challenges in satellite iot. *IEEE Communications Surveys & Tutorials*, 23(3):1693–1720, 2021.

[7] Qian Chen, Zheng Guo, Weixiao Meng, Shuai Han, Cheng Li, and Tony QS Quek. A survey on resource management in joint communication and computing-embedded sagin. *IEEE Communications Surveys & Tutorials*, 2024.

[8] Zhuo Cheng, Bradley Denby, Kyle McCleary, and Brandon Lucia. Eagleeye: Nanosatellite constellation design for high-coverage, high-resolution sensing. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2024.

[9] Bradley Denby, Krishna Chintalapudi, Ranveer Chandra, Brandon Lucia, and Shadi Abdollahian Noghabi. Kodan: Addressing the computational bottleneck in space. *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, 2023.

[10] Bradley Denby and Brandon Lucia. Orbital edge computing: Machine inference in space. *IEEE Computer Architecture Letters*, 18(1):59–62, 2019.

[11] Bradley Denby and Brandon Lucia. Orbital edge computing: Nanosatellite constellations as a new class of computer system. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 939–954, 2020.

[12] Changfeng Ding, Jun-Bo Wang, Ming Cheng, Min Lin, and Julian Cheng. Dynamic transmission and computation resource optimization for dense leo satellite assisted mobile-edge computing. *IEEE Transactions on Communications*, 71(5):3087–3102, 2023.

[13] Stijn Eyerman and Lieven Eeckhout. Fine-grained dvfs using on-chip regulators. *ACM Transactions on Architecture and Code Optimization (TACO)*, 8(1):1–24, 2011.

[14] Shu Fu, Jie Gao, and Lian Zhao. Integrated resource management for terrestrial-satellite systems. *IEEE Transactions on Vehicular Technology*, 69(3):3256–3266, 2020.

[15] Yizeng Han, Gao Huang, Shiji Song, Le Yang, Honghui Wang, and Yulin Wang. Dynamic neural networks: A survey. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 44(11):7436–7456, 2021.

[16] Xiaofeng Hou, Jiacheng Liu, Xuehan Tang, Chao Li, Jia Chen, Luhong Liang, Kwang-Ting Cheng, and Minyi Guo. Architecting efficient multi-modal aiot systems. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, ISCA '23, New York, NY, USA, 2023. Association for Computing Machinery.

[17] Xiaofeng Hou, Jiacheng Liu, Xuehan Tang, Chao Li, Kwang-Ting Cheng, Li Li, and Minyi Guo. Mmexit: Enabling fast and efficient multi-modal dnn inference with adaptive network exits. In *European Conference on Parallel Processing*, pages 426–440. Springer, 2023.

[18] Xiaofeng Hou, Peng Tang, Chao Li, Jiacheng Liu, Cheng Xu, Kwang-Ting Cheng, and Minyi Guo. Smg: A system-level modality gating facility for fast and energy-efficient multimodal computing. In *2023 IEEE Real-Time Systems Symposium (RTSS)*, pages 291–303. IEEE, 2023.

[19] Xiaofeng Hou, Cheng Xu, Chao Li, Jiacheng Liu, Xuehan Tang, Kwang-Ting Cheng, and Minyi Guo. Improving efficiency in multi-modal autonomous embedded systems through adaptive gating. *IEEE Transactions on Computers*, 2024.

[20] Jie Hu, Li Shen, and Gang Sun. Squeeze-and-excitation networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 7132–7141, 2018.

[21] Gao Huang, Danlu Chen, Tianhong Li, Felix Wu, Laurens Van Der Maaten, and Kilian Q Weinberger. Multi-scale dense networks for resource efficient image classification. *arXiv preprint arXiv:1703.09844*, 2017.

[22] Stefanos Laskaridis, Alexandros Kouris, and Nicholas D Lane. Adaptive inference through early-exit networks: Design, challenges and directions. In *Proceedings of the 5th International Workshop on Embedded and Mobile Deep Learning*, pages 1–6, 2021.

[23] Tingting Liang, Xiaojie Chu, Yudong Liu, Yongtao Wang, Zhi Tang, Wei Chu, Jingdong Chen, and Haibin Ling. Cbnet: A composite backbone network architecture for object detection. *IEEE Transactions on Image Processing*, 31:6893–6906, 2022.

[24] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C Lawrence Zitnick. Microsoft coco: Common objects in context. In *Computer Vision–ECCV 2014: 13th European Conference, Zurich, Switzerland, September 6-12, 2014, Proceedings, Part V 13*, pages 740–755. Springer, 2014.

[25] Zhi-Hao Lin, Yongtao Wang, Jinhe Zhang, and Xiaojie Chu. Dynamicdet: A unified dynamic architecture for object detection. *2023 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 6282–6291, 2023.

[26] Jiacheng Liu, Peng Tang, Xiaofeng Hou, Chao Li, and Pheng-Ann Heng. Loraexit: Empowering dynamic modulation of llms in resource-limited settings using low-rank adapters. In *Findings of the Association for Computational Linguistics: EMNLP 2024*, pages 9211–9225, 2024.

[27] Martina Lofqvist and José Cano. Accelerating deep learning applications in space. *arXiv preprint arXiv:2007.11089*, 2020.

[28] Lori A Magruder, Sinead L Farrell, Amy Neuenschwander, Laura Duncanson, Beata Csatho, Sahra Kacimi, and Helen A Fricker. Monitoring earth's climate variables with satellite laser altimetry. *Nature Reviews Earth & Environment*, 5(2):120–136, 2024.

[29] Deepak Murugan, Akanksha Garg, and Dharmendra Singh. Development of an adaptive approach for precision agriculture monitoring with drone and satellite data. *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, 10(12):5322–5328, 2017.

[30] Yifei Pu, Chi Wang, Xiaofeng Hou, Cheng Xu, Jiacheng Liu, Jing Wang, Minyi Guo, and Chao Li. M2sn: Adaptive and dynamic multi-modal shortcut network architecture for latency-aware applications. In *2024 IEEE International Conference on Multimedia and Expo (ICME)*, pages 1–6. IEEE, 2024.

[31] Yifei Pu, Xinfeng Xia, Xiaofeng Hou, Chi Wang, Cheng Xu, Jiacheng Liu, Jing Wang, Minyi Guo, Jingling Yuan, and Chao Li. Mmbypass: Towards efficient multi-modal ai computing with adaptive bypass network. *Journal of Parallel and Distributed Computing*, 201:105078, 2025.

[32] Surat Teerapittayanon, Bradley McDanel, and Hsiang-Tsung Kung. Branchynet: Fast inference via early exiting from deep neural networks. In *2016 23rd international conference on pattern recognition (ICPR)*, pages 2464–2469. IEEE, 2016.

[33] Beth Tellman, Jonathan A Sullivan, Catherine Kuhn, Albert J Kettner, Colin S Doyle, G Robert Brakenridge, Tyler A Erickson, and Daniel A Slayback. Satellite imaging reveals increased proportion of population exposed to floods. *Nature*, 596(7870):80–86, 2021.

[34] Casper Versteeg and David L Cotten. Preliminary thermal analysis of small satellites. *Small Satellite Research Laboratory, The University of Georgia, Athens, Georgia*, 30602, 2018.

[35] Stefan Voigt, Fabio Giulio-Tonolo, Josh Lyons, Jan Kučera, Brenda Jones, Tobias Schneiderhan, Gabriel Platzeck, Kazuya Kaku, Manzul Kumar Hazarika, Lorant Czaran, et al. Global trends in satellite-based emergency mapping. *Science*, 353(6296):247–252, 2016.

[36] Xin Wang, Fisher Yu, Zi-Yi Dou, Trevor Darrell, and Joseph E Gonzalez. Skipnet: Learning dynamic routing in convolutional networks. In *Proceedings of the European conference on computer vision (ECCV)*, pages 409–424, 2018.

[37] Changhao Wu, Yuanchun Li, Mengwei Xu, Chongbin Guo, Zengshan Yin, Weiwei Gao, and Chuanxiu Chi. A comprehensive survey on orbital edge computing: Systems, applications, and algorithms. *arXiv preprint arXiv:2306.00275*, 2023.

[38] Gui-Song Xia, Xiang Bai, Jian Ding, Zhen Zhu, Serge Belongie, Jiebo Luo, Mihai Datcu, Marcello Pelillo, and Liangpei Zhang. Dota: A large-scale dataset for object detection in aerial images. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 3974–3983, 2018.

[39] Ruolin Xing, Mengwei Xu, Ao Zhou, Qing Li, Yiran Zhang, Feng Qian, and Shangguang Wang. Deciphering the enigma of satellite computing with cots devices: Measurement and analysis. *Mobicom*, 2024.

[40] Qiyang Zhang, Xin Yuan, Ruolin Xing, Yiran Zhang, Zimu Zheng, Xiao Ma, Mengwei Xu, Schahram Dustdar, and Shangguang Wang. Resource-efficient in-orbit detection of earth objects. *Infocom*, 2024.

[41] Yuru Zhang, Chen Chen, Lei Liu, Dapeng Lan, Hongbo Jiang, and Shaohua Wan. Aerial edge computing on orbit: A task offloading and allocation scheme. *IEEE Transactions on Network Science and Engineering*, 10(1):275–285, 2022.

[42] Dakai Zhu, Rami Melhem, and Daniel Mossé. The effects of energy management on reliability in real-time embedded systems. In *IEEE/ACM International Conference on Computer Aided Design, 2004. ICCAD-2004.*, pages 35–40. IEEE, 2004.

A Artifact Appendix

Abstract

This artifact provides the computational environment, codebase, datasets, and detailed instructions necessary to reproduce the key experimental results presented in the paper.

Scope

This artifact allows readers to validate the following core claims and results from the paper:

- **Simulation Results**: The performance metrics (goodput, processing time, temperature impact) of CATS and SRAC schedulers are presented in Figures 5, 11, 12. The data underlying the scheduler behavior visualizations in Figures 6, 9, 10.
- **Static and Dynamic Object Detection Accuracy**: The static and dynamic object detection accuracy results (mAP) are reported in Table 1 using YOLO and multi-exit detector implementations.
- **Accuracy-Computation Trade-off Analysis**: The accuracy-computation trade-off curve for the dynamic detector presented in Figure 8.

Contents

This artifact contains the following elements:

- **Environment Configuration:**
  - The `env.yml` file provides Conda environment specifications for dependency management.
  - Docker setup instructions are included for GPU-accelerated experiments.
- **Code & Scripts:**
  - Simulation Framework: The `simulator/` directory contains the complete implementation of CATS and SRAC scheduling algorithms, execution scripts, and Python visualization scripts for generating experimental figures.
  - Image Complexity Analysis: The `image_tiling/` directory provides the computational framework for calculating the image complexity metric.
  - Geospatial-Contextual Adaptive Detector: The `gcad/` directory implements the GCAD with subfolders for YOLO static experiments, multi-exit dynamic detector, and geospatial threshold tuning.
- **Dataset & Resources:**
  - Links to download required external datasets (DOTA, pre-generated simulator data).
  - Pre-trained model weights for YOLO and multi-exit detectors.
  - Pre-aggregated results file for Figure 8.

Hosting

The artifact is hosted on GitHub: https://github.com/zhuxz0299/ATC-artifact-submission. Datasets are available via Google Drive links in the README.

Requirements

We developed and tested the artifact with the following specifications:

- **Software Dependencies**: Docker, Python 3.8+, CUDA-capable GPU drivers
- **Hardware Requirements**: NVIDIA GPU with CUDA support (tested on RTX 4090), Multi-core CPU (tested on Intel Xeon Platinum 8336C @ 2.30GHz)
- **Operating System**: Linux distribution (tested on Ubuntu 20.04)

A.1 Installation

Python Environment (Conda)

1. Create Conda environment: `conda env create -f env.yml`
2. Activate environment: `conda activate <env_name>`

Docker Environment (GPU required)

1. Pull base image:

```
docker pull nvcr.io/nvidia/pytorch:21.08-py3
```

2. Create and run container:

```
docker run --gpus all -it -v ./gcad:/GCAD \
--shm-size=64g --name atc_artifact \
nvcr.io/nvidia/pytorch:21.08-py3
```

3. Install dependencies inside container:

```
apt update && apt install -y zip libgl1-mesa-glx
pip install seaborn thop
```

XRT: An Accelerator-Aware Runtime for Accelerated Chip Multiprocessors

Neel Patel
Cornell University
Ithaca, NY, USA

Mohammad Alian
Cornell University
Ithaca, NY, USA

Abstract

Datacenter applications spend a considerable portion of compute resources executing common functions. This has led to the deployment of accelerators capable of executing these functions with higher performance and energy efficiency. At the same time, datacenter applications require microsecond-scale response times and low tail latency. To meet these strict requirements, recent Chip Multi-Processors (CMPs) incorporate several on-chip accelerators. This enables fast communication between the general-purpose cores, direct accelerator access to the on-chip memory subsystem, and scalable sharing of accelerator resources across applications running on many general-purpose cores. Despite hardware support for on-chip accelerators, a lack of support at the runtime level prevents their efficient use at scale.

Our key insight in this work is that current runtimes are unsuitable for applications that make heavy use of on-chip accelerators, yielding suboptimal throughput–sometimes even worse than a system without accelerators. To address this problem, we develop XRT, a runtime for accelerated CMPs designed to scale to many-core, many-accelerator CPUs. Across a set of representative services, XRT achieves up to 3.2× higher throughput-under-SLO compared to an unoptimized runtime and never experiences slowdowns compared to a system that executes all request processing on general-purpose cores.

1 Introduction

The integration of accelerators onto server processors, positioned near general-purpose cores, represents a significant advancement in datacenter computing [20, 34, 39, 40, 48]. We refer to these server processors as Accelerated Chip Multi-Processors (XMPs). The accelerators on XMPs are designed to efficiently execute fixed-function kernels, offering the potential for substantial performance improvements and energy savings. As XMPs enter the datacenter [6, 8], it becomes increasingly important to increase accelerator utilization so their performance efficiency and application-level throughput benefits can be realized at scale.

State-of-the-art runtimes can meet the strict tail-latency Service-Level Objectives (SLOs) expected of datacenter applications [25, 29, 30, 36, 41], but they do not consider accelerators. These runtimes instead assume the entire end-to-end execution of a request takes place on a general-purpose core. With the current runtimes, there is no guarantee that application throughput increases when requests offload to accelerators. Our results show that, without a specialized runtime for XMPs, offloading to an accelerator can result in worse performance than performing all processing on the cores.

To enumerate the challenges with adapting existing CMP runtimes for XMPs, we perform a design space exploration of prevailing CMP runtime architectures which yields three insights. First, we find that a centralized scheduler–similar to the one used in state-of-the-art runtimes like Concord [29]–quickly becomes the bottleneck on an XMP–as it is tasked with both managing accelerator notifications and load balancing requests to the worker cores. Second, we observe that even a runtime like Tiny Quanta [36]–which implements a scalable two-level scheduling policy–incurs unnecessary context switches as the scheduler resumes threads that are still waiting on an offload to complete. Last, we find that, on an XMP, both cores and accelerators can become the system bottleneck. Since an accelerator must be invoked by a core, a contended accelerator can stall a core when offloading a function, reducing system throughput.

To address these challenges, we develop a scalable system architecture that divides the work of managing accelerator notifications across the many cores in the system. XRT mitigates stalls in request processing when accelerators are contended through the use of an efficient software fallback mechanism. XRT also implements an accelerator-aware request scheduler, which eliminates unnecessary context switches by only resuming threads whose offloads are guaranteed to be complete. Across a set of representative services, our design achieves up to 3.2× higher throughput under SLO than state-of-the-art CMP runtimes, up to 32× higher throughput than an unaccelerated system, and never experiences slowdowns compared

to a system without accelerators.

2 Background

The State of On-Chip Acceleration on Server CMPs. Today's commodity server processors integrate a multitude of accelerators for datacenter taxes and key functions like (de)compression, de/encryption, database queries, data movement, and AI workloads [20, 22, 23, 39, 40]. Large-scale fleet studies show these taxes consume a measurable share of CPU cycles, energy, and infrastructure spending, making it lucrative to move them from the cores to the more efficient accelerators [31, 32, 49]. Software support for accelerators comes in the form of vendor-provided libraries that expose APIs for programmers to invoke an accelerator's functionality. In many cases, these libraries implement the same APIs as the functions that run on the general-purpose core, enabling applications to integrate accelerators without any changes to the source code [5, 16, 44, 53]. Intel Xeon 4th and 5th generation XMPs offer SKUs with four different accelerators–the Data Streaming Accelerator (DSA), the In-Memory Analytics Accelerator (IAA), QuickAssist Technology (QAT), and the Dynamic Load Balancer (DLB) [54]. IAA targets analytics primitives, compression, and checksum operations, while DSA targets data movement and transformation. QAT also provides compression support as well as cryptography capabilities. DLB accelerates communication between cores using hardware-managed queues.

Hardware support has been introduced into Intel XMPs to reduce communication and data movement overheads between the cores and accelerators. To reduce memory access overheads, XMPs implement shared virtual memory (SVM) and device-side translation lookaside buffers, enabling accelerators to efficiently access a process's address space using the same virtual addresses as the core. To enable efficient sharing of accelerators between multiple cores, accelerators implement shared work queues, which are device-side hardware queues that contain in-flight offloads submitted from multiple processes. Shared work queues are useful in multi-tenant settings since they enable cores to enqueue offload descriptors (that include operations such as memcpy, decompress, etc.) to an accelerator using specialized instructions that return a "success" or "retry" status. A "retry" lets the core know that its submission failed due to insufficient work queue capacity or other errors [7].

A low overhead interface minimizes the *offload tax*, which is an overhead paid by the core when it submits an offload to an accelerator and when it checks for a notification that the offload has been completed. This enables even microsecond-scale requests to benefit from the increased performance efficiency provided by accelerators [54]. On an XMP, the end-to-end execution of a request can look like any of the three modes of execution shown in Figure 1: NoAcceleration, Block&Wait, and Yield. NoAcceleration does not utilize the available accelerators on the XMP, executing all functions on the core. As explained previously, on state-of-the-art XMPs, it is possible to run some functions on one of the available accelerators. We use the term Accelerable function to refer to any function which can be run on either an accelerator or a core. Block&Wait seeks to improve on the NoAcceleration mode by offloading an Accelerable Function to a specialized accelerator, however it leaves the core underutilized. Yield makes use of the wasted core cycles left behind by Block&Wait, using them to execute a Filler function, which can be a function from another request (Pre- or Post-Processing Function in Figure 1) or an entirely different application.

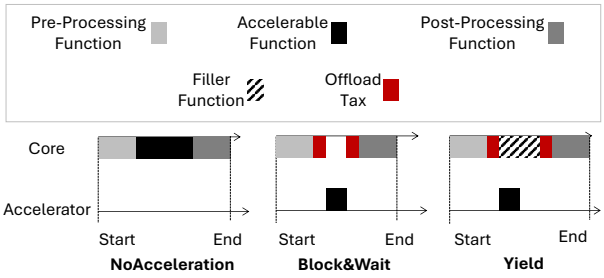


Figure 1: XMPs enable offloading Accelerable Functions to specialized accelerators. A Yielding mode of execution can make use of Blocking stalls when the core would otherwise be waiting for offload completion.

Current CMP Runtimes. State-of-the-art runtimes consider homogeneous CMPs, composed entirely of cores, and develop load balancing and scheduling policies to meet the SLOs of production workloads [25, 29, 30, 36]. These runtime systems use a dedicated thread, called the *dispatcher*, that runs on a single core and polls the NIC for incoming requests and enqueues them to a set of *worker* threads running on other cores. The workers are responsible for executing the requests [29, 30, 36].

A runtime system has two roles: (1) load balancing requests across the processing elements in the system and (2) scheduling the requests–choosing the request that executes next. State-of-the-art CMP runtimes fall into two categories: (1) *Dispatcher-Centric* where the dispatcher performs both load-balancing and request scheduling [29] and (2) *Worker-Centric*, which uses a two-level scheduling policy in which the dispatcher only distributes requests while each worker's local scheduler determines the execution order of its own requests [36]. Recent work has identified that more complex scheduling policies, like processor sharing [33], require a worker-centric, two-level scheduling policy–where the tasks of load-balancing and scheduling requests are divided between the dispatcher and workers–if they are to remain scalable to many-core systems [36]. In the era of XMPs, where even more intricate scheduling policies are needed to schedule functions on both cores and accelerators, this division of responsibilities becomes critical. A required feature of two-

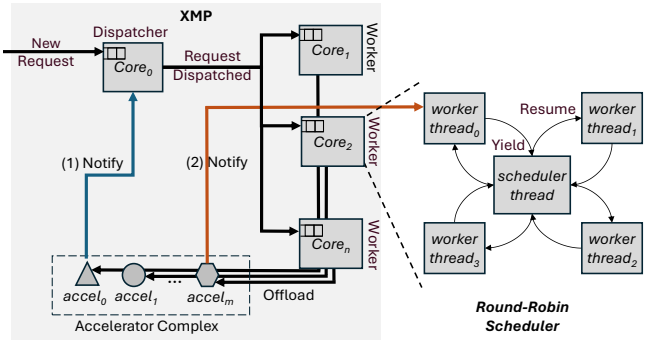


Figure 2: Integration of accelerators into existing CMP runtimes. In a Dispatcher-Centric system (Path 1), accelerator notifications are handled by the centralized dispatcher thread. In a Worker-Centric system (Path 2), notifications are handled by the core's local worker threads.

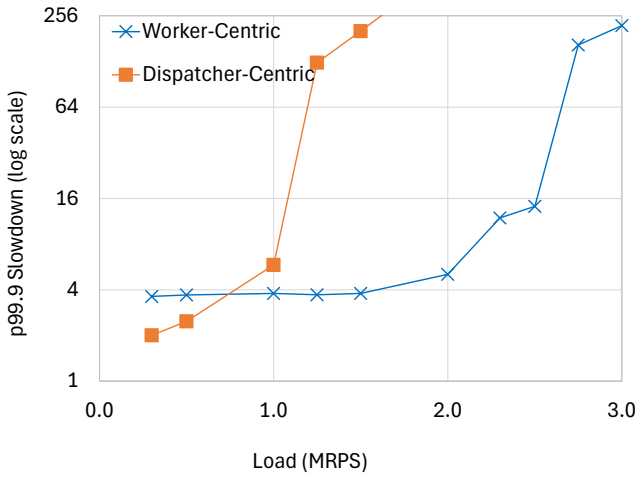


Figure 3: Comparison between Worker-Centric and Dispatcher-Centric runtime executing DDH workload on an XMP.

level scheduling policies is a userspace thread abstraction to enable worker cores to quickly context switch between different requests.

TinyQuanta [36]–which is the state-of-the-art Worker-Centric runtime–implements a shallow local queue per worker core that contains four outstanding requests, each represented by a userspace thread implemented using Boost coroutines [4]. TinyQuanta also maintains a fifth userspace thread context per worker core that implements the scheduling functionality of the runtime. The per-worker-core *scheduler thread* approximates processor sharing by cycling through its local queue in round-robin order, scheduling a thread that has either cooperatively yielded or is waiting to be scheduled for the first time.

3 A Runtime for XMPs

In this section, we search the space of CMP runtimes for a scalable system architecture suitable for XMPs. We start by integrating accelerator support into the two prevailing system architectures explained in Section 2, Worker-Centric and Dispatcher-Centric. Dispatcher-Centric can be advantageous since it has global visibility when making scheduling decisions, but complex scheduling algorithms, such as processor sharing, and notification signals from many accelerators can make the dispatcher a bottleneck. Worker-Centric removes the burden of scheduling from the dispatcher, but can only schedule a subset of the requests–those it has been given by the dispatcher. It is not abundantly clear which design better fits XMPs. To evaluate the tradeoffs of both designs, we develop runtimes that follow both approaches, adding support to enable applications to offload Accelerable Functions to accelerators.

We extend the centralized dispatcher in the baseline Dispatcher-Centric runtime to collect accelerator notifications (Path 1 in Figure 2). To enable a Worker-Centric runtime with accelerator support (Path 2 in Figure 2), we extend TinyQuanta [36]. This baseline Worker-Centric XMP runtime implements the Yield mode of execution by switching the worker core's context to the scheduler thread every time a thread yields. A thread yields after an offload, request completion, or failed polling of the accelerator. In this implementation, the thread context that performs an offload is responsible for polling the accelerator for completion. The polling is transparent to the scheduler thread, and the scheduler thread does not know whether a worker thread is going to begin executing a Pre-Processing or Post-Processing Function, or is polling for the completion of an Accelerable Function.

We compare the performance of the two runtime systems processing requests for an in-memory data stream processing workload (DDH in Table 1). Specifically, we evaluate the 99.9th percentile *slowdown* in request processing at different load levels. *Slowdown* at a given load is defined as the request processing latency normalized to the execution time in the Block&Wait mode. See Section 6 for more details on our methodology. Figure 3 shows that a Worker-Centric system avoids violating a 50× 99.9th percentile request slowdown SLO for this service up to approximately 2.5 MRPS, whereas the Dispatcher-Centric system only reaches around 1.0 MRPS. Our results indicate that the added burden of handling accelerator notifications at the dispatcher imposes a significant system throughput overhead. This highlights a fundamental limitation of Dispatcher-Centric in supporting XMPs, and thus we implement XRT as a Worker-Centric runtime.

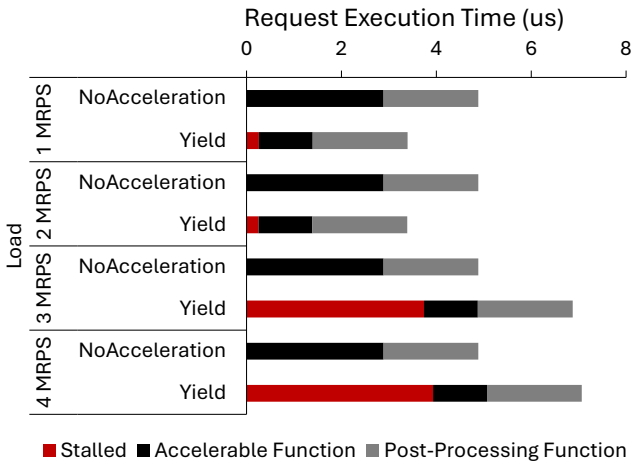


Figure 4: End-to-end breakdown of request execution time for a workload executing a memcpy Accelerable Function followed by a Post-Processing Function using the NoAcceleration and Yield modes of execution. "Stalled" is the time that a core waits for a full accelerator to become available.

4 Motivation

Despite demonstrating the potential of using Worker-Centric principles to develop a scalable runtime for XMPs, we identify two limitations in this baseline Worker-Centric runtimes:

Unnecessary Resumption Overhead. We find that a naive scheduler–unaware of requests' in-flight offloads–is unable to efficiently resume requests once their respective offloads are complete. Under the Yield mode of execution, worker threads relinquish the core to the scheduler after offloading. To complete request processing, the scheduler must later resume the offloading thread. The Worker-Centric runtime handles this by iterating through each busy worker thread in a round-robin manner, giving each thread a chance to complete any remaining Post-Processing Function after its offload has finished on an accelerator. However, this approach provides no guarantee that a worker thread's offload is complete when the scheduler resumes it. This leads to unnecessary resumptions, where the thread is woken up only to find that the offload is still in progress, causing it to yield the core back to the scheduler. In the experiment presented in Figure 3, each request experiences, on average, 21 unnecessary resumptions–indicating a substantial context-switching overhead. While the overhead of a single context switch is only ∼30 cycles, these wasted cycles accumulate. For the `DDH` workload, which has an unloaded per request latency of approximately 4μs with the Block&Wait mode of execution, unnecessary resumptions account for 13% of each request's end-to-end execution time.

Stalls on Contended Accelerators. Under the Yield mode of execution, an accelerator can become highly utilized as worker threads submit offloads and yield the core. This allows a single worker core to maintain multiple offloads in flight simultaneously. In contrast, under the Block&Wait mode of execution–where control is not yielded to the scheduler after offloading–the number of concurrent offloads is limited by the number of cores in the system (26 on our testbed system; see Section 6). Depending on the workload and request arrival rate, the accelerator's queue can become full as hundreds of worker threads issue concurrent offload requests. This results in threads stalling at offload time, as the accelerator is unable to accept additional requests.

To quantify the execution time spent stalling during an offload to a saturated accelerator, we run a synthetic workload that executes a memcpy function (i.e., Accelerable Function) followed by a Post-Processing Function that spins for 2μs. Figure 4 shows the breakdown of end-to-end request execution time as the server load increases under both Yield and NoAcceleration modes. As load increases toward the system's maximum sustainable throughput, request execution time in the Yield mode grows, eventually surpassing that of the NoAcceleration mode after reaching 3 MRPS. At high loads–where accelerator contention causes cores to spend a majority of their time stalled on a full accelerator–it becomes preferable to execute the Accelerable Function directly on the core rather than offloading it to the accelerator.

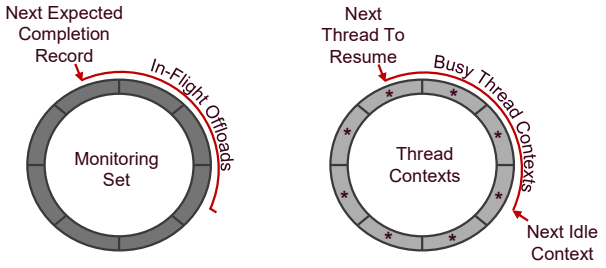


Figure 5: Ring buffers maintained by a worker core's scheduler thread. The scheduler periodically polls the `Next Expected Completion Record` to check for completion of an accelerator-offloaded function. If the *completion record* is set by the accelerator, the `Next Thread To Resume` is ready to run on the core.

5 XRT: An Optimized Runtime for XMPs

We develop **X**MP **R**un**T**ime (XRT) to address the limitations of the naive Worker-Centric runtime discussed in Section 3. Similar to TinyQuanta [36], XRT's two-level scheduling policy designates a single dispatcher core as the request load balancer and delegates request scheduling to the worker cores. At the dispatcher, XRT employs Join-the-Shortest-Queue (JSQ) load balancing, which has been shown to achieve near-optimal load distribution [45]. XRT maintains a shallow queue at each worker, eliminating the overhead incurred when worker cores poll an empty queue [29].

XRT mitigates unnecessary resumption overhead and reduces stalled core cycles on contended accelerators through a notification-aware scheduler and a software fallback mechanism.

5.1 Notification-Aware Scheduler

To avoid unnecessary request resumptions and the associated context-switching overhead, we design the worker's scheduler around the accelerator notification mechanism. The scheduler thread uses two ring buffers, shown in Figure 5, to track the state of pending offloads and manage the contexts of busy and idle userspace work threads.

The *Monitoring Set* ring buffer is an array of cacheline-sized *completion records*–simple data structures used by accelerators to notify the worker core when an offload is complete. Intel's accelerator interface implementation [52] includes a field in the offload descriptor specifying the address where the completion record should be written by the accelerator upon offload completion. XRT leverages this capability to populate the *Monitoring Set* ring buffer.

The *Thread Contexts* ring buffer is an array of pointers to worker thread contexts. The scheduler maintains a logical mapping between completion records and thread contexts, allowing it to promptly wake the correct idle thread once its offload completes.

By exploiting the fact that accelerators process offloads in a first-come, first-served order, the scheduler only needs to poll the completion record corresponding to the next expected offload completion.

XRT's notification-aware scheduler addresses unnecessary resumption overhead by allowing the scheduler to directly check the status of the next expected completion record in the L1 cache–within 2–3 cycles–compared to the baseline, where the scheduler context-switches to the worker thread and incurs approximately 30 cycles before determining that the offload has not yet completed.

5.2 Software Fallback

We leverage instruction set extensions on Intel XMPs [28] to implement a simple offload decision-making procedure. When attempting to offload an Accelerable Function, the worker core uses the ENQCMD instruction [7] to write an offload request descriptor to a memory-mapped register on the accelerator. In addition to performing a write-combined, uncacheable store operation, ENQCMD also returns a status indicating whether the offload was accepted by the accelerator. Rather than stalling on a full accelerator, XRT falls back to a core-executed software implementation of the Accelerable Function. This avoids wasting core cycles on repeated offload attempts.

5.3 Putting It All Together

Figure 6 shows the design of XRT. To illustrate its operation, we walk through the lifetime of a single request executed using XRT. For simplicity, we consider a three-phase application with a single Accelerable Function, similar to those shown in Figure 1.

A request is first received and parsed by the Dispatcher. The Dispatcher identifies the worker with the shortest queue of pending requests using a priority-queue-based implementation of the JSQ algorithm, which uses counters to track the number of enqueued requests at each worker. The request is then enqueued to the core of the worker with the fewest pending requests.

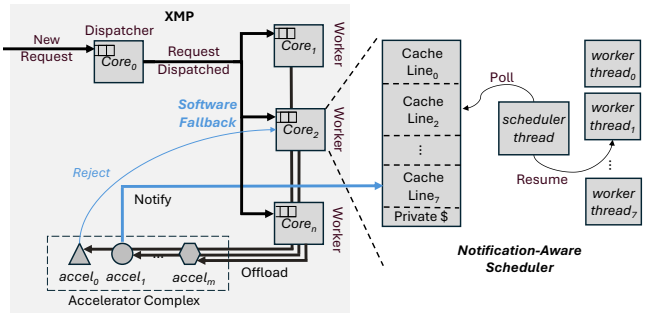


Figure 6: XRT uses a scalable two-level scheduling policy that features a notification-aware scheduler and a software fallback mechanism.

When the request reaches the head of the worker's queue, the scheduler thread assigns it to an idle thread and performs a context switch into that thread. The Pre-Processing Function then begins execution on the worker core. After the Pre-Processing Function completes, the worker attempts to offload the Accelerable Function to an accelerator. As described in Section 5.2, the worker makes a single offload attempt and falls back to a software implementation if the offload is unsuccessful.

If the offload is successful, the thread yields the core, allowing the worker to process other requests. Between request executions, the worker periodically polls the monitoring set for notifications from the accelerator. Upon receiving a notification, the worker resumes the corresponding ready-to-run thread, which is now guaranteed to have completed its offload. The thread then executes its Post-Processing Function, completing the request.

6 Methodology

Runtime Configurations. We compare the following runtime configurations: one that makes no use of accelerators (NoAcceleration); traditional block and wait runtime (Block&Wait); the naive XMP runtime presented in Section 3, which employs a round-robin scheduling policy to cyclically context-switch between threads that have yielded the core after offloading (RR-Worker); and XRT, which incorporates the optimizations described in Section 5.

Workloads. We develop the five representative workloads presented in Table 1. We select the combination of workloads to capture different computational intensities, access patterns, and use cases for the DSA and IAA accelerators on 4th and 5th generation Intel Xeon CPUs. Except for `DC` and `MC`, each workload follows the three-phase execution model illustrated in Figure 1. A request executes three functions: a Pre-Processing Function, an Accelerable Function, and a Post-Processing Function. The Pre-Processing Function and

Post-Processing Function always execute on a core, while the Accelerable Function can either run on a core or be offloaded to an accelerator. DC and MC do not have a Pre-Processing Function, modeling workloads where the request performs no core preprocessing and executes the Accelerable Function immediately. A request's response time is dependent on the input data size. Similar to prior work, we use an exponential service time distribution to represent μs-scale workloads with a light tail [29, 30, 36, 43] and choose the average input data size to match the respective workload [15, 27, 49–51].

Testbed. We run experiments on a 5th generation Intel Xeon 8571N CPU with four IAA and four DSA accelerators. We enable process address space identifiers (PASID), SVM, and configure accelerators with shared work queues using the accel-config library [17]. By configuring the accelerators with shared work queues, multiple cores can simultaneously submit offloads to the same accelerator, and receive notification of a failed offload attempt using the ENQCMD instruction [52], as explained in Section 5.2. Our testbed's CPU has two NUMA nodes. One NUMA node is configured to be the load generator, and the other is the server node. We have the load generator send requests according to a Poisson process to emulate bursty production traffic [29, 43].

In all experiments, all 26 cores are utilized–one for the dispatcher and the remaining as workers. Depending on the workload, worker threads offload to the two DSA or two IAA accelerators on the server node. Each accelerator is configured with a single shared work queue.

Metrics. The workloads under evaluation have very different latencies, so we report a single metric: 99.9th percentile slowdown, similar to prior work [29]. To calculate the slowdown of a request, the end-to-end request latency, including system overheads such as request dequeue and enqueue operations, is divided by the execution time of an uninterrupted, accelerated request running using the Block&Wait mode of execution.

7 Evaluation

In this section, we seek to evaluate XRT's performance on the workloads presented in Table 1. We compare against runtimes that have not been specialized for on-chip accelerators to show how XRT's optimizations improve performance.

For both workloads that use decompression (DDH and DC), NoAcceleration is outperformed by all other systems. The compute-intensive decompression function running on a core causes slowdowns even at low load levels. Despite most of the performance improvement coming from offloading decompression to the accelerator, XRT still outperforms RR-Worker on DDH and DC. Both XRT and RR-Worker outperform Block&Wait since they yield the core after submitting an offload to an accelerator, enabling higher core utilization.

Both workloads that perform memory copies only benefit from offloading when executing under the XRT runtime. Block&Wait cannot outperform NoAcceleration on DMD or MC because using an accelerator to perform memcpy of data held in a worker core's private caches actually takes longer than executing the memcpy on the core. RR-Worker succumbs to the throughput overheads that come with blocking on an accelerator, mentioned in Section 4. Since DMD and MC are accelerator-bound workloads, the accelerator becomes the bottleneck. We observe that, at high load, requests executing under the RR-Worker scheduler are slowed by 14% on MC and 188% on DMD, due to blocking on a full accelerator. Without a software-fallback mechanism, like the one implemented by XRT, both Block&Wait and RR-Worker suffer from the throughput overheads associated with blocking on an accelerator. RR-Worker outperforms Block&Wait due to the higher core utilization enabled by the Yield mode of execution, but it still suffers from overheads from unnecessary resumptions.

The only workloads in which XRT's tail latency improvements are negligible are MMP and UFH. This is because MMP is dominated by a lengthy Pre-Processing Function, which consumes 99.6% of the end-to-end execution time. Any overheads in the scheduler and offload mechanism are dwarfed by the long-running matrix-multiply which executes on the core, causing all systems to get similar performance. UFH also shows the same performance across the board since its Pre-Processing Function and Post-Processing Function make up the bulk of the execution time.

Workload				Description
ID	Pre-Processing	Accelerable	Post-Processing	
DDH	**D**eserialize	**D**ecompress	**H**ash	Representative of message processing in distributed streaming platforms [1, 2]. (De)serializes data structures [10], (de)compresses [42] to save network bandwidth, and applies a hashing algorithm [19, 37] to route between servers.
DC	—	**D**ecompress	**C**hase	Representative of data-intensive workloads like databases [26], key-value stores [27], and graph processing applications [47] which perform pointer chasing.
MC	—	**M**emcpy	**C**hase	Similar to DC but operates on uncompressed data.
MMP	**M**atMul	**M**emfill	**P**CA	Performs Principal Component Analysis (PCA) [51] after matrix multiplication and zeroing out half of the matrix. Representative of data analysis and preprocessing applications [46].
UFH	**U**pdate	**F**ilter	**H**istogram	Updates a table, extracts a subset of its entries, and constructs a histogram. Representative of data analytics workloads.
DMD	**D**ecrypt	**M**emcpy	**D**otProduct	Representative of similarity scoring services used for content recommendation [49, 50].

Table 1: List of workloads.

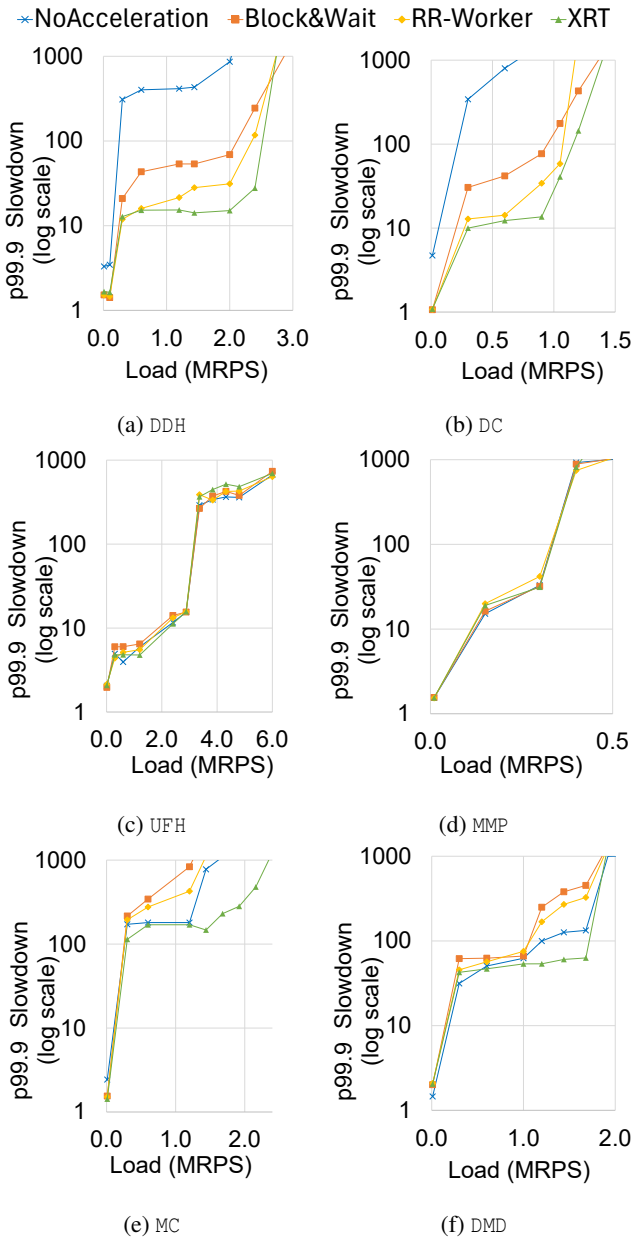


Figure 7: Performance comparison of workloads executing with different runtime configurations.

8 Discussion

Hardware Support for XRT. XRT relies on SVM to implement its notification-aware scheduler. Intel, AMD, and ARM already support SVM [12, 14, 52] and, given the support from vendors and industry standards [9, 13, 24, 38], future XMPs will likely support SVM. XRT also relies on instruction set extensions to implement software fallback. In our implementation, the ENQCMD [7] instruction from Intel's accelerator interface implementation [52] is used, but similar primitives, like RISC-V's Atomic IO Enqueue [11] and ARM's ST64BV [3], have been integrated on other architectures. These instructions would serve as drop-in replacements for ENQCMD when deploying XRT on other architectures.

Programming and Deployment XRT improves programmer productivity by enabling programmers to focus on application development rather than accelerator management and scheduling. Programmers write sequential programs using synchronous APIs [21], while XRT transparently handles scheduling decisions and accelerator management. XRT can be readily implemented in private clouds where the service provider has control over the entire hardware and software stack. Public cloud deployment is challenged by the fact that XRT is a single address space, userspace runtime, making it unsuitable to run mutually untrusted applications from multiple tenants. This could be addressed using userspace process abstractions which enforce isolation [35].

9 Conclusion

In this work, we demonstrated that existing runtime systems–originally designed for CMPs–are suboptimal for XMPs. We introduced XRT, a runtime system that efficiently integrates on-chip accelerators to support accelerable datacenter applications. XRT employs a scalable two-level scheduling policy to prevent the dispatcher thread from becoming a bottleneck. Additionally, it incorporates two key mechanisms–a notification-aware scheduler and a software fallback mechanism–to mitigate unnecessary resumption overhead and stalls on contended accelerators when worker threads offload parts of request processing to on-chip accelerators. We evaluated XRT using six representative accelerable workloads that utilize four distinct functions offered by accelerators on an Intel 5th Generation Xeon CPU. Across all workloads, XRT matches or outperforms an unaccelerated system and achieves up to 3.2× higher throughput compared to a runtime system without accelerator-specific optimizations. XRT represents an early but significant step toward enabling microsecond-scale accelerator offloads in future highly heterogeneous systems at scale.

Acknowledgments

This work was supported in part by NSF award 2239020 and by ACE one of the seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA. Any opinions, findings, conclusions, and recommendations expressed in this material are those of the authors and do not necessarily reflect those of the sponsors. We thank Dr. Ren Wang from Intel Labs for her valuable insights and guidance on this topic.

A Artifact Appendix

Abstract

This appendix describes the hardware and software needed to build and run XRT.

Hosting

XRT is hosted on GitHub.

Requirements

Tested on Intel Xeon 8571N CPU with two IAA and two DSA accelerators on Ubuntu 24.04. Our implementation of XRT uses the idxd-config [17] library version 4.1.6 to map IAA/DSA work queues into userspace. For enc/decryption, we use the ipp-crypto library version 2021.6 [18] for enc/decryption. For (de)serialization, we use protobuf version 28 [10]. The Principal Component Analysis, Matrix Multiply, and histogram kernels were taken from the pheonix benchmark suite [51]. All of these dependencies are built automatically using the convenience build script in the repository.

References

[1] Apache Flink® — Stateful Computations over Data Streams. Available at https://flink.apache.org/.

[2] Apache Pulsar | Apache Pulsar. Available at https://pulsar.apache.org/.

[3] Arm A64 Instruction Set Architecture. Available at https://developer.arm.com/documentation/ddi0596/2020-12/Base-Instructions/ST64BV--Single-copy-Atomic-64-byte-Store-with-Return-.

[4] Boost Coroutine. Available at https://www.boost.org/doc/libs/1_85_0/libs/coroutine/doc/html/coroutine/coroutine/asymmetric.html.

[5] Build Clickhouse with DEFLATE_qpl | ClickHouse Docs. Available at https://clickhouse.com/docs/en/development/building_and_benchmarking_deflate_qpl.

[6] C3 machine series on Intel Sapphire Rapids now GA. Available at https://cloud.google.com/blog/products/compute/c3-machine-series-on-intel-sapphire-rapids-now-ga.

[7] ENQCMD — Enqueue Command. Available at https://www.felixcloutier.com/x86/enqcmd.

[8] Introducing Amazon EC2 R7iz instances. Available at https://aws.amazon.com/about-aws/whats-new/2022/11/introducing-amazon-ec2-r7iz-instances/.

[9] The OpenCL™ Specification. Available at https://registry.khronos.org/OpenCL/specs/3.0-unified/html/OpenCL_API.html#_shared_virtual_memory.

[10] Protocol Buffers. Available at https://protobuf.dev/.

[11] RISC-V Summit Europe 2025 - Presentations. Available at https://riscv-europe.org/summit/2025/presentations.

[12] Shared Virtual Memory. Arm Immortalis and Mali GPU OpenCL Developer Guide. Available at https://developer.arm.com/documentation/101574/0600/OpenCL-2-0/Shared-virtual-memory.

[13] Standards – Heterogeneous System Architecture Foundation. Available at https://hsafoundation.com/standards/.

[14] Unified memory — HIP 6.2.41133 Documentation. Available at https://rocm.docs.amd.com/projects/HIP/en/docs-6.2.0/how-to/unified_memory.html.

[15] Introducing mcrouter: A memcached protocol router for scaling memcached deployments, September 2014. Available at https://engineering.fb.com/2014/09/15/web/introducing-mcrouter-a-memcached-protocol-router-for-scaling-memcached-deployments/.

[16] intel/DTO, May 2024, original-date: 2023-06-30T02:00:49Z. Available at https://github.com/intel/DTO.

[17] intel/idxd-config, March 2024, original-date: 2019-11-16T00:14:21Z. Available at https://github.com/intel/idxd-config.

[18] intel/ipp-crypto, September 2024, original-date: 2018-07-06T22:16:28Z. Available at https://github.com/intel/ipp-crypto.

[19] AAPPLEBY, smhasher/src/MurmurHash3.cpp. Available at https://github.com/aappleby/smhasher/blob/master/src/MurmurHash3.cpp.

[20] B. ABALI, B. BLANER, J. REILLY, M. KLEIN, A. MISHRA, C. B. AGRICOLA, B. SENDIR, A. BUYUKTOSUNOGLU, C. JACOBI, W. J. STARKE, H. MYNENI, and C. WANG, Data compression accelerator on IBM POWER9 and z15 processors : Industrial product, in *2020 ACM/IEEE 47th annual international symposium on computer architecture (ISCA)*, 2020, pp. 1–14. https://doi.org/10.1109/ISCA45697.2020.00012.

[21] L. BARROSO, M. MARTY, D. PATTERSON, and P. RANGANATHAN, Attack of the killer microseconds, *Communications of the ACM* **60** no. 4 (2017), 48–54 (en). https://doi.org/10.1145/3015146.

[22] R. BHARGAVA and K. TROESTER, AMD next-generation "zen 4" core and 4th gen AMD EPYC

server cpus, *IEEE Micro* **44** no. 3 (2024), 8–17. https://doi.org/10.1109/MM.2024.3375070.

[23] B. Blaner, B. Abali, B. M. Bass, S. Chari, R. Kalla, S. Kunkel, K. Lauricella, R. Leavens, J. J. Reilly, and P. A. Sandon, IBM POWER7+ processor on-chip accelerators for cryptography and active memory expansion, *IBM Journal of Research and Development* **57** no. 6 (2013), 3:1–3:16. https://doi.org/10.1147/JRD.2013.2280090.

[24] C. E. L. Consortium, Compute express link CXL 3.0 specification, 2024. Available at https://computeexpresslink.org/wp-content/uploads/2024/01/CXL_3.0-Specification-Release_FINAL-1.pdf.

[25] H. M. Demoulin, J. Fried, I. Pedisich, M. Kogias, B. T. Loo, L. T. X. Phan, and I. Zhang, When Idling is Ideal: Optimizing Tail-Latency for Heavy-Tailed Datacenter Workloads with Perséphone, in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, *SOSP '21*, Association for Computing Machinery, New York, NY, USA, October 2021, pp. 621–637. https://doi.org/10.1145/3477132.3483571.

[26] R. Elmasri, S. B. Navathe, R. Elmasri, and S. Navathe, Fundamentals of database systems, in *Advances in databases and information systems*, **139**, Springer, 2015.

[27] B. Fitzpatrick, Distributed caching with memcached, *Linux journal* **2004** no. 124 (2004), 5, Publisher: Belltown Media Houston, TX. Available at https://www.linuxjournal.com/article/7451.

[28] Intel, *Intel® Architecture Instruction Set Extensions and Future Features*, 319433-044 ed., 2021. Available at https://www.intel.com/content/dam/develop/external/us/en/documents/architecture-instruction-set-extensions-programming-reference.pdf.

[29] Iyer, Rishabh, Unal, Musa, Kogias, Marios, and Candea, George, Achieving Microsecond-Scale Tail Latency Efficiently with Approximate Optimal Scheduling | Proceedings of the 29th Symposium on Operating Systems Principles, Association for Computing Machinery, Koblenz, Germany. Available at https://dslab.epfl.ch/pubs/concord.pdf.

[30] K. Kaffes, T. Chong, J. T. Humphries, A. Belay, D. Mazières, and C. Kozyrakis, Shinjuku: preemptive scheduling for µsecond-scale tail latency, in *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation*, *NSDI'19*, USENIX Association, USA, February 2019, pp. 345–359.

[31] S. Kanev, J. Darago, K. Hazelwood, P. Ranganathan, T. Moseley, G.-Y. Wei, and D. Brooks, Profiling a warehouse-scale computer, in *ISCA '15 Proceedings of the 42nd Annual International Symposium on Computer Architecture*, 2014, pp. 158–169.

[32] S. Karandikar, A. Udipi, J. Choi, J. Whangbo, J. Zhao, S. Kanev, E. Lim, J. A. Alakuijala, V. Madduri, Y. S. Shao, B. Nikolic, K. Asanovic, and P. Ranganathan, CDPU: Co-designing Compression and Decompression Processing Units for Hyperscale Systems, in *Proceedings of the 50th Annual International Symposium on Computer Architecture*, New York, NY, USA, 2023. Available at https://dl.acm.org/doi/abs/10.1145/3579371.3589074.

[33] L. Kleinrock and R. R. Muntz, Processor sharing queueing models of mixed scheduling disciplines for time shared system, *Journal of The Acm* **19** no. 3 (1972), 464–482, Number of pages: 19 Place: New York, NY, USA Publisher: Association for Computing Machinery tex.issue_date: July 1972. https://doi.org/10.1145/321707.321717.

[34] C. Lichtenau, A. Buyuktosunoglu, R. Bertran, P. Figuli, C. Jacobi, N. Papandreou, H. Pozidis, A. Saporito, A. Sica, and E. Tzortzatos, AI accelerator on IBM Telum processor: industrial product, in *Proceedings of the 49th annual international symposium on computer architecture*, *Isca '22*, Association for Computing Machinery, New York, NY, USA, 2022, Number of pages: 17 Place: New York, New York, pp. 1012–1028. https://doi.org/10.1145/3470496.3533042.

[35] J. Lin, Y. Chen, S. Gao, and Y. Lu, Fast core scheduling with userspace process abstraction, in *Proceedings of the ACM SIGOPS 30th symposium on operating systems principles*, 2024, pp. 280–295.

[36] Z. Luo, S. Son, D. Bali, E. Amaro, A.Ousterhout, S. Ratnasamy, and S. Shenker, Efficient Microsecond-scale Blind Scheduling with Tiny Quanta, in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, *ASPLOS '24* **2**, Association for Computing Machinery, New York, NY, USA, April 2024, pp. 305–319. https://doi.org/10.1145/3620665.3640381.

[37] Mark Rabkin, Austin Appleby, FurcHash. Available at https://github.com/facebook/mcrouter/blob/main/mcrouter/lib/fbi/hash.c#L151.

[38] Millind Mittal, CCIX: Interconnect for Seamless Acceleration. Available at https:

//www.amd.com/content/dam/xilinx/imgs/developer-forum/2018-silicon-valley/CCIX-Interconnect-for-Seamless-Acceleration.pdf.

[39] A. O. MUNCH, N. NASSIF, C. L. MOLNAR, J. CROP, R. GAMMACK, C. P. JOSHI, G. ZELIC, K. MUNSHI, M. HUANG, C. R. MORGANTI, S. KANDULA, and A. BISWAS, 2.3 emerald rapids: 5th-generation intel® xeon® scalable processors, in *2024 IEEE international solid-state circuits conference (ISSCC)*, **67**, 2024, pp. 40–42. https://doi.org/10.1109/ISSCC49657.2024.10454434.

[40] N. NASSIF, A. O. MUNCH, C. L. MOLNAR, G. PASDAST, S. V. LYER, Z. YANG, O. MENDOZA, M. HUDDART, S. VENKATARAMAN, S. KANDULA, R. MAROM, A. M. KERN, B. BOWHILL, D. R. MULVIHILL, S. NIMMAGADDA, V. KALIDINDI, J. KRAUSE, M. M. HAQ, R. SHARMA, and K. DUDA, Sapphire rapids: The next-generation intel xeon scalable processor, in *2022 IEEE international solid-state circuits conference (ISSCC)*, **65**, 2022, pp. 44–46. https://doi.org/10.1109/ISSCC42614.2022.9731107.

[41] A. OUSTERHOUT, J. FRIED, J. BEHRENS, A. BELAY, and H. BALAKRISHNAN, Shenango: Achieving High {CPU} Efficiency for Latency-sensitive Datacenter Workloads, 2019, pp. 361–378 (en). Available at https://www.usenix.org/conference/nsdi19/presentation/ousterhout.

[42] P. DEUTSCH, DEFLATE Compressed Data Format Specification version 1.3, May 1996. Available at https://www.ietf.org/rfc/rfc1951.txt.

[43] G. PREKAS, M. KOGIAS, and E. BUGNION, ZygOS: Achieving Low Tail Latency for Microsecond-scale Networked Tasks, in *Proceedings of the 26th Symposium on Operating Systems Principles*, *SOSP '17*, Association for Computing Machinery, New York, NY, USA, October 2017, pp. 325–341. https://doi.org/10.1145/3132747.3132780.

[44] RAVINDRAN, BINURAJ AND GIACCHINO, LUCA AND JEE, KALYAN AND BABU, NANDITA NARENDRA, Intel® In-Memory Analytics Accelerator Plugin for RocksDB Storage Engine (Intel® IAA Plugin for RocksDB Storage Engine). Available at https://cdrdv2-public.intel.com/788783/357191EN-Intel-IAA-Plugin-RocksDB.pdf.

[45] H. SEYEDROUDBARI, S. VANAVASAM, and A. DAGLIS, Turbo: Smartnic-enabled dynamic load balancing of s-scale rpcs, in *2023 IEEE international symposium on high-performance computer architecture (HPCA)*, IEEE, 2023, pp. 1045–1058.

[46] J. SHLENS, A tutorial on principal component analysis, (2005). Available at https://www.cs.cmu.edu/~elaw/papers/pca.pdf.

[47] J. SHUN and G. E. BLELLOCH, Ligra: a lightweight graph processing framework for shared memory, in *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*, 2013, pp. 135–146.

[48] A. SMITH, E. CHAPMAN, C. PATEL, R. SWAMINATHAN, J. WUU, T. HUANG, W. JUNG, A. KAGANOV, H. MCINTYRE, and R. MANGASER, 11.1 AMD InstinctTM MI300 series modular chiplet package – HPC and AI accelerator for exa-class systems, in *2024 IEEE international solid-state circuits conference (ISSCC)*, **67**, 2024, pp. 490–492. https://doi.org/10.1109/ISSCC49657.2024.10454441.

[49] A. SRIRAMAN and A. DHANOTIA, Accelerometer: Understanding acceleration opportunities for data center overheads at hyperscale, in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 733–750.

[50] A. SRIRAMAN and T. F. WENISCH, Suite: A Benchmark Suite for Microservices, *2018 IEEE International Symposium on Workload Characterization (IISWC)* (2018), 1–12, Conference Name: 2018 IEEE International Symposium on Workload Characterization (IISWC) ISBN: 9781538667804 Place: Raleigh, NC Publisher: IEEE. https://doi.org/10.1109/IISWC.2018.8573515.

[51] J. TALBOT, R. M. YOO, and C. KOZYRAKIS, Phoenix++ modular mapreduce for shared-memory systems, in *Proceedings of the second international workshop on MapReduce and its applications*, 2011, pp. 9–16.

[52] UTKARSH Y KAKAIYA, SANJAY KUMAR, RAJESH MADUKKARUMUKUMANA SANKARAN, PRASHANT SETHI, Scalable I/O Between Accelerators and Host Processors. Available at https://www.intel.com/content/www/us/en/developer/articles/technical/scalable-io-between-accelerators-host-processors.html.

[53] WILL, BRIAN AND SHEMER, KAREN, Intel® QuickAssist Technology (Intel® QAT) - NGINX* Performance, (2023). Available at https://cdrdv2-public.intel.com/767645/Intel_QuickAssist_Technology_NGINX%20Performance_Whitepaper_767645v1.pdf.

[54] Y. YUAN, R. WANG, N. RANGANATHAN, N. RAO, S. KUMAR, P. LANTZ, V. SANJEEPAN, J. CABRERA, A. KWATRA, R. SANKARAN, I. JEONG, and N. S. KIM, Intel accelerators ecosystem: An SoC-oriented perspective : Industry product, in *2024 ACM/IEEE 51st annual international symposium on computer architecture (ISCA)*, 2024, pp. 848–862. https://doi.org/10.1109/ISCA59077.2024.00066.

DShuffle: DPU-Optimized Shuffle Framework for Large-scale Data Processing

Chen Ding[1]*, Sicen Li[1]*, Kai Lu[1]†, Ting Yao[2], Daohui Wang[2], Huatao Wu[2],
Jiguang Wan[1], Zhihu Tan[1], Changsheng Xie[1]
[1]*WNLO, Huazhong University of Science and Technology*
[2]*Huawei Cloud*

Abstract

Shuffle is a crucial operation in distributed data processing, responsible for transferring intermediate data between nodes. However, it is highly resource-intensive, consuming significant CPU power and often becoming a major performance bottleneck, particularly in data analysis tasks involving large datasets.

In this paper, we introduce DShuffle, an efficient framework that leverages DPUs to offload and accelerate shuffle operations. The DPU, with its specialized compute and I/O hardware, is ideally suited for offloading on-path shuffle tasks. However, its complex architecture requires careful design for effective offloading. To fully harness the DPU's capabilities, DShuffle divides the shuffle process into three stages: serialization, preprocessing, and I/O, and organizes them in a pipelined manner for efficient execution on the DPU. By leveraging high-concurrency memory access units to accelerate the serialization phase and using the DPU to directly write intermediate data to disk, DShuffle effectively accelerates the shuffle process and eliminates unnecessary data copies. Our experiments on a real DPU platform with industrial-grade Spark demonstrate that DShuffle enhances both host CPU and I/O efficiency and effectively reduce Spark task completion times.

1 Introduction

Distributed data processing frameworks, such as Hadoop [11] and Spark [47], have been widely used in large-scale data analytics over the past decade, including real-time analysis [3], machine learning [33], and data mining [4]. These frameworks enable parallel computation by distributing data across multiple nodes, effectively addressing the challenge of processing massive datasets that cannot be handled by single-node systems. At the same time, distributed data processing frameworks offer high scalability, allowing for the flexible addition of computational resources to accommodate the growing volume of data and increasing computational complexity.

Data shuffling is a fundamental component of these distributed data processing frameworks, responsible for exchanging intermediate data between nodes. Shuffling is a data-intensive operation involving complex preprocessing, serialization, network, and storage I/O operations. These operations consume significant CPU and memory resources on the host and can interfere with foreground computations, often becoming a performance bottleneck for the entire data analysis task. For example, in the commonly used sort workload, shuffle operations can account for up to 70% of the total execution time, with most of that time spent on serialization and memory garbage collection. As a result, many efforts have been made to optimize the shuffle process, from both software [9, 25, 30, 39, 41] and hardware [22, 28] perspectives.

DPU [2, 5, 7, 21, 31] is a new type of infrastructure widely deployed in cloud data centers [10, 15, 27]. Built on traditional NIC, DPUs are equipped with additional computing and memory units, which can be used to offload and accelerate host workloads, thereby providing opportunities to reduce shuffle overhead. In fact, the shuffle operation aligns well with the hardware features of the DPU. First, DPUs are accelerators situated in the data path, making them a natural fit for handling I/O-intensive data exchange operations. Second, DPUs include general-purpose computational cores that can be used to offload operations such as partitioning, aggregation, and sorting in the shuffle process. Finally, DPUs are equipped with highly concurrent memory access units (e.g., DPA) that can efficiently access host memory via load/store instructions, which offers an opportunity to significantly accelerate the serialization computation. By offloading the shuffle operation to the DPU, we can reduce the consumption of CPU and memory resources on the host, allowing the CPU to focus on higher-level application logic, thereby improving the overall efficiency of the cluster.

However, offloading shuffle to the DPU poses challenges. First, the number of general-purpose computational cores on

*Equal Contribution
†Corresponding Author

the DPU is limited, and their frequency is lower than that of host cores. As a result, simply offloading the entire shuffle operation to the DPU may not yield performance benefits and could even slow down the overall task execution. Second, the DPU's onboard memory capacity is limited, and when the intermediate data generated by host computation exceeds the DPU's memory capacity, the system must wait for the DPU to free up memory before continuing, which can lead to computation stalls.

To address these challenges, we propose DShuffle, a DPU-optimized shuffle framework. DShuffle decomposes the shuffle operation into three steps—serialization, computation, and I/O, and introduces three novel techniques to enhance the efficiency of shuffle operations on the DPU. (1) A DPA-accelerated serializer, which uses the multiple hardware threads to concurrently perform the serialization process, reducing CPU consumption and memory interference on the host. (2) A fine-grained pipeline shuffle strategy, which organizes different shuffle steps in a pipelined fashion, improving the overall execution efficiency and DPU usage. (3) A DPU-direct data spilling mechanism, which uses the DPU instead of the CPU to manage the spilling of intermediate data, eliminating redundant data copies and further reduces the CPU overhead.

We implemented a DShuffle prototype based on Spark [47] and NVIDIA's BlueField-3 DPU [36] and conducted detailed tests using the HiBench [20] benchmark suite. The test results show that DShuffle completely eliminates the CPU and memory overhead associated with shuffle operations, effectively reducing the execution time of Spark jobs and improving the overall resource efficiency of the cluster.

In summary, our contributions include:

- A comprehensive quantitative analysis of shuffle overhead from the perspectives of CPU, network, and I/O (§2).
- An exploration of the opportunities and challenges of offloading shuffle operations to DPUs (§3).
- A DPU-optimized shuffle framework (§4), incorporating three novel techniques: DPA-offloaded serialization (§5), fine-grained pipeline shuffle (§6), and DPU-direct data spilling (§7).
- A prototype system implemented on a real platform (§8), with an evaluation of its effectiveness (§9).

2 Shuffle Overhead

2.1 Spark Shuffle

Data shuffle is a fundamental operation in big data analysis and processing systems, such as Spark [47] and Hadoop [11]. It occurs when data needs to be redistributed across partitions

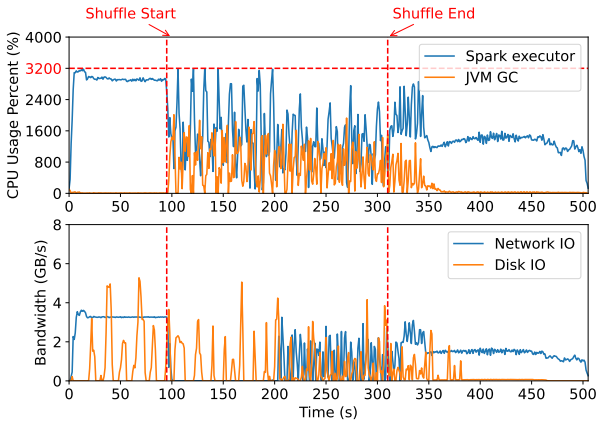


Figure 1: CPU Utilization, Disk and Network Bandwidth of a Two-node Spark Cluster under Sort Application.

to facilitate specific operations, such as groupByKey, sortByKey, or reduceByKey. The shuffling process ensures that related data elements are colocated within the same partition, enabling subsequent computations to run efficiently.

Here, we use the popular Spark as an example to explain the shuffle process. In Spark's shuffle process, (1) mappers first partition the data and reorganize tuples within each partition using operations like aggregation and sorting to prepare for parallel processing in the reduce stage. (2) They then serialize and compress the data into byte streams, store the intermediate results on disk, and notify the Spark driver of the data's location. (3) Reducers subsequently retrieve the data over the network, deserialize it, and apply user-defined reduce functions to complete the computation.

2.2 CPU Overhead of Shuffle

Shuffle supports efficient and scalable distributed computing by redistributing data across nodes. However, it is a CPU- and IO-intensive operation and is often the bottleneck of the entire task. To quantify the overhead of shuffle operations, we ran a 285GB HiBench sort workload on a 2-node Spark cluster.

Figure 1 illustrates the CPU, network, and IO consumption of the cluster during the task execution. It can be observed that the shuffle phase takes the longest time in the entire task execution, consuming approximately 226 seconds, while the rest map and reduce phases each take around 100 seconds. Additionally, we observed a significant amount of network and disk IO during the shuffle phase. Traditionally, shuffle operations have been limited primarily by these slow IOs. However, with the improvement in IO device performance, the CPU overhead of the shuffle has become increasingly noticeable. In our tests, shuffle consumed approximately 30 CPU cores, while the utilization of network and storage bandwidth was only 25% and 51%.

To further demonstrate the CPU overhead of the shuffle,

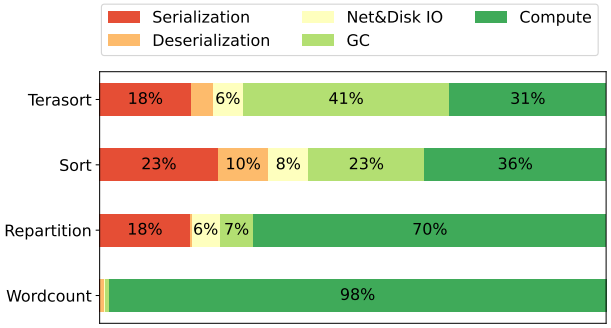


Figure 2: Time Breakdown of Different HiBench Workloads.

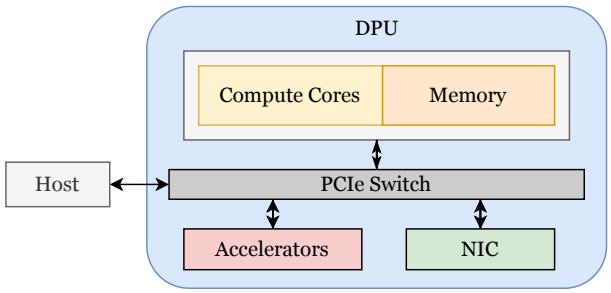


Figure 3: The DPU Architecture.

we conducted a fine-grained breakdown of the execution time for different types of workloads using HiBench [20]. As shown in Figure 2, network and disk I/O accounts for only about 1% to 8% of the total execution time, with most of the time spent on computational tasks such as (de)serialization, garbage collection, and map/reduce functions. For shuffle-intensive workloads, such as sort and terasort, (de)serialization and garbage collection account for approximately 64% to 69% of the total execution time. This indicates that these two steps are the primary factors that affect shuffle performance. We will now analyze these two operations in more detail.

Frequent (De)Serialization. During the shuffle process, a large amount of intermediate results are generated. These data are either persisted to disk or transmitted over the network. Each I/O operation requires data (de)serialization, which involves traversing, encoding, and converting Java object trees into byte streams. This process incurs significant CPU overhead. Although various optimization techniques (e.g., Kryo [14], Skyway [35]) have been proposed to reduce the computational overhead of serialization, they cannot entirely eliminate serialization costs since they still rely on host CPU cores for serialization. ZCOT [45] introduces a global metadata server to share JVM memory metadata across different nodes, allowing objects to be directly copied to the target node's memory via RDMA, thereby eliminating serialization overhead. However, it relies on additional CPU servers and results in a large number of small network I/O operations.

Expensive Garbage Collection. Shuffle serialization and I/O processes involve the allocation and release of a large number of temporary objects (e.g., byte streams, memory buffers, file handles). This is particularly true for large-scale datasets, which significantly increase the garbage collection (GC) pressure on the JVM [6, 24]. The test results in Figure 1 also support this point, showing that once the shuffle phase begins, CPU overhead caused by GC increases significantly. When the JVM performs garbage collection (GC), shuffle serialization and I/O processes must wait for it to complete, which can significantly impact the overall performance of the shuffle operation.

3 DPU as An Opportunity

3.1 Data Processing Units

The DPU (Data Processing Unit) is a programmable network card based on SoC, which adds additional compute, storage, and interconnect units compared to traditional network cards. These units can offload storage and networking functions from the host, reducing the CPU load. Nowadays, major chip manufacturers have released their own DPU products [2, 5, 7, 21, 31], and leading cloud service providers have widely deployed DPU chips in their infrastructure [10, 15, 27]. Figure 3 shows a typical DPU architecture which consists of five components:

- **High-performance network card** capable of providing hundreds of Gbps of network bandwidth. For example, the BlueField-3 is equipped with a ConnectX-7 network interface card, which includes one or two ports, each supporting up to 200 Gbps.
- **General-purpose compute cores**, often energy-efficient ones like ARM or RISC-V cores, offer high programmability that can be used to offload and execute arbitrary host logic.
- **Onboard memory**, typically with a certain amount of DDR memory and onboard flash storage for the operating system. For example, the BlueField-3 features 32 GB of onboard DDR5 memory.
- **Accelerators**, including those for data-intensive tasks (e.g., compression, encryption) and those for accelerating memory access operations (e.g., DMA, DPA).
- **PCIe switch** for interconnecting with the host and other PCIe devices.

3.2 Offload Opportunities and Challenges

The DPU offers an opportunity to reduce the CPU overhead of data shuffling. As an accelerator in the data path, offloading the shuffling operations of data flow across multiple devices

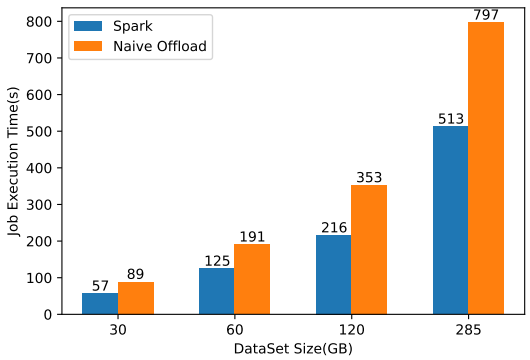


Figure 4: Task Completion Time for Spark and Naive Offload with Different Dataset Sizes under Sort Application.

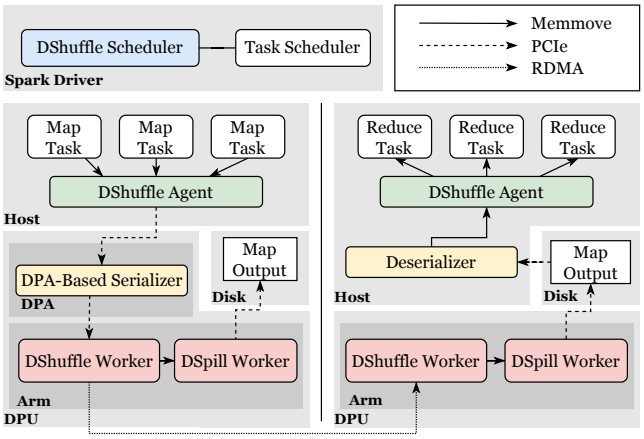


Figure 5: The DShuffle Architecture.

and nodes to the DPU is a natural idea. Additionally, we found that the hardware characteristics of the DPU align well with the shuffle process. In this section, we provide a detailed analysis of the opportunities and challenges of offloading data shuffling operations to the DPU.

Opportunities. Firstly, the DPU has efficient memory access units that can be used to accelerate the serialization process. Serialization operations primarily involve traversing the Java memory object tree, which is mostly a series of memory-intensive operations. The DPU offers high memory access efficiency, especially with its DPA (Data Path Accelerator), which has numerous memory access units (e.g. 256 hardware threads in BlueField-3) that can execute concurrently, allowing for efficient access to host memory through load/store instructions. By leveraging these concurrent memory access units for parallel serialization, serialization speed can be significantly improved, and the CPU overhead on the host can be reduced.

Secondly, the DPU is capable of coordinating data flow between nodes. It can communicate with storage devices on the host through PCIe P2P, and with remote nodes via network interfaces. Therefore, data transfer across devices and nodes can be managed by the DPU without involving the host CPU. This not only reduces the CPU overhead on the host but also effectively mitigates the impact of the large number of temporary objects generated during serialization and I/O on the host JVM, thus reducing GC overhead.

Furthermore, the DPU features general-purpose computation cores that can be used to perform preprocessing computations during the shuffle process, further reducing the CPU overhead on the host.

Challenges. However, there are several challenges associated with offloading shuffle operations to the DPU. Firstly, the general-purpose computation cores on the DPU are slower compared to x86 cores on the host. This can slow down the execution of certain data-intensive tasks, such as partitioning and sorting. Secondly, the DPU has limited onboard memory resources, which cannot accommodate large volumes of intermediate results generated by the shuffle. When the volume of intermediate results exceeds the DPU's memory threshold, the data produced by the map tasks on the host must wait until the DPU completes its processing before it can be sent back to the DPU, thereby affecting the overall execution efficiency of the task. Figure 4 illustrates the overall execution time of the task after decoupling and naive offloading the shuffle operation to the DPU. It can be observed that after offloading, the task completion time increased by a factor of 1.52 to 1.68, and as the dataset size increases, the performance degradation becomes more pronounced.

4 DShuffle Overview

Design goals. DShuffle aims to bridge the gap between the resource demands of data-intensive shuffle operations and the characteristics of DPU hardware. Its specific design goals are as follows:

- **Minimize shuffle overhead.** Shuffle operations consume significant host CPU and memory resources. DShuffle aims to reduce the consumption of host resources by offloading the shuffle task to the DPU, thereby freeing up more host computational resources for actual tasks like map and reduce.
- **Ensure shuffle performance.** The limited computational power and memory resources of the DPU may affect the execution efficiency of shuffle operations. DShuffle aims to accelerate the shuffle process through pipelining parallelism and hardware acceleration.
- **Ease of deployment.** Data systems often require major changes to adopt new technologies. DShuffle seeks to enable data systems to leverage its benefits with minimal modification.

Architecture. Figure 5 illustrates an architecture designed to achieve the aforementioned goals, which comprises a set

of components for collaboratively optimizing the shuffle process. Specifically, DShuffle offloads the shuffle execution to the DPU, leaving only a lightweight *DShuffle Agent* on the host. This agent receives shuffle requests initiated by the host, forwards them to the DPU for processing, and polls to retrieve the results. As previously mentioned, traditional shuffle processes are CPU-intensive and impose significant pressure on garbage collection (GC) in host memory. Executing the shuffle on the DPU effectively saves host CPU resources, allowing more computational power to be allocated to complex tasks such as map and reduce operations.

On the DPU, there are three components responsible for executing the shuffle operation:

- *DPA-Based Serializer.* It runs on DPA hardware threads and is primarily responsible for serializing Java objects in host memory and copying them into DPU memory.
- *DShuffle Worker.* It operates on the general-purpose compute cores of the DPU. For the map side, it handles data partitioning and aggregation, while on the reduce side, it focuses on merging and sorting the data.
- *DSpill Worker.* It manages the process of writing intermediate preprocessing results to local disks or transmitting them to remote nodes over the network. Leveraging the DPU's PCIe peer-to-peer (P2P) and networking capabilities, the DSpill worker can directly read/write local disks and send data to remote nodes without CPU intervention, thereby significantly reducing the host CPU load and memory pressure.

In addition to the above components, there is a *DShuffle Scheduler* running on the Spark driver node. This scheduler coordinates and manages host task execution based on the runtime states of the *DShuffle Worker* and *DSpill Worker* on the DPU. We next present a workflow to better illustrate how these components collaborate and work together.

Workflow. (1) When the intermediate results of a map task exceed the predefined threshold (e.g., 32MB), the Spark executor sends a signal to the DPA. The DPA then launches multiple hardware threads to perform serialization and transfers the results to the DPU memory via DMA, notifying the DShuffle Worker that a request has arrived. (2) The DShuffle Worker on the map side performs aggregation, partitioning, sorting, and other operations on the serialized data, and caches the intermediate results in DPU memory. When the amount of intermediate data exceeds the predefined memory threshold, the Map side triggers a DSpill Worker, which either spills the data to local disk or communicates with the DShuffle Scheduler to notify it to schedule a reduce target node. (3) If the target node is successfully acquired, the DSpill Worker on the map side sends the data to the reduce node via RDMA. (4) Upon receiving the data, the DSpill Worker on the reduce node notifies the DShuffle Worker on the reduce side to execute data merging. (5) The merged results are written to disk by the DSpill Worker on the reduce side, awaiting subsequent computational tasks to read directly from the disk.

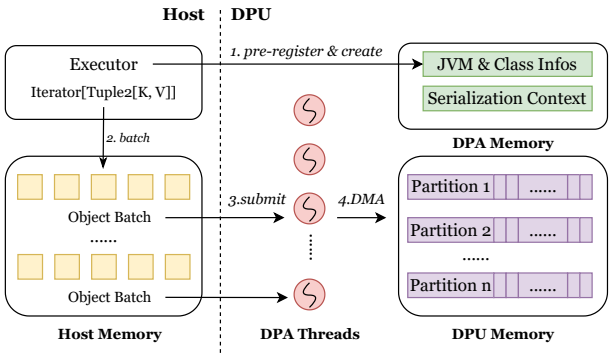


Figure 6: The DPA-based Serializer.

5 DPA-Offloaded Serialization

The state-of-the-art SmartShuffle offloads only the shuffle computation and network data transmission, leaving the serialization operations to be handled by an agent on the host. As demonstrated in Chapter 2, frequent serialization during the shuffle process incurs significant CPU overhead. Serialization involves intensive memory access and encoding operations, which consume substantial CPU cycles on the host. Additionally, the creation of numerous temporary objects during serialization imposes a heavy burden on the JVM's garbage collection (GC). Consequently, reducing serialization overhead on the host is also a must.

Offloading serialization to the DPU is a feasible solution as it reduces the strain on the host CPU and isolates its impact on JVM memory. However, utilizing the DPU for serialization is non-trival and comes with several challenges: (1) The input for serialization consists of discrete memory objects linked by pointers, which the DPU cannot inherently interpret; (2) The latency for DPUs to access host memory is higher than that of CPUs, and the clock speed of individual DPU cores is lower, potentially affecting serialization performance. To address these challenges, DShuffle introduces a DPA-based serializer. This serializer takes advantage of the DPA's hardware capabilities for byte-addressable and highly concurrent access to host memory, accelerating the serialization process.

DPA-based Serializer. Figure 6 shows the architecture of DPA-based serializer. It provides the same interface as other Java serializers, including initialization, type registration, serialization, deserialization, and destruction. As a result, the upper-layer application remains unaware of the specific serializer being used. Unlike CPU-based serializers, the DPA-based serializer retrieves the current JVM runtime parameters during initialization and passes them to the DPA. These parameters include the base address and size of the JVM heap memory, as well as JVM's compressed object pointer optimization settings.

Given that the DPA can access host memory using load/store instructions, it can use these parameters to access objects at any address within the JVM heap memory. When the serialization interface is invoked, the user only needs to provide the root object's address. The DPA hardware threads can then traverse the Java object tree from that address, copying scattered memory object fields into contiguous hardware memory to complete the serialization process. This approach effectively resolves the first challenge.

For the second challenge, the DPA-based serializer overcomes the long latency of single-threaded serialization by parallelizing the process using multiple DPA hardware threads. Specifically, the DPA serializer batches the objects to be serialized and, once a certain threshold is reached, activates multiple DPA hardware threads to serialize these objects in parallel, thereby improving serialization efficiency. As mentioned earlier, DPA has more than 190 hardware threads. By combining batching with multi-threading, the DPA serializer can achieve serialization speeds comparable to or even faster than those of the host machine.

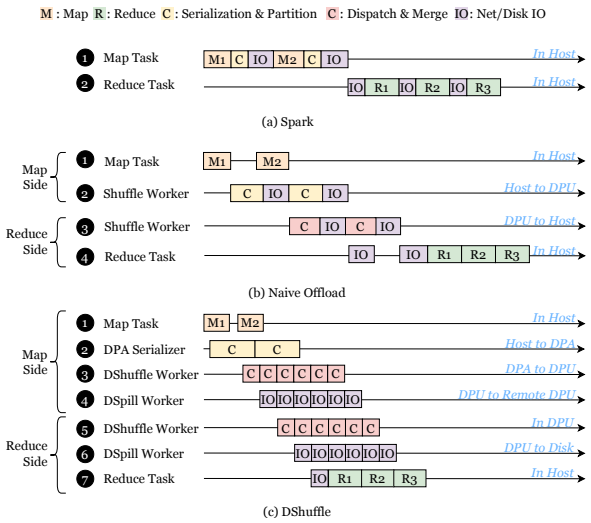


Figure 7: Execution Pipeline of Native Spark, Naive Offload Scheme and DShuffle.

6 Fine-Grained Pipeline Shuffle

The DPA-based serializer ensures fast serialization, but the shuffle process still involves additional steps such as preprocessing and I/O operations. Since the DPU has a limited number of general-purpose compute cores and is slower than the host CPU, the execution speed of these steps may be constrained, potentially impacting the overall shuffle performance. To address this challenge, existing approaches dynamically allocate tasks between the host and DPU based on the DPU's load. Although this prevents performance degradation, it introduces additional overhead on the host CPU.

In this paper, we propose a fine-grained pipelined shuffle technique that guarantees shuffle efficiency on the DPU, without consuming any host CPU resources. This is achieved by exploiting parallelism both across different shuffle workers and within individual shuffle workers. We will explain this approach in detail in this section.

Inter-Worker Parallelism. Figure 7 illustrates the differences between DShuffle, native Spark, and naive offload. In native Spark, map computations and shuffle operations are interleaved within the same thread, causing these two operations to block each other. The same applies to the reduce side, where the reduce task must wait for the shuffle to complete before it can start. Naive offload decouples the shuffle operation and offloads it to the DPU, enabling some overlap between map and serialization computations. However, due to the slower processing speed and limited memory capacity of the DPU, having it execute all preprocessing and I/O tasks in a single thread can negatively impact shuffle efficiency, thus blocking subsequent map computations. DShuffle improves the shuffle operation's execution efficiency on the DPU by splitting the data into fine-grained blocks and assigning them to preprocessing threads and I/O threads for pipelined execution. The specific process is described as follows.

On the map side, once the DPA Serializer completes serializing the intermediate results of the map computation, the corresponding DPA hardware thread is freed to serialize the next batch of intermediate results. Simultaneously, the DShuffle worker can start preprocessing the serialized results from the previous batch. After the DShuffle worker finishes its task, the results are passed to the DSpill worker for I/O, allowing the DShuffle worker to immediately begin preprocessing the next batch of serialized results. Through pipeline parallelism among workers, the shuffle operation's execution efficiency on the DPU is significantly enhanced.

On the reduce side, the pipeline process is essentially the reverse of the map side, with the distinction that the deserialization operation is handled by the DShuffle agent on the host. DShuffle assigns deserialization to the host CPU primarily because it involves creating a large number of Java objects. If the DPA were to handle this task, it would need to communicate with the JVM on the host, resulting in numerous small cross-PCIe requests, which would introduce significant access latency. By having the host agent handle this process, it can read a batch of data from the local disk at once for processing, ensuring the efficiency of deserialization.

Intra-Worker Parallelism. In the pipeline described above, if a worker in a specific stage processes data at a slower speed, it will impact the overall efficiency of the pipeline. To address this issue, DShuffle segments the data being processed by slower workers based on keys and then initiates multiple threads to process different segments in parallel. This approach enhances the execution speed of slower workers, ensuring that the throughput of different stages is balanced.

In addition to thread-level parallelism, DShuffle employs coroutines to enhance the resource utilization of DPU proces-

sors. Specifically, DShuffle breaks down each worker's execution process into finer-grained tasks, allowing these tasks to overlap on DPU computational cores. For example, a DShuffle worker's process is divided into three tasks: polling requests from the DPA Serializer result queue, performing preprocessing computations, and submitting I/O requests to the DSpill worker's request queue. These tasks are scheduled on demand to the DPU's general-purpose computational cores, avoiding the continuous occupation of CPU resources. For DPUs with a limited number of cores, this approach effectively prevents resource waste.

7 DPU-Direct Spilling

Beyond the computational overhead of serialization and preprocessing, the shuffle process can lead to significant data spill costs. When the intermediate result data generated by serialization and preprocessing exceeds the host memory threshold, spill operations are triggered to free up memory. This process not only blocks foreground computation but also consumes a considerable amount of host CPU resources to perform I/O operations. DShuffle mitigates these issues by leveraging high-speed DPA serialization to rapidly transfer intermediate results to DPU memory, thereby reducing foreground computation blocking and host memory usage. However, since DPU memory has limited capacity, spill operations remain unavoidable when its memory is insufficient.

To address this challenge, SmartShuffle spills unprocessed intermediate results to remote DPUs or the local host when the DPU memory reaches a certain threshold. While this approach alleviates the memory pressure on the DPU, it introduces redundant data movement and additional host CPU overhead. This is because data spilled to the local host or remote DPU is eventually handed over to the host CPU after preprocessing, which then writes the data to disk. These I/O operations not only consume host CPU resources but also create temporary objects during the process, increasing JVM garbage collection (GC) overhead.

In this paper, we propose a **DPU-Direct Spilling** scheme to further eliminate I/O overhead during the spill process. Figure 8 shows the key idea. Compared with native Spark, DShuffle leverages the DPU to manage data flow across devices and nodes. When shuffle intermediate results need to overflow or be written to disk, DShuffle enables the DPU to directly write intermediate results to the local disk via PCIe P2P or transmit them to remote DPUs via RDMA for direct disk writes. This approach completely eliminates CPU consumption and memory interference caused by I/O operations during the shuffle process.

The key to implementing DPU-Direct Spilling lies in enabling the host CPU to be aware of the data written to disk by the DPU. To achieve this, DShuffle employs a file preallocation strategy. Specifically, during node initialization, a disk partition is created, and several fixed-size files are allocated within this partition to accommodate spilled data. Once created, the metadata of these files remains unaltered. The partition is mounted as read-only on the host and as read-write on the DPU.

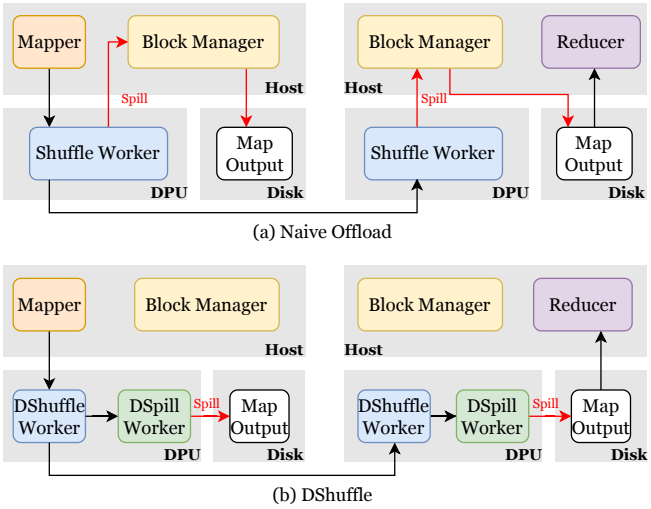


Figure 8: The Data Spilling Workflow of Naive Offload Scheme and DShuffle.

At runtime, when the shuffle scheduler dispatches shuffle requests to a node, the DPU on that node directly writes data to the partition through the local file system. After writing a fixed-sized block (e.g., 32MB), the DPU notifies the reducer on the host via DMA that new data is available. The reducer reads the file blocks based on metadata provided by the DPU and performs the necessary computations. Both the DPU's data writes and the reducer's reads utilize direct I/O to bypass the local file system cache, ensuring data consistency.

To effectively utilize the spill partition's space, the DPU also tracks the metadata of free file blocks. Once a file block is consumed by the reducer, the corresponding space is released and made available for subsequent spilled data writes.

8 Implementation

We use NVIDIA's BlueField-3 (BF-3) DPU as the platform for development and deployment. The BF-3 connects to the host system via a PCIe Gen 4.0 interface and runs Ubuntu 20.04 along with the DOCA 2.9 programming suite. The codebase for all DShuffle modules consists of approximately 6,700 lines and will be open-source.

Host. The DShuffle agent is divided into two parts. The first part is an interface integrated with Spark, comprising approximately 500 lines of Java code. The interface includes methods such as *Initialize*, *TriggerSpillStart*, *BatchAppendKV*, *WaitForSpillDone*, and *RegisterClassInfo*. Among these, *Initialize* and *RegisterClassInfo* handle preprocessing of Java type information or retrieve and register JVM startup parameters.

BatchAppendKV collects batched key-value objects into the serializer queue and then partitions them into the DMA buffer. Based on these interfaces, we developed a *DShuffleWriter* compatible with the native Spark shuffle writer, allowing seamless replacement and deployment. The second part is the native interface implemented via JNI, with approximately 1,000 lines of C++ code. This primarily handles interactions with the DPU, such as notifying the DPU to pull intermediate result buffers via DMA.

DPU. The DPU worker component consists of approximately 4,000 lines of C++ code. We implemented a unified data transmission layer that supports DMA, RDMA, and TCP communication protocols. On top of this layer, we developed two types of workers using the Boost.Fibers coroutine runtime: SpillWorker and ShuffleWorker. Tasks are passed between workers on the same DPU via single-producer single-consumer (SPSC) lock-free queues. The DPA Serializer, developed using the DOCA DPA module, is split into two parts. The first part, about 1,000 lines of code, is implemented using the JNI framework. It processes type metadata information in the JVM and handles interactions with the DPA hardware. The second part, approximately 1,200 lines, is the hardware code for the DPA, which implements serialization logic for Java object trees on DPA threads.

9 Evaluation

9.1 Goals

We evaluate DShuffle to answer the following questions:

- How much does DShuffle reduce the host CPU usage and improve the IO efficiency? (§9.3)
- How much can DShuffle reduce the completion time of spark jobs? (§9.4) How DShuffle performs under different workloads? (§9.5)
- How effective are the optimizations in each component at improving the efficiency of DShuffle? (§9.6)

These questions correspond to the first two design goals of DShuffle, while the third design goal has already been analyzed in 8.

9.2 Testbed and Methodology

System Setup. We evaluated DShuffle on a 2-node testbed, where each node is equipped with an Intel Xeon Gold CPU and an NVIDIA BlueField-3 DPU [36]. The detailed hardware and software configurations are shown in table 1. Note that we locked the CPU frequency at 4.0 GHz and disabled hyper-threading to ensure consistent performance across multiple runs. On each node, we deployed a 16-core, 32 GB Spark Executor, forming a two-node Spark cluster.

Intel Xeon 6418H Server
CPU: 24 Xeon cores @ 4.0GHz w/HT disabled **Memory:** 64GB DDR4 DRAM **SSD:** Samsung 980 pro 2TB **OS:** Ubuntu 22.04 LTS, Linux Kernel 6.8
NVIDIA BlueField-3 DPU
Network: ConnectX-7 two 100Gbps Ethernet ports **CPU:** 16 ARMv8.2 A78 cores **Memory:** 32GB DDR5 DRAM **DPA:** 16 RISC-V cores with 256 hardware threads **OS:** Ubuntu 20.04 LTS, Linux Kernel 5.15

Table 1: The Node Configuration of the Testbed

Workloads. We use HiBench [20] as our primary benchmarking tool. HiBench provides various types of workloads, including Sort, TeraSort, WordCount, and Repartition. Most of our tests are conducted using the Sort workload because it contains only a single Shuffle phase, making it ideal for demonstrating the optimization effects on the shuffle process. Nevertheless, we also evaluated other types of workloads. Unless otherwise specified, the dataset size used for evaluation is 285 GB with 32 partitions. Additionally, data compression in the Shuffle process was disabled during all tests to avoid additional CPU overhead on the host, which could reduce overall performance.

Baselines. Since the most relevant SmartShuffle [28] is not open-sourced, we use the following baselines to evaluate DShuffle:

- **Native Spark:** Using Spark 2.4.3 [17] as the baseline implementation without any offloading.
- **Naive Offloading:** Similar to the existing SmartShuffle, it offloads tasks such as partitioning, sorting, and network I/O of shuffle operations to the DPU, while serialization computation and intermediate data spilling are still handled by the host. The only difference from SmartShuffle is that Naive Offloading does not implement dynamic offloading logic. Since the goal of DShuffle is to fully offload the shuffle operation without sacrificing performance, using Naive Offloading as a baseline allows for a clearer demonstration of DShuffle's optimization benefits. Nevertheless, we provide a theoretical comparison between DShuffle and SmartShuffle in Section 10.
- **DShuffle:** The shuffle process offloads serialization, preprocessing, network I/O, and intermediate result spilling entirely to the DPU. These steps are organized in a pipeline manner, leveraging DPA for offloading and accelerating serialization. The DPU directly writes intermediate data to disk, ensuring efficient data handling throughout the process.

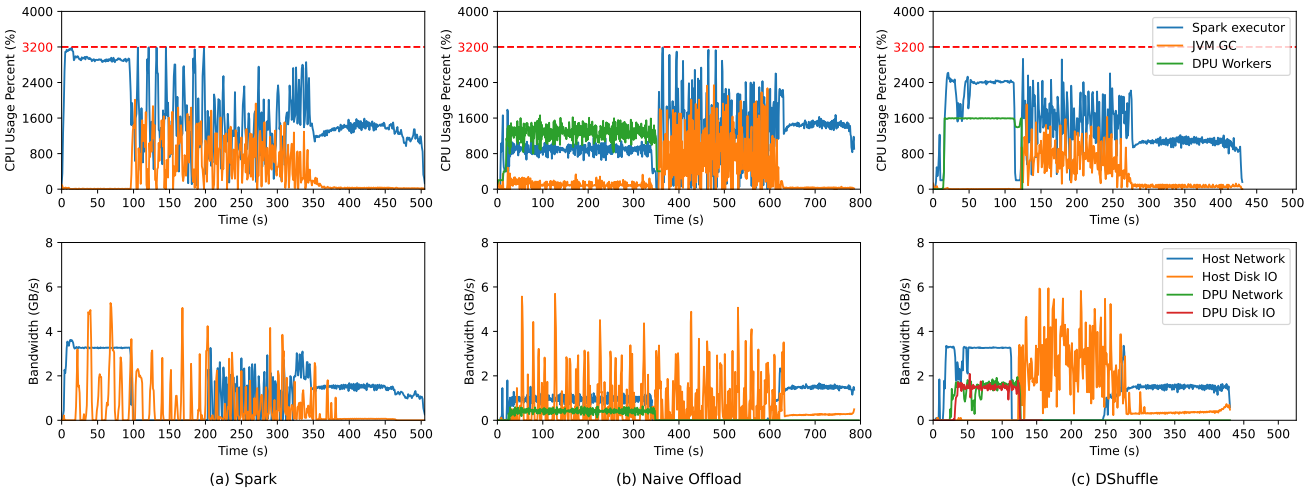


Figure 9: The CPU Utilization, Disk and Network Bandwidth of the Spark Cluster During the Execution of the Sort Application.

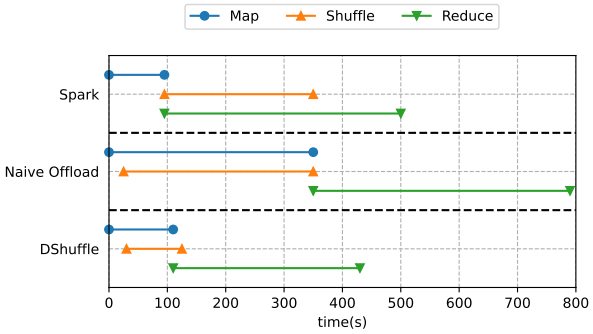


Figure 10: Execution Timeline for Native Spark, Naive Offload Scheme and DShuffle under Sort Application.

9.3 Resource Efficiency

Figure 9 illustrates the CPU utilization, disk, and network IO bandwidth of the Spark cluster during the execution of the sort application. It is worth noting that simple offloading increased the overall execution time of the task by 57%, while optimization through DShuffle reduced the overall execution time by approximately 16%. Below, we analyze the execution performance during the map and reduce phases for different comparison objects.

Map phase. Compared to native Spark, the naive offloading method decouples the shuffle from the host and offloads it to the DPU, overlapping it with the host's map computation. As a result, during the map phase, we observe the consumption of DPU compute cores and network IO. However, the naive offloading method does not reduce the overall execution time of the task. This is because the DPU cores are slower than the host cores, and naive offloading leads to a decrease in shuffle speed. When the DPU's memory capacity reaches its threshold, the map computation on the host needs to wait for the shuffle to complete, thereby affecting the overall task execution efficiency.

In contrast, DShuffle improves the execution speed of shuffle operations on the DPU through optimization techniques such as serialization acceleration, pipeline parallelism, and direct spilling, effectively eliminating the stall of map computations. This is also confirmed in Figure 10, which illustrates the execution timelines for each phase of different comparison groups. Compared to native Spark and naive offloading, DShuffle reduces the shuffle execution time by 62.7% and 70.7%, respectively. Additionally, since DShuffle offloads serialization to the DPA, the CPU resources consumed by the host during the map phase (24 cores) are lower than those in native Spark (30 cores).

Reduce phase. In native Spark, the reduce nodes pull data from the map nodes to the local machine for computation, which results in a significant amount of small network and disk I/O. Naive offloading improves network I/O efficiency by aggregating data and sending it to the reduce nodes in advance. However, during the reduce execution phase, naive offloading still relies on the host to perform serialization and file I/O, which introduces additional data copies and significant garbage collection (GC) overhead. Additionally, serialization can block the reduce computation, negatively impacting task execution efficiency.

In contrast, DShuffle leverages the DPU to directly write intermediate data to disk, eliminating redundant data transfer and improving I/O efficiency. As shown in Figure 1, DShuffle's I/O bandwidth is significantly higher than that of native Spark and naive offloading. Additionally, DShuffle uses DPA to offload and accelerate serialization, which reduces both the blocking effect on reduce tasks and the garbage collection (GC) overhead. This is also evident in Figure 1, where DShuffle exhibits lower CPU overhead due to GC and a more stable CPU utilization for reduce tasks. Overall, by implementing these optimization techniques, DShuffle reduces the execution time of reduce phase by approximately 45.6% and 50.2%, respectively, compared to native Spark and naive offloading.

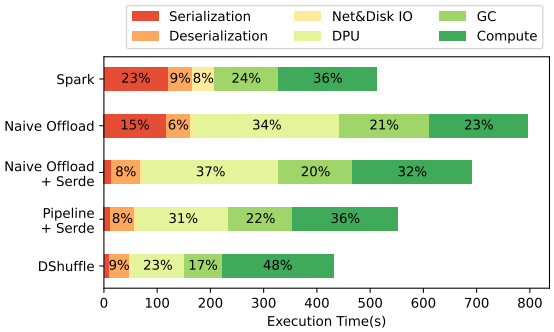


Figure 11: Time Breakdown of Spark Tasks with Different Optimization Techniques Turned on.

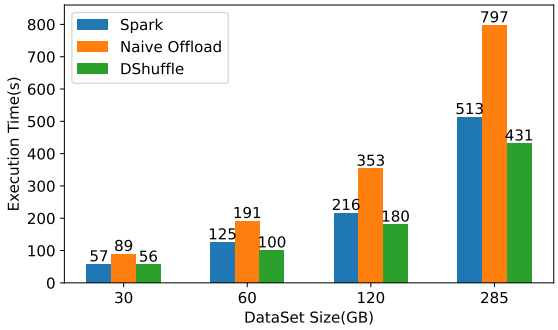


Figure 12: Execution Time of Different Comparison Groups under Different Dataset Size.

9.4 Performance Analysis

In this section, we will provide a detailed analysis of the performance advantages of DShuffle. First, we evaluate the performance gains brought about by different optimization techniques. Then, we examine the scalability of DShuffle's performance with respect to dataset size and the number of partitions.

Performance Breakdown. To demonstrate the performance gains from different optimization techniques, we progressively enabled various optimizations based on naive offloading: DPA-accelerated serialization, pipelined parallelism, and direct spilling via DPU. We then measured the overall execution time and performed a fine-grained time breakdown under different configurations. The test results are shown in Figure 11. Here, "Naive Offload + Serde" refers to enabling DPA serialization on top of the Naive Offload approach, while "Pipeline + Serde" further accelerates the shuffle steps on the DPU through pipelined parallelism.

It can be observed that DPA-accelerated serialization reduced the overall execution time of naive offloading by 13.2%, with serialization time decreasing from 15% to 3%. Additionally, offloading serialization reduced the GC pressure on the host, leading to a lower overall GC time. After enabling pipelined parallelism, the overall execution time was further reduced by 17.4% due to more efficient shuffle computation on the DPU. Finally, enabling direct spilling via DPU reduced the overall execution time by another 15.1%. This optimization eliminated redundant data copying, significantly reducing data transmission time on the DPU. Furthermore, offloading the I/O process to the DPU further alleviated GC pressure on the host, leading to a further reduction in GC time.

Performance with dataset size. To demonstrate the scalability of DShuffle with respect to dataset size. We generated sort datasets of 30GB, 60GB, 120GB, and 285GB to evaluate the task execution time of native Spark, naive offloading, and DShuffle. As shown in Figure 12, with the dataset size increases, the performance improvement of DShuffle becomes more pronounced. This is primarily because, with larger datasets, the IO overhead during the shuffle process becomes more significant, and the proportion of serialization and GC overhead is higher. DShuffle's optimizations effectively reduce these two types of overhead, resulting in greater performance gains.

Performance with partition number. We also tested the scalability of DShuffle in terms of partition number. We measured the task execution times of native Spark, naive offloading, and DShuffle under different partition numbers. As shown in Figure 13, with the number of partitions increases, the total task execution time for all comparison groups decreases, as the reduce tasks are more finely divided, which increases parallelism. The execution time for reduce tasks decreases because the amount of data being processed is smaller. However, for Map tasks, naive offloading becomes slower as the number of partitions increases. This is due to a higher number of network requests for data forwarding, which increases network overhead. In contrast, DShuffle, with its fixed pipeline granularity, ensures stable Map task execution times, thereby maintaining the same partition scalability as native Spark.

9.5 HiBench

In this section, we evaluate the benefits of DShuffle using different types of workloads from HiBench. The test workloads include Sort, TeraSort, Repartition, and Wordcount. The Sort workload involves sorting a 285GB dataset containing random-length random strings. TeraSort is similar to the Sort workload but mainly consists of smaller random strings, which results in greater GC pressure. The Repartition workload focuses on re-partitioning 300GB of random strings. Wordcount is primarily concerned with counting word frequencies in a 150GB dataset of random strings which is computationally heavy but has low shuffle overhead.

Figure 14 shows the test results. It can be observed that for workloads with high shuffle pressure, DShuffle exhibits a significant performance gain, as it significantly reduces serialization and GC overhead. However, for workloads with high computational pressure, DShuffle's performance is similar to native Spark, as most of the overhead occurs in the map computation, with a small proportion spent on shuffle. Overall,

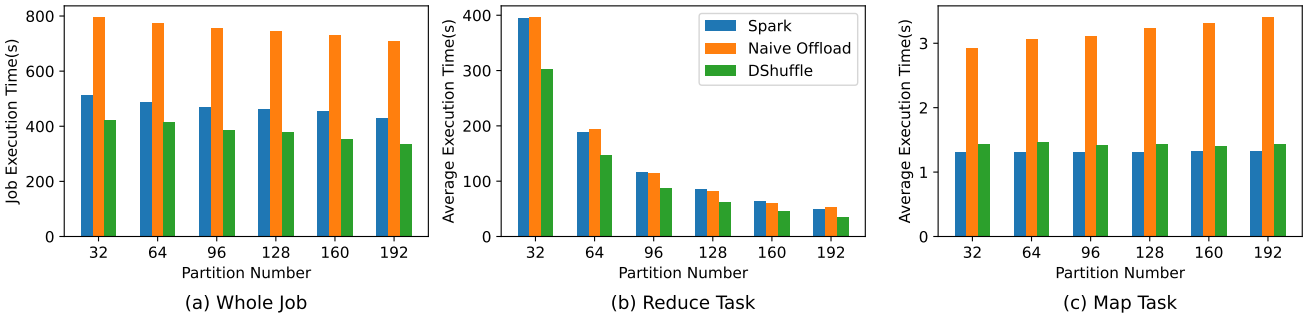


Figure 13: Execution Time of Different Comparison Groups under Different Partition Numbers.

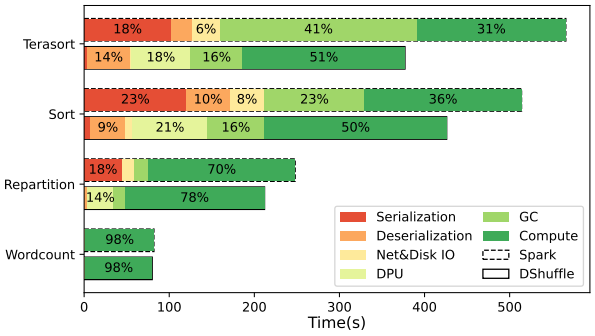


Figure 14: The Time Breakdown of Different HiBench Workloads.

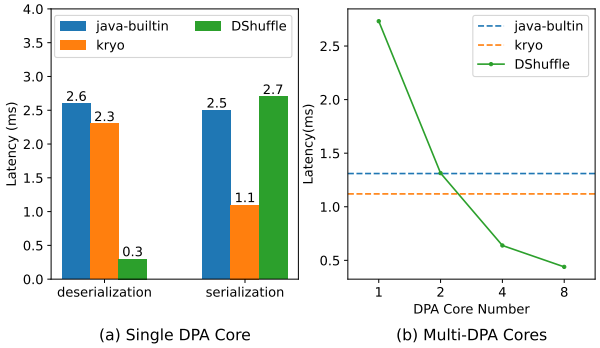


Figure 15: Serialization Latency of Different Serialization Libraries.

DShuffle can effectively accelerate the execution speed of tasks with high shuffle overhead, without affecting the execution efficiency of tasks with high computational overhead.

9.6 Component Efficiency

In this section, we test the efficiency of the key components of the DShuffle, including the DPA-based serializer and the pipeline workers in the DPU.

DPA-based Serializer. To evaluate the DPA-based serializer, we use HiBench to generate a dataset and extract the strings to be serialized from the dataset, forming key-value data with a length of 128, where both the key and value are 4KB each. We used the native Java serialization library, Kryo serialization library, and DShuffle serialization library to process the dataset, recording the execution times for each comparison group. For all tests, we first performed a 5-second warm-up, followed by a 5-second testing period.

Figure 15(a) shows the serialization and deserialization latency for different comparison groups in a single-core scenario. For DShuffle, deserialization runs on the CPU, while serialization runs on the DPA (as explained in §5). It can be observed that the deserialization speed of the DShuffle library is significantly faster than that of the native Java and Kryo libraries. This is primarily because DShuffle's serialization library directly copies the object content to the target buffer address rather than invoking the Java constructor to create the object, thereby reducing the number of interactions with the JVM. For serialization, however, the processing delay is higher than Java native and Kryo due to the lower clock speed of a single DPA core (16 hardware threads) compared to the host CPU.

Nevertheless, the DPU has a large number of DPA cores that can execute concurrently. By leveraging multiple hardware threads for parallel serialization, the DPA serializer can achieve throughput higher than Java native and Kryo. Figure 15(b) illustrates the throughput of the DPA serializer with different numbers of DPA cores, showing that when more than 2 cores (32 hardware threads) are used, the DPA serializer's latency becomes lower than that of Java native and Kryo. While multi-threading on the host can also be used for parallel serialization, it consumes significant CPU resources. DShuffle avoids this issue by offloading the serialization process to the DPA.

Pipeline Workers. To test the maximum throughput of the workers on the DPU, we generated 640GB of key-value data using the HiBench tool, with both keys and values being random strings. We used this key-value data to stress the DPU, evaluating the maximum throughput that DShuffle can achieve and the resource consumption on the DPU with different numbers of workers. The results are shown in Figure 16. It can be observed that as the number of workers increases, the processing bandwidth of DShuffle gradually increases. With 15 workers, the bandwidth reaches approximately 10.53GB/s,

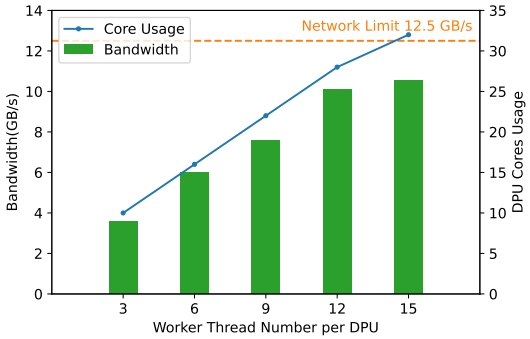


Figure 16: The bandwidth of DShuffle workers with different thread numbers.

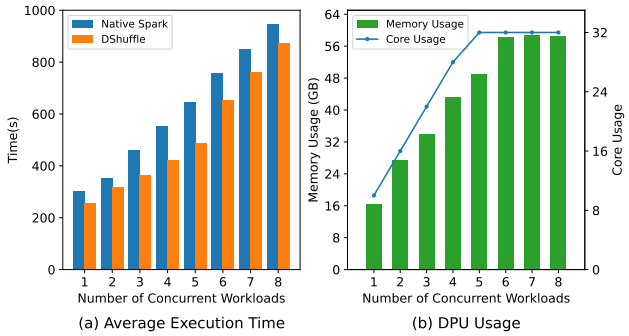


Figure 17: The scalability of DShuffle with different numbers of workloads.

which is close to the limit of a 100Gbps network. At this point, the number of DPU cores consumed by the map and reduce nodes is around 32, which also reaches the maximum number of DPU cores.

9.7 Scalability

In this section, we evaluate the scalability of DShuffle. Specifically, we gradually increase the number of concurrently running Sort workloads. Each 100GB Sort workload is executed by two executors distributed across two physical machines, with each executor provisioned with 6 CPU cores and 12GB of memory.

Figure 17 presents the evaluation results. As shown in Figure 17(a), the average execution time of native Spark increases steadily with the number of concurrent workloads. This trend is primarily attributed to two factors: (1) resource contention among concurrently running Spark executors, particularly during the reduce phase where intensive garbage collection significantly consumes CPU resources and impairs computation efficiency; and (2) network bandwidth contention across executors, which slows down data fetching in the map phase and data exchange during the shuffle phase.

DShuffle mitigates these bottlenecks by offloading shuffle operations, thereby reducing CPU contention and minimizing garbage collection overhead during I/O. As a result, it delivers superior performance under increasing load. The scalability of DShuffle is mainly bounded by the hardware capacity of the DPU. As depicted in Figure 17(b), when the number of concurrent workloads approaches 6, the compute and memory resources of the DPU become saturated. Beyond this point, additional workloads exacerbate resource contention on the DPU, diminishing incremental performance benefits.

Despite this limitation, DShuffle consistently outperforms native Spark and can efficiently support workloads equivalent to a host server with up to 48 CPU cores and 128GB of memory, which is sufficient for the majority of practical applications.

10 Discussion

Comparison with non-DPU solutions. Software-based shuffle optimizations [9, 16, 25, 29, 39, 41, 45] focus primarily on improving I/O process, such as using dedicated shuffle service clusters to schedule subsequent tasks. However, these approaches do not eliminate CPU and memory consumption on the host and may even introduce additional overhead from intermediate servers. Hardware-based shuffle optimizations typically target specific stages of the shuffle pipeline, but lack a comprehensive end-to-end solution. For instance, Spark RDMA [32] leverages high-bandwidth RDMA NICs to accelerate intermediate data transfers but ignores the computational overhead of shuffle. Cereal [22] uses custom hardware to speed up Spark's serialization, while other approaches [8, 37] offload compute-intensive operations—such as joins and merges—to GPUs or FPGAs. However, these solutions often fail to mitigate the I/O overhead associated with intermediate results. By contrast, DShuffle leverages the hardware capabilities of DPUs to provide a holistic optimization of the shuffle process, effectively addressing both computational and I/O bottlenecks.

Difference between DShuffle and SmartShuffle. There are two main differences between DShuffle and SmartShuffle [28]: (1) First, DShuffle is designed to fully offload shuffle operations to DPUs without sacrificing performance. It achieves this through fine-grained pipelined acceleration, which allows for efficient execution of shuffle steps. As a result, DShuffle frees up more host resources for map and reduce computations. In contrast, SmartShuffle, tailored for older-generation DPUs, adopts a dynamic offloading strategy that still relies on CPU and host memory resources. (2) Second, SmartShuffle's evaluation primarily focuses on small datasets (20–50GB) that fit entirely within host memory (64GB), thereby avoiding disk I/O as well as the associated serialization and garbage collection overheads. However, our experiments reveal that these overheads become significant when processing large datasets (several hundred GB) that exceed host memory capacity. DShuffle mitigates these overheads by leveraging DPA-accelerated serialization and DPU-direct spilling, demonstrating strong effectiveness in large-scale data processing.

Portability of DShuffle for other DPU architectures. For SoC-based DPUs (e.g., AMD Pensando [2], Stingray PS1100R [5], Intel IPU E2100 [21], Marvell OCTEON [31]), DShuffle can be easily ported since most of its logic is implemented on general-purpose Arm cores. The hardware serialization logic implemented on the DPA can also be replicated on the host CPU, albeit with some resource consumption, but it remains faster than Java's native serialization library. For FPGA-based DPUs (e.g., AMD Alveo U200 [1]), DShuffle's logic can also be migrated, but fully hardening the implementation presents significant development challenges and longer development cycles. Additionally, FPGAs are costly and better suited for parallel computing tasks. Therefore, we believe SoC-based DPUs are more suitable for shuffle tasks.

11 Related Work

Improving Shuffle I/O Efficiency. For large-scale data analytics workloads, I/O speed is a critical factor affecting shuffle performance [46]. A lot of work has been done to optimize network and disk I/O in the shuffle process. Sailfish [39], AggShuffle [29], and Riffle [25] propose aggregating fragmented intermediate shuffle results into large files to convert random small I/O requests into large sequential I/O, effectively improving the utilization of network and disk bandwidth. Magnet [41], Celeborn [16] and OPS [9] propose directly pushing intermediate results from the map side to the reduce side, allowing network and disk I/O to overlap with the map computation process, thus effectively improving task execution speed. Exoshuffle [30] introduces a shuffle framework that integrates various optimization techniques such as pre-aggregation and push-based shuffle. However, all of these works require reserving host resources or using additional servers to implement aggregation operations. In contrast, DShuffle offloads these processes to the DPU, completely eliminating the dependency on extra hardware resources.

Accelerating Shuffle Serialization. Serialization occupies a significant portion of CPU time during the shuffle process. Many works have focused on accelerating Spark's serialization operations from both software and hardware perspectives. Kryo [14] is a widely used and mature Java serialization library that avoids the extensive string comparison operations in the native Java serialization type reflection mechanism through unique type encoding, effectively improving serialization speed. Skyway [35] extends JVM heap space based on type encoding to achieve lightweight object direct transfer, reducing serialization overhead. ZCOT [45] avoids serialization during the object transfer process through distributed shared memory, but it relies on an additional central metadata server and increases the number of network I/O operations during object transfer. Cereal [22], Zerializer [44], and Cerebros [38] use customized hardware to accelerate serialization operations. In contrast to these approaches, DShuffle uses widely deployed DPUs to offload and accelerate serialization operations without the need for additional hardware modifications, effectively reducing serialization CPU overhead.

Using DPUs to offload and accelerate distributed applications. With the widespread deployment of DPUs in data centers, numerous tasks are leveraging DPUs to offload and accelerate distributed applications, such as distributed file systems [18, 23, 26, 50], distributed key-value stores [13, 19, 40, 43], and disaggregated storage servers [12, 34, 42, 48, 49]. Among these, SmartShuffle [28] is the most relevant to our work. It uses the Stingray DPU to offload shuffle operations. Since its code is not open source, we provide a qualitative analysis here. The goal of SmartShuffle is to offload the shuffle computation to the DPU to reduce the host CPU overhead. However, it focuses only on offloading preprocessing computations and network I/O. For large-scale datasets, serialization and disk I/O also have a significant impact on shuffle performance. DShuffle addresses these two issues by accelerating serialization via DPA and using DPU direct spilling. In addition, SmartShuffle adopts a partial offloading policy, which still requires substantial host resource consumption. In contrast, DShuffle accelerates the shuffle process on the DPU through fine-grained pipeline parallelism, effectively eliminating the consumption of host resources during shuffle operations.

12 Conclusion

In this paper, we demonstrate that shuffle operations introduce significant CPU and IO overhead, often becoming a key performance bottleneck in data-intensive applications. We propose a novel architecture, DShuffle, which leverages DPUs to offload and accelerate shuffle operations. DShuffle introduces three key techniques to enhance the execution efficiency of shuffle operations on DPUs: DPA-accelerated serialization, fine-grained pipelined parallelism, and DPU-direct spilling. We have implemented DShuffle based on Spark and a real DPU hardware platform. The experimental results show that DShuffle effectively reduces CPU consumption, improves IO efficiency, and decreases the completion time of Spark jobs.

Acknowledgments

We thank our shepherd Gyuyeong Kim and anonymous ATC reviewers for their constructive feedback. This work was sponsored by the National Key Research and Development Program of China under Grant No.2023YFB4502701, the National Natural Science Foundation of China under Grant No.62172175, the Shandong Provincial Natural Science Foundation under Grant No.ZR2024LZH004, the China Postdoctoral Science Foundation under Grant No.2024M751011, the Postdoctor Project of Hubei Province under Grant No.2024HBBHCXA027.

References

[1] AMD. Alveo u200 data center accelerator card., 2023.

[2] AMD. Amd pensando infrastructure accelerators., 2023.

[3] Michael Armbrust, Tathagata Das, Joseph Torres, Burak Yavuz, Shixiong Zhu, Reynold Xin, Ali Ghodsi, Ion Stoica, and Matei Zaharia. Structured streaming: A declarative api for real-time applications in apache spark. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, page 601–613, New York, NY, USA, 2018. Association for Computing Machinery.

[4] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, page 1383–1394, New York, NY, USA, 2015. Association for Computing Machinery.

[5] Broadcom. Broadcom stingray smartnic accelerates baidu cloud services., 2020.

[6] Rodrigo Bruno, Luís Picciochi Oliveira, and Paulo Ferreira. Ng2c: pretenuring garbage collection with dynamic generations for hotspot big data applications. *SIGPLAN Not.*, 52(9):2–13, June 2017.

[7] Idan Burstein. Nvidia data center processing unit (DPU) architecture. In *IEEE Hot Chips 33 Symposium, HCS 2021, Palo Alto, CA, USA, August 22-24, 2021*, pages 1–20. IEEE, 2021.

[8] Yu-Ting Chen, Jason Cong, Zhenman Fang, Jie Lei, and Peng Wei. When apache spark meets fpgas: a case study for next-generation dna sequencing acceleration. In *Proceedings of the 8th USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'16, page 64–70, USA, 2016. USENIX Association.

[9] Yuchen Cheng, Chunghsuan Wu, Yanqiang Liu, Rui Ren, Hong Xu, Bin Yang, and Zhengwei Qi. Ops: Optimized shuffle management system for apache spark. In *Proceedings of the 49th International Conference on Parallel Processing*, ICPP '20, New York, NY, USA, 2020. Association for Computing Machinery.

[10] Alibaba Cloud. A detailed explanation about alibaba cloud cipu., 2022.

[11] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In Eric A. Brewer and Peter Chen, editors, *6th Symposium on Operating System Design and Implementation (OSDI 2004), San Francisco, California, USA, December 6-8, 2004*, pages 137–150. USENIX Association, 2004.

[12] Chen Ding, Jian Zhou, Kai Lu, Sicen Li, Yiqin Xiong, Jiguang Wan, and Ling Zhan. D2comp: Efficient offload of lsm-tree compaction with data processing units on disaggregated storage. *ACM Transactions on Architecture and Code Optimization*, 2024.

[13] Chen Ding, Jian Zhou, Jiguang Wan, Yiqin Xiong, Sicen Li, Shuning Chen, Hanyang Liu, Liu Tang, Ling Zhan, Kai Lu, and Peng Xu. Dcomp: Efficient offload of lsm-tree compaction with data processing units. In *Proceedings of the 52nd International Conference on Parallel Processing, ICPP 2023, Salt Lake City, UT, USA, August 7-10, 2023*, pages 233–243. ACM, 2023.

[14] EsdotericSofware. kryo, 2025.

[15] Daniel Firestone, Andrew Putnam, Sambrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian M. Caulfield, Eric S. Chung, Harish Kumar Chandrappa, Somesh Chaturmohta, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, and Albert G. Greenberg. Azure accelerated networking: Smartnics in the public cloud. In Sujata Banerjee and Srinivasan Seshan, editors, *15th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2018, Renton, WA, USA, April 9-11, 2018*, pages 51–66. USENIX Association, 2018.

[16] Apache Software Foundation. Apache celeborn, 2024.

[17] Apache Software Foundation. Apache spark, 2024.

[18] Peter-Jan Gootzen, Jonas Pfefferle, Radu Stoica, and Animesh Trivedi. DPFS: dpu-powered file system virtualization. In Yosef Moatti, Ofer Biran, Yossi Gilad, and Dejan Kostic, editors, *Proceedings of the 16th ACM International Conference on Systems and Storage, SYSTOR 2023, Haifa, Israel, June 5-7, 2023*, pages 1–7. ACM, 2023.

[19] Zerui Guo, Hua Zhang, Chenxingyu Zhao, Yuebin Bai, Michael M. Swift, and Ming Liu. LEED: A low-power, fast persistent key-value store on smartnic jbofs. In Henning Schulzrinne, Vishal Misra, Eddie Kohler, and David A. Maltz, editors, *Proceedings of the ACM SIGCOMM 2023 Conference, ACM SIGCOMM 2023, New York, NY, USA, 10-14 September 2023*, pages 1012–1027. ACM, 2023.

[20] Shengsheng Huang, Jie Huang, Jinquan Dai, Tao Xie, and Bo Huang. The hibench benchmark suite: Characterization of the mapreduce-based data analysis. In *2010 IEEE 26th International Conference on Data Engineering Workshops (ICDEW 2010)*, pages 41–51, 2010.

[21] Intel. Infrastructure processing unit, 2024.

[22] Jaeyoung Jang, Sung Jun Jung, Sunmin Jeong, Jun Heo, Hoon Shin, Tae Jun Ham, and Jae W. Lee. A specialized architecture for object serialization with applications to big data analytics. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 322–334, 2020.

[23] Jongyul Kim, Insu Jang, Waleed Reda, Jaeseong Im, Marco Canini, Dejan Kostic, Youngjin Kwon, Simon Peter, and Emmett Witchel. Linefs: Efficient smartnic offload of a distributed file system with pipeline parallelism. In Robbert van Renesse and Nickolai Zeldovich, editors, *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021*, pages 756–771. ACM, 2021.

[24] Iacovos G. Kolokasis, Giannos Evdorou, Shoaib Akram, Christos Kozanitis, Anastasios Papagiannis, Foivos S. Zakkak, Polyvios Pratikakis, and Angelos Bilas. Teraheap: Reducing memory pressure in managed big data frameworks. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS 2023, page 694–709, New York, NY, USA, 2023. Association for Computing Machinery.

[25] Albert Hyukjae Kwon, David Lazar, Srinivas Devadas, and Bryan Ford. Riffle: An efficient communication system with strong anonymity. 2015.

[26] Qiang Li, Lulu Chen, Xiaoliang Wang, Shuo Huang, Qiao Xiang, Yuanyuan Dong, Wenhui Yao, Minfei Huang, Puyuan Yang, Shanyang Liu, Zhaosheng Zhu, Huayong Wang, Haonan Qiu, Derui Liu, Shaozong Liu, Yujie Zhou, Yaohui Wu, Zhiwu Wu, Shang Gao, Chao Han, Zicheng Luo, Yuchao Shao, Gexiao Tian, Zhongjie Wu, Zheng Cao, Jinbo Wu, Jiwu Shu, Jie Wu, and Jiesheng Wu. Fisc: A large-scale cloud-native-oriented file system. In Ashvin Goel and Dalit Naor, editors, *21st USENIX Conference on File and Storage Technologies, FAST 2023, Santa Clara, CA, USA, February 21-23, 2023*, pages 231–246. USENIX Association, 2023.

[27] Anthony Liguori. The nitro project–next generation aws infrastructure. In *Hot Chips: A Symposium on High Performance Chips*, 2018.

[28] Jiaxin Lin, Tao Ji, Xiangpeng Hao, Hokeun Cha, Yanfang Le, Xiangyao Yu, and Aditya Akella. Towards accelerating data intensive application's shuffle process using smartnics. *Proc. ACM Meas. Anal. Comput. Syst.*, 7(2), May 2023.

[29] Shuhao Liu, Hao Wang, and Baochun Li. Optimizing shuffle in wide-area data analytics. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, pages 560–571, 2017.

[30] Frank Sifei Luan, Stephanie Wang, Samyukta Yagati, Sean Kim, Kenneth Lien, Isaac Ong, Tony Hong, Sangbin Cho, Eric Liang, and Ion Stoica. Exoshuffle: An extensible shuffle architecture. In *Proceedings of the ACM SIGCOMM 2023 Conference*, ACM SIGCOMM '23, page 564–577, New York, NY, USA, 2023. Association for Computing Machinery.

[31] Marvell. Marvell octeon data processing units (dpus)., 2023.

[32] Mellanox. Sparkrdma, 2022.

[33] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, Doris Xin, Reynold Xin, Michael J. Franklin, Reza Zadeh, Matei Zaharia, and Ameet Talwalkar. Mllib: Machine learning in apache spark. *Journal of Machine Learning Research*, 17(34):1–7, 2016.

[34] Jaehong Min, Ming Liu, Tapan Chugh, Chenxingyu Zhao, Andrew Wei, In Hwan Doh, and Arvind Krishnamurthy. Gimbal: enabling multi-tenant storage disaggregation on smartnic jbofs. In Fernando A. Kuipers and Matthew C. Caesar, editors, *ACM SIGCOMM 2021 Conference, Virtual Event, USA, August 23-27, 2021*, pages 106–122. ACM, 2021.

[35] Khanh Nguyen, Lu Fang, Christian Navasca, Guoqing Xu, Brian Demsky, and Shan Lu. Skyway: Connecting managed heaps in distributed big data systems. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, page 56–69, New York, NY, USA, 2018. Association for Computing Machinery.

[36] NVIDIA. Bluefield-3 dpu, 2024.

[37] NVIDIA. spark-rapids, 2025.

[38] Arash Pourhabibi, Mark Sutherland, Alexandros Daglis, and Babak Falsafi. Cerebros: Evading the rpc tax in datacenters. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '21, page 407–420, New York, NY, USA, 2021. Association for Computing Machinery.

[39] Sriram Rao, Raghu Ramakrishnan, Adam Silberstein, Mike Ovsiannikov, and Damian Reeves. Sailfish: a framework for large scale data processing. In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, New York, NY, USA, 2012. Association for Computing Machinery.

[40] Henry N. Schuh, Weihao Liang, Ming Liu, Jacob Nelson, and Arvind Krishnamurthy. Xenic: Smartnic-accelerated distributed transactions. In Robbert van Renesse and Nickolai Zeldovich, editors, *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021*, pages 740–755. ACM, 2021.

[41] Min Shen, Ye Zhou, and Chandni Singh. Magnet: push-based shuffle service for large-scale data processing. *Proc. VLDB Endow.*, 13(12):3382–3395, August 2020.

[42] Junyi Shu, Kun Qian, Ennan Zhai, Xuanzhe Liu, and Xin Jin. Burstable cloud block storage with data processing units. In Ada Gavrilovska and Douglas B. Terry, editors, *18th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2024, Santa Clara, CA, USA, July 10-12, 2024*, pages 783–799. USENIX Association, 2024.

[43] Shangyi Sun, Rui Zhang, Ming Yan, and Jie Wu. SKV: A smartnic-offloaded distributed key-value store. In *IEEE International Conference on Cluster Computing, CLUSTER 2022, Heidelberg, Germany, September 5-8, 2022*, pages 1–11. IEEE, 2022.

[44] Adam Wolnikowski, Stephen Ibanez, Jonathan Stone, Changhoon Kim, Rajit Manohar, and Robert Soulé. Zerializer: towards zero-copy serialization. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '21, page 206–212, New York, NY, USA, 2021. Association for Computing Machinery.

[45] Mingyu Wu, Shuaiwei Wang, Haibo Chen, and Binyu Zang. Zero-Change object transmission for distributed big data analytics. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 137–150, Carlsbad, CA, July 2022. USENIX Association.

[46] Yixin Wu, Xiuqi Huang, Zhongjia Wei, Hang Cheng, Chaohui Xin, Zuzhi Chen, Binbin Chen, Yufei Wu, Hao Wang, Tieying Zhang, et al. Towards resource efficiency: Practical insights into large-scale spark workloads at bytedance. *Proceedings of the VLDB Endowment*, 17(12):3759–3771, 2024.

[47] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In Steven D. Gribble and Dina Katabi, editors, *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2012, San Jose, CA, USA, April 25-27, 2012*, pages 15–28. USENIX Association, 2012.

[48] Jie Zhang, Hongjing Huang, Lingjun Zhu, Shu Ma, Dazhong Rong, Yijun Hou, Mo Sun, Chaojie Gu, Peng Cheng, Chao Shi, and Zeke Wang. Smartds: Middle-tier-centric smartnic enabling application-aware message split for disaggregated block storage. In Yan Solihin and Mark A. Heinrich, editors, *Proceedings of the 50th Annual International Symposium on Computer Architecture, ISCA 2023, Orlando, FL, USA, June 17-21, 2023*, pages 42:1–42:13. ACM, 2023.

[49] Qizhen Zhang, Philip A. Bernstein, Badrish Chandramouli, Jason Hu, and Yiming Zheng. DDS: dpu-optimized disaggregated storage. *Proc. VLDB Endow.*, 17(11):3304–3317, 2024.

[50] Kan Zhong, Zhiwang Yu, Qiao Li, Xianqiang Luo, Linbo Long, Yujian Tan, Ao Ren, and Duo Liu. Dpc: Dpu-accelerated high-performance file system client. In *Proceedings of the 53rd International Conference on Parallel Processing*, pages 63–72, 2024.

Accelerating Model Training on Ascend Chips: An Industrial System for Profiling, Analysis and Optimization

Yuhang Zhou[1], Zibo Wang[1], Zhibin Wang[1]*, Ruyi Zhang[1], Chen Tian[1], Xiaoliang Wang[1], Wanchun Dou[1], Guihai Chen[1], Bingqiang Wang[2], Yonghong Tian[2], Yan Zhang[2], Hui Wang[2], Fuchun Wei[3], Boquan Sun[3], Jingyi Zhang[3], Bin She[3], Teng Su[3], Yifan Yao[3], Chunsheng Li[3], Ziyang Zhang[3], Yaoyuan Wang[3], Bin Zhou[4], Guyue Liu[5]

[1] *State Key Laboratory for Novel Software Technology, Nanjing University*

[2] *Peng Cheng Laboratory* [3] *Huawei* [4] *Shandong University* [5] *Peking University*

Abstract

Training large-scale deep learning (DL) models is a resource-intensive and time-consuming endeavor, yet optimizing training efficiency poses significant challenges. The sporadic performance fluctuations during long training require advanced profiling capabilities. It is not easy to perform comprehensive and accurate bottleneck analysis amidst numerous influencing factors. Selecting effective optimization strategies without proper guidance further complicates the process. This paper shares our practical insights on optimizing training on Huawei Ascend chips based on three years of experience with 135 typical cases. We propose a systematic optimization system, *Hermes*, including a lightweight profiling approach, a hierarchical bottleneck analysis framework, and an optimization advisor. Our real-world experiments demonstrate significant acceleration in training for models like PanGu-α, MobileNetV1, and MoE (Mixture of Experts), with respective speedups of 3.05×, 1.91×, and 1.19×.

1 Introduction

Large-scale deep learning model training is notoriously costly and time-consuming. For example, training a GPT-3 model with 100.8 billion parameters using 3,072 NVIDIA A100 GPUs took 84 days [41]. Similarly, training the BLOOM-176B model with 384 A100 GPUs required approximately 3.5 months [60]. Even the optimized M6 model with 10 trillion parameters took 10 days on 512 V100 GPUs [32].

In optimizing large-scale training, various roles are involved, including developers, deployers, and maintainers. Developers must precisely identify bottlenecks hindering training efficiency to develop targeted optimization strategies. Deployers, on the other hand, are responsible for efficiently deploying models on hardware. However, as models and hardware evolve, bottlenecks also shift, requiring precise identification and appropriate optimization choices. Meanwhile, maintainers confront random performance fluctuations [27] during long training and require the ability to accurately capture the underlying causes to maintain stable training. Overall, there is an urgent need for a comprehensive and systematic approach to profile, analyze, and optimize training.

However, existing solutions for profiling, analysis, and optimization face significant limitations. First, capturing transient performance fluctuations requires continuous monitoring. However, mainstream profiling tools like Pytorch Profiler [50] not only incur substantial overhead but also frequently interrupt training because users must manually adjust the profiling granularity. Second, existing bottleneck analysis relies largely on manual effort, with available tools focusing solely on specific bottlenecks (PRESTO [24] for I/O, R-Pingmesh [34] and Meta's work [19] for RDMA), leaving out a comprehensive analysis. Last, current optimization heavily depends on the expertise of users, and the few available optimization tools are limited to particular scenarios, such as DayDream [70] and dPRO [22] designed for data parallelism. The lack of guidance makes selecting the appropriate optimizations challenging.

In this paper, we share our 3 years of experience in bottleneck optimization of deep learning model training on Ascend chips (NPUs) [31]. From January 2022 to December 2024, we resolved 223 performance anomalies for over 40 clients and summarized 135 key cases. Our practice reveals several critical lessons that have guided our optimization efforts (§3.1). CPU bottlenecks are prevalent but often overlooked. Computation bottlenecks are mainly due to underutilization, which requires joint software and hardware optimization. We also found that overlapping computation and communication may degrade performance due to resource contention. I/O bottlenecks primarily result from CPU-side data reading overhead, and minimizing remote access is crucial. Hardware faults like port flapping and link failures usually cause network issues.

To address the complex and diverse bottlenecks in production, we introduce *Hermes*, a system based on these lessons to guide the optimization of DL model training on Ascend NPUs. *Hermes* supports the common DL frameworks PyTorch [2] and MindSpore [39] and tackles the following problems. (i)

*Corresponding author: wzbwangzhibin@nju.edu.cn

Besides profiling common performance anomalies, how can we effectively detect performance fluctuations in large-scale training, a newly emerging issue? (ii) For the vast and complex bottleneck space in training, how do we achieve systematic and accurate bottleneck analysis? (iii) Finally, how do we select effective optimizations for identified bottlenecks or potential new arising bottlenecks?

Our contributions can be summarized as follows.

Coarse-to-fine profiling (§4.1): We propose a lightweight monitor to collect critical metrics and identify problematic steps and devices, as well as a detailed profiler to further capture the performance of each operator[1] for bottleneck analysis in problematic steps and devices. These two tools work together to provide a complete view of the training process with minimal overhead. Combined with our dynamic profiling mechanism, they enable efficient profiling of performance fluctuations without interrupting the training.

Hierarchical bottleneck analysis framework (§4.2): We observe that the training pipeline operates hierarchically, with different types of bottlenecks occurring in the corresponding components (host, device, and network). Thus, we propose a hierarchical analysis that first considers the inter-operator parallelism and then delves into the intra-operator implementation. The inter-operator analysis verifies the existence of a poor overlap between components and further identifies the bottleneck operators with the critical path when the overlap is sufficient. In the intra-operator analysis, we delve into the execution process of each component (e.g., data processing, dispatch of operators, memory and compute competition, network communication) to identify the root cause.

Bottleneck cause-optimization match (§4.3): Based on 3 years of training optimization experience and 135 typical bottleneck cases, we have matched different bottleneck causes with possible optimization strategies (Table 4). We also developed an automatic optimization tool, mindstudio training tool (mstt) advisor [6]. Based on its built-in optimization rules learned from practice, it can automatically detect the causes of bottlenecks and provide optimization suggestions.

To show how to use *Hermes* for optimization, we present numerous cases, covering optimizations of classic bottlenecks and production tasks (§5). Our real-world cases include optimizing the 100B PanGu-α model [65], deploying MobileNetV1-SSD [35], and addressing performance fluctuations in MoE model training across over 9000 NPUs. These case studies show how to apply optimization based on bottleneck analysis and achieve speedups of 3.05×, 1.91×, and 1.19×, respectively, confirming the effectiveness of *Hermes*.

2 Background and Motivation

In this section, we present the goals and problems faced by different users in training and compare the differences and connections between Ascend NPU and NVIDIA GPU, which together motivate our systematic analysis and optimization.

[1] A basic mathematical operation (e.g., matrix multiply, convolution).

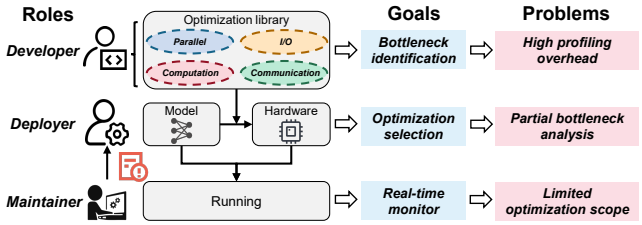


Figure 1: Different roles in model training.

2.1 Roles in Real-World Model Training

As shown in Figure 1, in real-world model training, three roles of users are involved: *developers*, *deployers*, and *maintainers*. Each role has specific responsibilities, challenges, and goals in optimizing training performance. (i) *Developers* are responsible for developing optimizations for different bottlenecks, such as I/O [24], computation [38],communication [46], or parallelism [25, 57, 67]. Therefore, developers must first *identify specific bottlenecks that limit training efficiency*, and then design targeted optimizations to mitigate them. (ii) *Deployers* deploy models on hardware and achieve high training efficiency. However, in practical deployment, users face constantly evolving models and hardware, leading to shifting training bottlenecks and rendering previous optimizations potentially unsuitable. Therefore, deployers must *analyze bottlenecks and select appropriate optimizations* to achieve the expected training performance. (iii) *Maintainers* monitor model training, ensuring smooth progress. However, in large-scale training, performance fluctuations [27] occur frequently and undermine training efficiency. In a 5-month cluster observation, an average of 2.4 incidents were recorded per month. In an 18,000-card cluster, single-card throughput ranged from 10 to 106 TFLOPs/s, with an average of only 85 TFLOPs/s, achieving only 84% of the expected throughput. Therefore, maintainers must *accurately capture performance fluctuations and identify the underlying causes*.

Training optimization usually consists of three key steps: 1) *Profiling* to collect key performance metrics; 2) *Analyzing* to identify training bottlenecks; and 3) *Optimizing* by applying suitable optimizations. Combining the goals of different roles, we summarize the requirements for training optimization.

- *Detailed but lightweight profiling.* Users should be able to collect detailed performance metrics, while profiling should be lightweight to monitor performance fluctuations.
- *Locating various bottlenecks accurately.* Users should be able to accurately identify I/O, computation, communication, and parallel bottlenecks.
- *Efficient optimization suggestions.* Users should be able to choose effective optimizations based on bottleneck causes. Although many technologies have been proposed to meet

these requirements, there are still some significant challenges.

Profiling: To capture transient performance fluctuations in long-term training, maintainers need to continuously monitor and collect detailed performance data for each step/NPU [50]. As shown in §4.1.3, simply profiling the single-step training of an 8B Llama-3 model on 8 NPUs incurs an overhead of 1.77× the original cost. Given the high per-step overhead, the cumulative cost of continuous profiling over thousands of steps in large model training becomes intolerable.

Analysis: Existing bottlenecks analysis are largely *manual*, users must explore all potential bottlenecks and compare the profiling results with the expertise [44]. Even if some auto-analysis can help users identify bottlenecks, they are limited to specific types, such as I/O [24] and RDMA bottlenecks [19, 34]. However, isolated analyses often miss the interdependencies between bottlenecks. For example, synchronization issues in communication bottlenecks may stem from uneven computational loads, while long computation times could result from slow operator compilation or dispatch on the CPU side. Therefore, the challenge lies in comprehensively identifying different types of bottlenecks.

Optimization: Although some tools like DayDream [70] and dPRO [22] can predict optimization effects based on task dependencies, their application is restricted to data parallelism [36]. In most cases, developers select optimizations without knowing the causes of bottlenecks, which can render the proposed optimization less effective. However, due to the complexity of the root causes and the diversity of optimizations, it is also challenging to match effective optimizations to different bottlenecks.

2.2 Comparison of NPU and GPU

Before addressing these challenges, it is necessary to learn the background of Ascend NPUs, especially the similarities and differences with NVIDIA GPUs.

Similar to GPUs, Ascend NPUs also follow the same hierarchical training paradigm. And NCCL, CUDA, and NVLink for GPUs correspond to HCCL (Huawei Collective Communication Library) [10], CANN (Compute Architecture for Neural Networks) [3], and HCCS (Huawei Cache Coherent System) [20] in Ascend, respectively. As shown in Figure 2, the training process on Ascend NPUs can be divided into the following layers: (i) *Application*. Users can train different types of models according to their needs, such as NLP, vision, recommendation, and multimodal. (ii) *Framework*. Ascend can support mainstream deep learning frameworks such as PyTorch, MindSpore, and TensorFlow [1] to convert model code into computation graphs. (iii) *Communication*. HCCL provides parallel communication in distributed training, supporting algorithms such as Ring and Mesh, and primitives such as AllReduce and AlltoAll. (iv) *Platform*. CANN is responsible for converting the computation graph into executable operators on Ascend NPUs, consisting of the operator

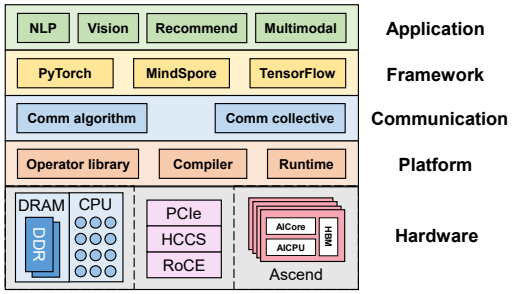


Figure 2: The training paradigm on Ascend NPUs.

library, compiler, and runtime (task scheduling). (v) *Hardware*. Finally, the operators are dispatched to NPUs for execution, while PCIe and HCCS for intra-node communication, and RDMA for inter-node communication. The detailed topology is given in Appendix A.

It is worth noting that there are some differences between the two in terms of chip architecture. In NPUs, AICore [31] is the computing core (similar to CUDA Core and Tensor Core in GPUs), responsible for performing computation-intensive tasks such as matrix and vector operations. However, unlike the GPU, there is another computing unit with slightly lower performance, AICPU [31], which is responsible for non-matrix computing tasks that AICore does not support.

Taking these into account, we can categorize training bottlenecks into two groups: (i) *Hardware-agnostic bottlenecks*. Since the consistent training paradigm and behaviors (data preprocessing, forward and backward computation, gradient update, and parallel strategies), common training bottlenecks, such as low parallelism, high I/O delays, inefficient computation, and slow communication synchronization, are universal regardless of whether the hardware is NVIDIA GPU or Ascend NPU. (ii) *Hardware-specific bottlenecks* refers to performance bottlenecks introduced due to Ascend's architecture characteristics that need to be analyzed separately, such as the inefficient AICPU operators, the overhead of private format for AICore, and the byte alignment requirements in HCCS.

Therefore, different bottleneck types lead to different profiling requirements. Compared to NVIDIA GPUs, the profiling of Ascend NPUs differs in two key aspects: (i) First, considering differences in architecture such as AICore, AICPU, and HCCS, dedicated profiling interfaces for these units are necessary to capture their performance metrics. (ii) Second, while NVIDIA Nsight [44] provides granular profiling interfaces for GPU kernels, network communication, storage, and more, it primarily treats these as isolated bottlenecks. The profiling of Ascend needs a holistic bottleneck analysis and actionable optimization recommendations.

3 Lessons and Insights

In this section, we will share the lessons learned from our practices and insights to achieve a holistic bottleneck analysis.

3.1 Lessons

We have summarized the following lessons from 135 training optimization cases on Ascend NPUs, including I/O, CPU, parallel, computation, and communication bottlenecks.

CPU scheduling bottlenecks dominate in practice but are often overlooked. As shown in Table 4, among all our optimization cases, host bottlenecks account for 45.9%, with CPU bottlenecks reaching 37.0%. This problem exists on both GPUs and NPUs. Although the executing streams of the CPU and device can operate asynchronously, synchronization and data transfers between the CPU and device can severely impact training efficiency. For example, recompilation of the operators can introduce the CPU bottleneck. Specifically, in the Pytorch framework, operators follow this compilation and execution logic. When an operator is invoked, the CPU first checks if it has been pre-compiled. If not, it undergoes just-in-time (JIT) compilation [49] to generate a binary executable dispatched to the device, which will introduce extra overhead.

Underutilization dominates computation bottlenecks. The computation bottleneck analysis reveals a significantly higher proportion of underutilized operators, ranging from 61.48% in 100B PanGu-α to 97.8% in ResNet50. And smaller models tend to exhibit a higher percentage of underutilization. When optimizing underutilized operators, it is essential to consider the characteristics of the Ascend hardware. The first possible cause is the AICPU operators. Due to the input type (not supported by AICore) or an implementation problem, the operators can only be executed on poorly performing AICPUs. In such cases, we should eliminate AICPU operators by converting data types (supported by AICore) or other substitutions. Similarly, replacing operators with better hardware affinity for AICore is another method to improve model FLOPs utilization (MFU). We define "good affinity" as whether the operator's shape meets the hardware's memory layout and alignment requirements (e.g., whether the inner axes of Cube computation are divisible by 256).

Contention between computation and communication. Parallel computing and communication can boost performance, but we observe that their concurrent execution may degrade performance in some situations. The root cause lies in resource contention, particularly the contention for HBM bandwidth. Specifically, communication tasks such as HCCS can access HBM via Direct Memory Access (DMA), whereas the primary computation matrix multiplication (MatMul) is often memory-bound. This leads to HBM bandwidth contention, resulting in lower-than-expected computing performance, often a drop of 20% to 40%. For example, in tensor parallelism, splitting the serial GEMM computation and communication into chunks [13, 27] can enable fine-grained overlap and greatly improve performance. However, overlapping computation and communication chunks can compete for HBM bandwidth, which affects individual performance. Thus, setting fine-grained priorities for HBM access can prevent such contention and further accelerate training.

Minimize remote access to alleviate I/O bottlenecks. I/O bottlenecks stem mainly from high data reading and processing overhead on the CPU side, which cannot overlap with NPU computations. In large-scale training, data reading bottlenecks are particularly prominent due to read/write on remote storage, including other remote access except for training data, such as checking file paths or writing logs. Therefore, unnecessary remote accesses should be minimized, for instance, by reading data only from the local card or DP domain. When unavoidable, like logging, data can be written to local storage first and periodically flushed to remote storage.

Port flapping and link failures dominate network issues. Based on our experience, network problems are mainly due to hardware, such as port flapping and link failures. Moreover, the closer these issues occur to the compute nodes, the greater their impact on the training performance of clusters. Thus, it is essential to promptly clean and replace optical modules and repair links with packet loss or error.

3.2 Insights

Based on these lessons, we have the following insights to achieve a holistic bottleneck analysis.

The training pipeline operates hierarchically, with different types of bottlenecks occurring in the corresponding components. As shown in Figure 2, the framework converts the program into a computational graph, with parallel strategies organized by HCCL, and finally, the graph is converted into executable operators on the hardware through the CANN. Specifically, data preparation is executed by CPU and DRAM storage on the host, while computation operators are mainly executed by AICore and AICPU on the device, and communication operators involve HCCS and RDMA on the network. Therefore, operator bottlenecks typically manifest in specific components, stemming from (i) issues in the operator's implementation and (ii) poor hardware efficiency.

In addition to bottlenecks in operators, the parallelism between operators can also become a training bottleneck. For example, Ascend supports multi-stream parallel execution in Pytorch, with the parallel efficiency between streams directly affecting the training. This includes not only the computation-communication parallel bottlenecks that the well-known parallel strategies (DP, TP, PP) aim to solve but also the parallelism between I/O and computation, as well as CPU and computation. Parallel bottlenecks, in turn, affect the execution of specific I/O, computation, and communication.

Considering parallel bottlenecks between operators and performance bottlenecks within each operator, we propose a hierarchical analysis. Specifically, bottleneck analysis is divided into two layers: inter-operator parallel analysis and intra-operator bottleneck analysis, covering all host, device, and network components. In this way, it provides a structured framework that can systematically decompose the complex

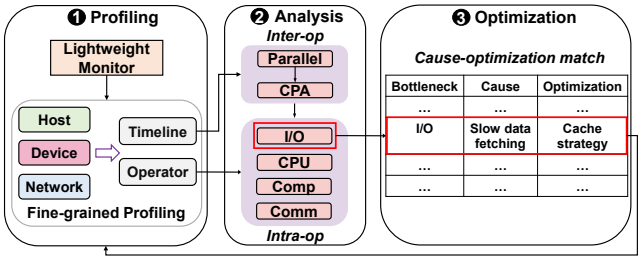


Figure 3: The workflow of *Hermes*.

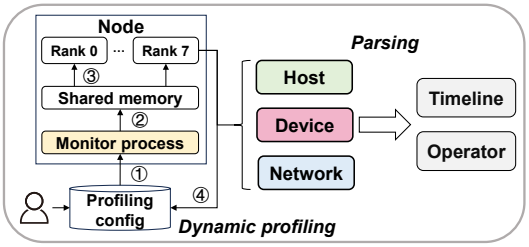


Figure 4: Fine-grained profiling.

training process into different layers and accurately identify bottlenecks in the most common locations.

4 System Design

In this section, we first introduce the main workflow of *Hermes*, then dive into the design of each core module. As shown in Figure 3, the main workflow consists of the following steps.

Coarse-to-fine profiling(§4.1): We observe that it only requires a few key metrics to determine the presence of performance fluctuations. Therefore, we propose a lightweight but fine-grained profiler to identify the problematic steps/devices and capture the necessary metrics.

Hierarchical bottleneck analysis(§4.2): We propose a hierarchical analysis model that first considers the inter-operator parallelism and then delves into the intra-operator implementation. In addition to identifying the bottleneck, we provide a set of mechanisms to trace the root causes.

Experience-guided optimization(§4.3): Based on our experience in bottleneck optimization, we propose an optimization advisor that automatically provides effective recommendations by establishing a match between bottleneck causes and corresponding optimizations.

Users can select the necessary steps based on their goals, e.g., the developer only needs to perform the profiling and analysis, and then directly develop their optimization strategy. The optimization workflow is iterative. After optimizing the current bottleneck, users can repeat the process until the desired performance is achieved.

4.1 Coarse-to-fine Profiling

As illustrated in Figure 3, the profiling module comprises a lightweight monitor and fine-grained profiling. The former collects key metrics in real-time to quickly identify problematic steps and devices with little overhead. Next, fine-grained profiling focuses on the problematic steps and devices, collecting detailed performance data without interrupting training while minimizing storage requirements and parsing times.

4.1.1 Lightweight Monitor

For model developers and deployers, it suffices to profile a selected step offline to analyze persistent slowdowns, as the performance of each training step is consistent. However, continuous online profiling is the only viable approach for maintainers who aim to pinpoint transient performance fluctuations. Therefore, to minimize the overhead of online profiling, this monitor collects only some coarse-grained key metrics during the whole training, including (i) the execution time of each step and (ii) overall performance such as throughput, MFU, and communication bandwidth, which are also the indicator of performance fluctuation.

Cluster Analysis. Based on the monitoring data, we use two steps to identify problematic steps and devices. (i) The first is to compare the execution time of the current step with the historical ones. If the current step is significantly slower, it indicates a performance fluctuation. (ii) We further need to determine which devices are experiencing issues in this step. Considering the various parallel strategies in training (e.g., data parallelism [30], tensor parallelism [41], and pipeline parallelism [23, 40]), the whole cluster is divided into different communication domains (devices involved in the same synchronization). Devices within the same communication domain theoretically have the same execution time. Therefore, we compare the execution times of devices within each communication domain to identify the slowest device, which is often the problem device and requires further profiling.

4.1.2 Fine-grained Profiling

After identifying the problematic steps and devices, we perform fine-grained profiling. We first profile the operator (the smallest execution unit) and collect metrics from various components: (i) *host operator*, including I/O operator queues, and CPU operator metrics related to the Pytorch framework and the CANN operator library; (ii) *computation operator*, computation performance (precision, frequency, FLOPs), and memory access performance (byte and bandwidth) on NPUs; (iii) *communication operator*, transfer time and bandwidth of PCIe, HCCS, and RDMA interconnect. Furthermore, we profile the timeline to illustrate the timing dependencies of operators across different components. However, there are two challenges when fine-grained profiling is applied.

Table 1: Profiling cost comparison of end-to-end training.

Model	#Para	#NPU	Time		Memory		CPU usage	
			Before	After	Before	After	Before	After
Bloom	7B	8	1300 s	1000 s	1650 MB	1800 MB	1%	5%
PanGu	230B	128	120 min	102 min	13.8 GB	16.2 GB	1%	5%

Dynamic profiling. Common profiling tools such as PyTorch Profiler [50] require users to pre-configure monitoring parameters (e.g., via torch.profiler.schedule) before training begins. This intrusive approach requires interrupting training, which incurs prohibitive overhead for large-scale jobs due to checkpoint reloading and warm-up costs. Recent works like Llama-3 [17] have recognized this limitation and enabled on-the-fly configuration changes [55]. On this basis, we propose a more specialized and lightweight dynamic profiling framework with the following features: (i) fully decoupled from the program; (ii) faster synchronization based on shared memory and thread-level updates; (iii) automatically perceives the context to activate profiling and adapts to Ascend-specific metrics. As shown in Figure 4, after modifying the configuration, each node uses a dedicated thread to monitor configuration changes (①) and places it in shared memory (②). Before the next step starts, each rank reads the updated configuration and begins executing profiling (③). Once the step ends, each rank outputs the profiling results for subsequent parsing and configuration adjustments (④). In addition, dynamic profiling can seamlessly integrate with the lightweight monitor, automatically triggering when performance fluctuations are detected.

Efficient parsing. Fine-grained profiling data must go through several parsing steps before being presented to users, including reading, calculating, and visualizing. Considering a cluster with 10,000 NPUs, the profiling data can reach several TBs, potentially requiring days for processing and causing visualization tools such as TensorBoard [21] and Chrome Trace Viewer [16] to slow down or crash. Therefore, we standardize the database format for all metrics, resulting in a 75% reduction in memory usage. We not only leverage multi-threading and high I/O concurrency to accelerate parsing but also utilize idle CPU resources across multiple machines like Canopy [28]. In addition, we can hide the parsing latency within the training phase by transitioning from synchronous to asynchronous parsing. Finally, the MindStudio Insight tool [5] can handle extremely large datasets (more than 10 GB) and reduce latency to under 30 seconds, even with 10,000 cards.

4.1.3 Profiling Cost

Our design balances the overhead associated with the lightweight monitor and fine-grained profiling. When training the 8B Llama-3 model on 8 NPUs, each step takes 85.19 s without profiling and 150.58 s with detailed profiling. With a lightweight monitor, the time is only 85.20 s. After end-to-end validation, our profiling significantly reduced training time compared to existing profiling methods, with minimal impact on resources. As shown in Table 1, when using coarse-to-fine profiling compared to detailed profiling, the end-to-end training time is reduced by 23% for the 7B Bloom model with 8 NPUs and 15% for the 230B PanGu model with 128 NPUs. Meanwhile, memory usage increased by only 9% and 5%, respectively, while CPU usage increased by only 4%.

4.2 Hierarchical Bottleneck Analysis

Hermes employs a hierarchical analysis, which begins with a parallel analysis across different operators and then delves deeper into each operator to identify its bottleneck causes.

4.2.1 Inter-operator Analysis

Inter-operator analysis consists of two steps: (i) identifying parallel bottlenecks among operators by multi-component parallel analysis; (ii) pinpointing bottleneck operators by critical path analysis [29]. The overlap between host, computation, and communication will directly affect the composition of operators on the critical path, causing shifts in the bottleneck operators. Consequently, two steps work together to first resolve parallel bottlenecks before optimizing specific operators (I/O, CPU, computation, or communication).

Multi-component Parallel Analysis. Since components can run in parallel, different types of operators can overlap to accelerate training. Conversely, shorter overlaps reduce component utilization, causing parallel bottlenecks. However, existing works (such as Syndicate [37]) focus solely on the overlap between computation and communication without considering the host operators. We perform a parallel analysis of operators across all components. A typical step time is divided into several segments: *overlap*, where computation and communication/host operators run simultaneously; *non-overlap computation/communication/host*; and *free time*. Therefore, if the overlap ratio is significantly lower than the threshold, it indicates the presence of parallel bottlenecks. Further, for the non-overlapping parts, we should identify the *critical operators* that are truly slowing down the training.

Critical Path Analysis. We extract the critical path [29] composed of operators from different components along the timeline, and accelerate training by reducing the execution time of operators on it. Specifically, we infer a dependency between two operators across different components when the completion time of one operator aligns with the start time of the subsequent operator. We validate this by the inter-operator relationships defined in the computation graph. Accordingly, we develop a traversal algorithm starting from the last operator and searching backward to identify the longest sequence of dependent operators that leads back to the initial operator. In this way, we can pinpoint the critical operators that significantly influence the execution time, reducing the number of operators to be analyzed by more than half. We then rank these critical operators by their execution times to identify the most time-consuming ones for cause analysis.

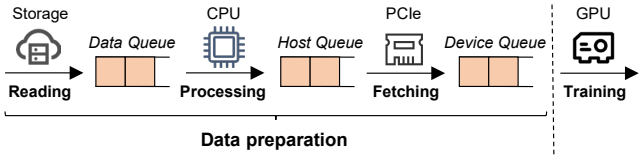


Figure 5: Queue-based I/O analysis.

4.2.2 Intra-operator Analysis

Based on practical experience, we have summarized the causes and detection mechanisms of various types of bottlenecks for critical operators, including host-side I/O and CPU, computation, and communication.

Queue-based I/O Analysis. To provide training data for computation, I/O usually consists of the following stages. As shown in Figure 5, the data first needs to be read from the storage (*data reading*) and then transferred to the CPU for processing (*data processing*). For example, the processing in language models involves tokenization, padding & truncation, and word vectorization [12]. Then, the processed data will queue up on the host side, waiting to be transferred to the devices (*data fetching*). All stages can lead to bottlenecks, and we can use the corresponding queue states to identify them. The Device Queue stores data transferred from the host to the device. If the device queue is not empty, it indicates that there is enough data to continue training, which does not happen in I/O bottlenecks. Conversely, we then move to the Host Queue, which stores processed data. If the host queue is not empty, it indicates a fetching bottleneck, often due to slow transfer between the host and device. Conversely, we need to trace back to the Data Queue, which holds the reading data. If the data queue is not empty, it indicates a processing bottleneck. Otherwise, it points to a reading bottleneck.

Based on Meta's I/O analysis [66], considering that data processing is mainly CPU-based in our practice, we have summarized the following common causes of I/O bottlenecks: (i) *Data reading.* Low disk I/O bandwidth, or restricted network bandwidth to access remote storage. (ii) *Data processing.* Extra overhead from data format conversions, such as with compressed files like zip and tar; no multiprocessing in the dataloader ($num_worker = 1$); other multiprocessing operations that may affect the *dataloader*, like the taskset command [33] for process CPU binding. (iii) *Data fetching.* PCIe transfer from host to device becomes a bottleneck. In this case, consider enabling the *pin_memory* option to transfer data directly to the device through DMA. Faced with complex causes of I/O bottlenecks, *Hermes* only lists potential causes corresponding to the current bottleneck to guide users in inspecting the targeted configurations manually.

CPU Causes Analysis. The CPU handles the compilation of operators and dispatches them to the device, and its performance directly impacts overall training. Based on extensive practice experience, we observe that various causes may result in CPU bottleneck and summarize their reasons in Table 2.

Table 2: CPU bottleneck causes.

Cause	Reason
Operator Compilation	Dynamic shape operators require recompilation
Operator Dispatch	Excessive dispatch or synchronous operations
Garbage Collection	Frequent garbage collection events
CPU Resource Contention	Interference of external CPU process
Environment Configuration	Unsuitable hardware or software configurations

- *Operator Compilation.* Operators must be compiled on the CPU to generate NPU-executable code. If an operator's shape changes, JIT compilation is triggered, adding extra compile time and causing a CPU bottleneck. Thus, *Hermes* detects compilation timeout operators, especially dynamic shape operators, and advises users to replace them or disable JIT compilation.
- *Operator Dispatch.* Dispatching compiled operators from the CPU to the NPU for execution can cause a CPU bottleneck. First, numerous small operators may be dispatched frequently. Thus, *Hermes* provides fusible operator analysis to show the sequence of operators with fusion value. It advises users to fuse these small operators into one, reducing dispatch times. Second, although dispatch is asynchronous, developers might introduce synchronous operations between CPU and NPU streams like *SyncBatchNorm*, *tensor.item*, and *reduce_all*, which slow down the process. Thus, *Hermes* detects the time-consuming synchronous streams and displays their stack, which helps users modify the code to eliminate these sync streams.
- *Garbage Collection (GC).* Python's garbage collection mechanism can automatically detect and free up memory that is no longer in use. However, during large-scale training, the significant fluctuation in memory usage can trigger garbage collection more frequently, which in turn leads to drastic performance degradation. Thus, *Hermes* provides a GC analysis to record the occurrences and durations of abnormal GC events. It suggests that users address GC problems by adjusting the threshold using $gc.set_threshold()$ or disabling GC using $gc.disable()$.
- *CPU Resource Contention.* Other CPU processes unrelated to training may compete for resources, which affects the training performance. In our experience, performance monitoring tools like Prometheus [18] often preempt CPU resources. So, *Hermes* periodically checks the CPU usage of common processes to prevent such contention.
- *Environment Configuration.* Both hardware and software configurations during training can affect performance. Hardware factors like the number of CPU cores, frequency, and computational power may be at play. Potential software issues include OS bugs, high logging levels, memory usage, and inconsistent driver versions. *Hermes* provides self-test scripts to troubleshoot these potential problems.

Computation Cause Analysis. We use the Roofline model [59, 64] to evaluate the computation performance. As shown in Figure 6, the solid line represents the peak perfor-

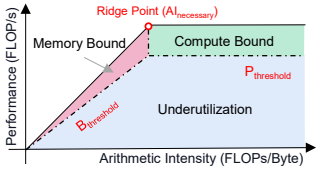


Figure 6: Roofline-based computation Analysis.

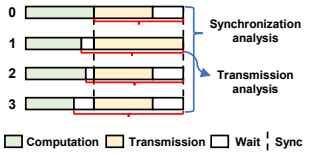


Figure 7: Time breakdown of AllReduce operator.

mance that an operator can achieve with a given arithmetic intensity. The height of the line reflects the arithmetic performance, while the slope of the line reflects the memory bandwidth. We add two dashed lines for the thresholds: the left one marks the bandwidth utilization threshold, and the right one marks the arithmetic utilization threshold. The red area upon the left threshold shows *memory bound*, while the green area upon the right threshold shows *compute bound*. The blue area below the two thresholds represents the *underutilization*. The operator's performance point falling in a specific region determines its computation bottleneck (Appendix B).

Different computation bottlenecks often have different causes. When compute bound, *Hermes* will check whether the node is downclocked over the threshold (often 5%) or has poor intrinsic arithmetic. When memory bound, *Hermes* will suggest operator fusion [43, 68], quantization [38, 56, 63], or using memory optimization techniques like ZeRO [51, 52]. In practice, most operators are underutilized due to various reasons, such as inefficient AICPU operators, operators with suboptimal hardware affinity with Ascend, or private data formats causing extra conversion overhead. For all of these, *Hermes* has the corresponding detection and optimization rules. For a more detailed analysis of operator performance, refer to [69] and the Ascend Optimization Engine [9].

Communication Cause Analysis. As shown in Figure 7, we analyze the causes of communication bottlenecks using the AllReduce operator as an example. Its execution consists of two synchronizations and one transmission. First, it waits for all devices to complete their computations before starting transmission, and synchronization occurs again after all devices have finished transmitting. Consequently, both synchronization and transmission can lead to communication bottlenecks, necessitating a separate analysis of each.

Synchronization analysis examines the wasted time when a device waits for others to complete. For example, in post-transmission synchronization[2] among n devices, we use T_i to denote the transmission time of i-th device. As each device needs to wait for the slowest device to finish its transmission, we use $T_{\max} = \max_{i=1}^{n} T_i$ to denote the slowest transmission time. Accordingly, the overall wait time can be modeled as:

$$T_{\text{wait}} = \sum_{i=1}^{n} (T_{\max} - T_i) = n\left(T_{\max} - \frac{1}{n}\sum_{i=1}^{n} T_i\right) = n(T_{\max} - T_{\text{avg}}) \tag{1}$$

[2]It can be seamlessly extended to synchronization after computation.

Table 3: Communication bottleneck causes.

Cause	Reason
Bandwidth Contention	Intra-node bandwidth contention for parallel computing and communication
RDMA Retransmission	Frequent RDMA timeout retransmission
Small Packet	Underutilization of bandwidth by small packet
Byte Alignment	HCCS data size misalignment
Network Configuration	Switch congestion or UDP port hash collisions

In addition, we have total time, where each device needs $T_{\max}$ including transmission and wait time, by summing them up, we obtain:

$$T_{\text{total}} = n \cdot T_{\max}, \quad R_{\text{wait}} = \frac{T_{\text{wait}}}{T_{\text{total}}} = 1 - \frac{T_{\text{avg}}}{T_{\max}}, \tag{2}$$

where R_{wait} is the ratio of wasted wait time. We set a threshold to determine whether slow synchronization is the cause.

Transmission analysis aims to identify the causes affecting transmission efficiency, including PCIe, HCCS, and RDMA links. *Hermes* first confirms whether the link bandwidth meets the expected threshold. If all links have reached their physical limit, it will advise users to upgrade the network. Otherwise, it will detect all the problems summarized in Table 3 to determine the true cause of the suboptimal bandwidth.

- *Bandwidth Contention.* The transmission of communication operators can be affected by computation operators, particularly due to the bandwidth contention from the intra-node interconnect (HCCS and PCIe). This problem is particularly pronounced when memory-intensive operators, such as MatMul, are executed concurrently. *Hermes* monitors whether the communication operator's intra-node bandwidth falls below the threshold. If so, it will recommend users to re-schedule operators.
- *RDMA Retransmission.* For inter-node transmissions, RDMA retransmissions are a significant cause of bandwidth degradation. *Hermes* compares the RDMA transmission time of communication operators to a retransmission threshold (often 4 seconds). If a retransmission occurs, *Hermes* will advise users to verify the retransmission duration and the network configuration of switches and servers.
- *Small Packet.* Small packet sizes can lead to multiple transmissions, reducing bandwidth utilization. *Hermes* has determined the packet size thresholds at which each link reaches its bandwidth limit in practice. Thus, *Hermes* checks for an excess of packets much smaller than the threshold. If found, it will advise increasing the batch size, gradient fusion [26, 46], or operator fusion.
- *Byte Alignment.* HCCS communication requires memory alignment; the data size must be a multiple of 512 bytes to avoid a drop in the transmission bandwidth. *Hermes* offers a byte alignment analysis for communication operators. It detects operators that do not meet this requirement and suggests adjusting their data size.
- *Network Configuration.* Other network misconfigurations

can also seriously affect communication. *Hermes* summarizes and automatically detects common ones, such as anomalies in HCCL environment variables, congestion from switch settings, or UDP port hash conflicts. Unfortunately, if these analyses fail to reveal the problem, manual intervention is essential.

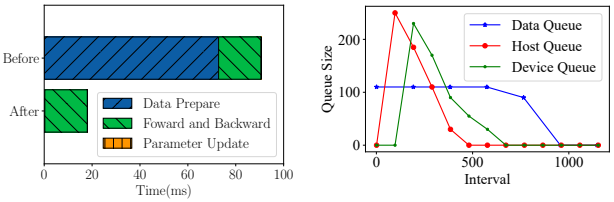
4.3 Bottleneck Cause-Optimization Match

As of December 2024, we have addressed 223 training performance issues for over 40 clients, identifying 135 key cases for analysis and summarization. Among these, both CPU and computation bottlenecks account for over 30%. The former is often caused by system, hardware, or implementation issues and can be overlooked. The latter is mainly due to poor operator implementation, insufficient hardware affinity, and a lack of operator fusion. Based on these cases, we developed the mstt advisor [6]. By inputting profiling data, it will automatically identify the bottleneck through hierarchical bottleneck analysis. As shown in Table 4, the advisor has established many analytical rules for each bottleneck cause and provides all applicable optimization suggestions. Notably, the optimizations provided by the advisor mainly target hardware-agnostic bottlenecks, including parallel, CPU, and most communication bottlenecks. These bottlenecks inherently stem from unified training paradigms and frameworks, allowing the optimizations to be extended to other hardware platforms. However, bottlenecks directly tied to Ascend features, such as AICPU operators, API affinity conflicts, private data formats, and HCCS byte misalignment, cannot be extended.

In summary, using profiling data as input, the advisor can automatically generate a complete analysis report in HTML showing bottlenecks in current training. For example, when detecting AICPU operators whose execution time exceeds the threshold, the advisor will output *AICPU issues* and provide the following optimization suggestions: (i) *Convert the data type to those supported by AICore.* AICPU only executes operators if their data formats are not supported by AICore. Therefore, the advisor lists common operators' supported data formats on both AICPU and AICore. If the target operator is on the list, the advisor will recommend that users reformat the input to AICore compatible type, such as converting the Mul operator's input from int16 to int32. (ii) *Modify code to avoid AICPU operators.* For AICPU operators that cannot convert the format, the advisor suggests replacing them with equivalent operators in AICore or avoiding their use.

5 Case Study

In this section, we provide extensive case studies, including classic bottlenecks and real-world production problems, to demonstrate how *Hermes* guides optimizations.



(a) Time breakdown. (b) Queue size distribution.

Figure 8: I/O analysis and optimization of ResNet50.

5.1 Case Study of Classic Bottlenecks

Here, we present typical optimization cases for I/O, CPU, and communication bottlenecks, explaining the reasons behind the optimizations (more cases are in Appendix C and D).

5.1.1 Case Study of I/O Optimization

The single-card ResNet50 model training is an example of experiencing an I/O bottleneck. As revealed in Figure 8(a), before optimization, the execution time per step is approximately 90 ms, with data preparation taking around 72.8 ms, indicating an I/O bottleneck. Therefore, the advisor performed a queue-based I/O analysis, where the data distribution in different queues is shown in Figure 8(b). As training begins, the data, host, and device queues peak in sequence. Over time, the device and host queues gradually empty, while the data queue remains full. And the advisor concluded that the I/O bottleneck was attributed to slow data processing.

Optimization: Based on the suggestion of the mstt advisor, we found that only one CPU core was active during processing ($num_worker = 1$). To address this, we increased the number of concurrent threads to 12 to better utilize CPU parallelism.

Results: Although the parallelism only increased by 12×, this optimization enables overlap between data processing and subsequent NPU computation, allowing the processing time to be almost completely hidden within the computation time. As shown in Figure 8(a), the explicit data preparation time was reduced to 0.25 ms. The step time was reduced to 18.07 ms, achieving a speedup of 5.34×.

5.1.2 Case Study of CPU Optimization

During GPT-3 training on a single node with 8 NPUs, we observed an average step time of 444.40 ms, accompanied by significant performance fluctuations, with many steps exceeding 1000 ms and a maximum of 3027.6 ms. Cluster analysis identified the 153rd step as a major performance degradation, pinpointing rank-3 NPU as the slowest card. Further bottleneck analysis revealed a long non-overlap host time of 130.86 ms, indicating a severe host bottleneck, primarily due to the excessive execution time of the $aten :: mul$ operator.

Optimization: *Hermes* analyzed the CPU performance and found no issues with operator compilation, dispatch, synchronization, garbage collection, or SW/HW configurations.

Table 4: Cause-optimization match.

Bottleneck	Cause	Optimization	Ratio
Parallel	Poor Parallelism	Auto hybrid parallel [67] / Multi-shard parallel	5.2%
I/O	Slow Data Reading	Increase I/O bandwidth / Remote to local storage	8.9%
	Slow Data Processing	Improve CPU parallelism (num_workers)	
		Avoid compression formats (zip, tar)	
		Cancel the taskset process binding [33]	
	Slow Data Fetching	Cache strategy (pin_memory, data prefetcher) [24, 66]	
CPU	Operator Complication	Replace dynamic shape operators / Disable JIT compilation	37.0%
	Operator Dispatch	Operator fusion [43, 68] / Eliminate synchronization operations	
	Garbage Collection	Disable gc / Increase gc threshold	
	CPU Resources Contention	Disable other CPU process	
	Environment Configuration	Align software versions / Reduce logging level	
Computation	Compute Bound	Avoid decreasing computing frequency / Isolate slow nodes	31.9%
	Memory Bound	Operator fusion [43, 68] / Quantization [38, 56, 63] / ZeRO [51, 52]	
	Underutilization	Eliminate AICPU operators	
		Replace operators with affinity APIs	
		Forbid private format	
Communication	Bandwidth Contention	Avoid bandwidth contention by re-scheduling operators	17.0%
	RDMA Retransmission	Adjust RDMA network configurations of switch and server	
	Small Packet	Increase batch size / Gradient fusion [26, 46] / Operator fusion	
	Byte Alignment	Align HCCS data size	
	Network Configuration	HCCL environment variables / Switch congestion control / UDP hashing collision	

Table 5: Gradient fusion optimization of VGG16.

	Computation (ms)	Non-overlap Communication (ms)	Free (ms)	Total (ms)
Before	54.555	21.760	0.177	76.492
After	52.907	3.594	0.139	56.640

Eventually, CPU usage shows that a Prometheus monitoring program was deployed on this machine, consuming a massive 4000% of CPU resources. This plugin was originally intended for the master node but was mistakenly deployed on the worker node, leading to performance degradation.

Result. After terminating the Prometheus process, the average step time for GPT-3 training decreased to 374.88 ms, with only 4 of 4989 steps exhibiting performance fluctuations exceeding 10%, marking a significant improvement in stability compared to the original 128 steps.

5.1.3 Case Study of Communication Optimization

As shown in Table 5, the step time of VGG16 8-card training is 76.492 ms, with the non-overlap AllReduce operators taking 21.760 ms, resulting in a communication bottleneck. According to the synchronization analysis, the maximum value of R_{wait} is 0.3 and exceeds the threshold. Therefore, the bottleneck is caused by the slow transmission from Rank 7 NPU. This operator only involves HCCS communication, with a bandwidth utilization of only 53%. The average size of HCCS packets is only 12.81 MB, which is too small compared to the 32 MB threshold, resulting in bandwidth underutilization.

Optimization: Faced with this problem, *Hermes* suggests using gradient fusion [11] to improve bandwidth utilization,

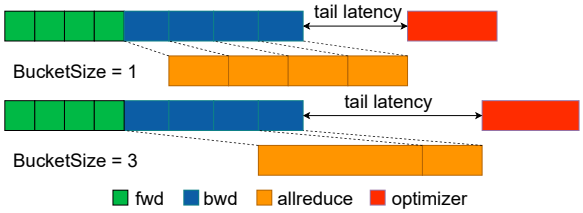


Figure 9: Inappropriate bucket size setting.

which fuses the gradients of multiple backward computations. However, existing frameworks like Pytorch and Horovod [53] only allow for fixed capacity or manual configuration. As shown in Figure 9, undersized capacity (BucketSize=1) cannot improve bandwidth utilization, while oversized capacity (BucketSize=3) can result in long-tail latency. Thus, we further integrate the profiling result to guide our fusion. Specifically, we simulate the AllReduce fusion for gradients of different sizes, which reveals the benefit (time reduction/bandwidth improvement). In addition, we collect execution information to predict the tail latency for a given fusion strategy, including 1) the size of each gradient, 2) the dependencies between operators, and 3) the duration of each computation operator. We model an optimization problem and introduce a greedy forward search algorithm to find a solution by maximizing bandwidth and minimizing tail latency. As shown in Figure 10, we can adaptively fuse the gradients according to the search algorithm. The details can be found in Appendix E.

Results. After this optimization, the non-overlap communication time decreased to 3.594 ms, and the step time subsequently dropped to 56.64 ms with a 1.35× speedup. In addition, we also evaluated common vision, NLP, and recommen-

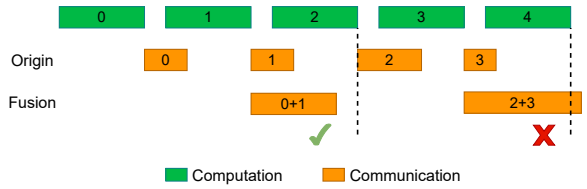
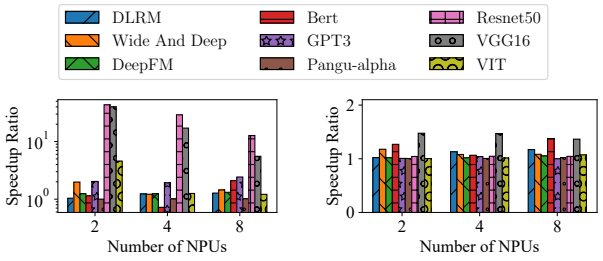


Figure 10: Forward search fusion.



(a) Non-overlap communication. (b) Step time.

Figure 11: Speedup of communication optimization.

dation models. Figure 11 shows the speedup of non-overlap communication and step time before and after optimization. Specifically, *1) if a small packet size is not the root cause pinpointing by Hermes, then the corresponding optimization is limited.* DLRM [42] exhibits a limited improvement in Figure 11(a) because the packet size is large enough. *2) if one is not identified as a bottleneck by Hermes, then the improvement is limited.* For instance, in ResNet50, the optimization brings a significant reduction in non-overlap communication in Figure 11(a), but the step time is not significantly improved in Figure 11(b), since communication is not the bottleneck.

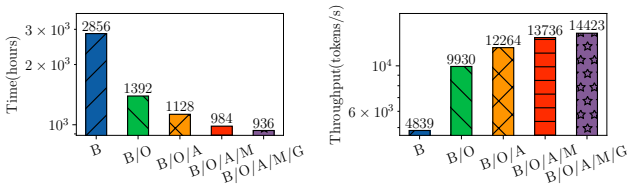
5.2 Case Study of Real-world Model Training

Our practices are conducted on the Ascend training clusters shown in Appendix A. Here, we present three real-world cases in diverse task scenarios, sharing our practical insights.

5.2.1 Iterative Optimization Development for PanGu

We present the iterative analysis and optimization by *Hermes* on the 100B PanGu-α model training. The process consists of three iterations corresponding to bottlenecks: (i) computation, (ii) parallel, and (iii) communication. We can address these bottlenecks by applying optimization recommendations, resulting in the 3.05× speedup in the overall training time.

Baseline: The 100 billion parameters PanGu-α language model is deployed on 128 Ascend NPUs. We selected 50 billion tokens from the original 1.1TB Chinese text corpus from [65] as the training dataset. The baseline configuration is from MindSpore's official SOTA models [4], which employs a re-computation and hybrid parallelism strategy, with data



(a) Overall training time. (b) Overall training throughput.

Figure 12: PanGu-α overall training performance, where B denotes Baseline, O denotes Operator optimization, P denotes Auto hybrid parallelism, M denotes Multi-shard parallelism, and G denotes Gradient fusion.

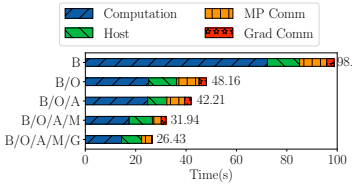


Figure 13: Step time of PanGu-α model.

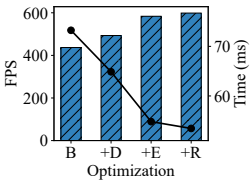


Figure 14: Deploy optimization.

parallelism is 2, tensor parallelism is 8, and pipeline parallelism is 8. The *batch_size* is 512, while the *micro batch_size* is 32. As shown in Figure 12, the total training time is 2856 hours, achieving a throughput of 4839 tokens/s.

Iteration 1: Computation Bottleneck. In the inter-operator analysis, the step time consists of three parts: computation (including overlap), non-overlap communication, and host. Considering the parallel strategy, non-overlap communication can be divided into model parallel communication (Send/Receive) and gradient communication (AllReduce). As shown in Figure 13, the computation time in the baseline accounts for a staggering 73.5% of the iteration time. Thus, even with a satisfactory overlap ratio (8.76%), the computation remains a bottleneck. According to the intra-operator analysis, the top 10 time-consuming computation operators totally consume 59.59 s, accounting for 83.57% of all computation operators. *Therefore, addressing these operators is our top priority.*

Facing the computation bottleneck, we have identified the primary cause of underutilization by Roofline analysis. Following the suggestions from *Hermes*, we have adopted operator optimization, such as replacing high-performance operator, operator fusion [43], and forbidding private formats to accelerate computations. The details can be found in Appendix C. Based on Figure 12, the operator optimizations reduce the step time from 98.01 to 48.16 s and the total time from 2,856 to 1,392 hours, a speedup of 2.05×.

Iteration 2: Parallel Bottleneck. After addressing the computation bottleneck, we observed a reduced computation ratio to a reasonable level of 52.2%. However, we noticed that the overlap between computation and communication only ac-

counted for 2.66% of the step time. The bottleneck has moved to *a significant parallel bottleneck*.

Based on the advice of the advisor, we have adopted two different optimizations to improve parallelism. (i) *Auto hybrid parallelism.* Encouraged by auto parallelism [67], we propose the pipeline parallel simulation to identify a more optimal parallel configuration. Based on the simulation results, we find that incremental *micro batch_size* within a certain range improves throughput, especially at higher pipeline parallelism. Furthermore, if the LayerNorm operator is used for data parallelism, reducing the tensor parallelism can enhance computation performance. Consequently, we adjust the parallel configuration by increasing *micro batch_size* from 32 to 64, setting the pipeline parallelism to 16, and halving the tensor parallelism. By applying the better parallel configuration, the step time decreased from 48.16 to 42.21 s. (ii) *Multi-shard parallelism.* In brief, we divide the task executed within an NPU into two parallel parts: one for computation and the other for communication (details in Appendix D). This approach has improved the overlap ratio and reduced the step time to 31.94 s. Together, the two optimizations accelerated the total time from 1,392 to 984 hours, a speedup of 1.41×.

Iteration 3: Communication Bottleneck. After enhancing the parallelism to 11.19%, we further analyzed the non-overlap communication, including Send, Receive, AllGather, and AllReduce. Among these, AllReduce accounts for 46.83% of the communication time and 22.55% of the step time. Therefore, we must focus on *the communication bottleneck from AllReduce operators*.

Through the communication analysis of AllReduce, *Hermes* has identified that the bottleneck originates from the small packet. Thus, we applied the adaptive gradient fusion in §5.1.3 to address it. As depicted in Figure 12 and Figure 13, the optimization reduced the step time from 31.94 to 26.43 s, with a 1.05× speedup in total time to 936 hours.

5.2.2 MobileNetv1 Deployment Optimization

The MobileNetv1-SSD [35] vision model is applied to object detection in autonomous driving. When migrating this model from an NVIDIA A800 GPU to Ascend NPU directly, users observed significant performance degradation (only achieved 43% of the former), requiring deployment optimization.

To simplify the analysis, we start with single-card training to identify bottlenecks in NPUs. In the baseline, the model already incorporates the hardware-affinity optimizer (NpuFusedSGD [8]) and mixed precision optimization [38], achieving a step time of 73.24 ms and 436.942 FPS (Frame Per Second) as shown in Figure 14. Using *Hermes* for analysis, the results indicate the presence of significant CPU bottlenecks and provide the following suggestions.

Disable JIT compilation. *Hermes* reveals that operator compilation time on the CPU is high, with a total of 762 operators, which consume a staggering 57.862 ms. This issue stems primarily from the dynamic shape [48] of operators. If the input and output shapes of an operator remain constant during training, enabling JIT compilation allows for fusion optimization based on existing operator information, generating more efficient operators online. However, the dynamic shape will invalidate prior operator information and bring re-compilation costs. Thus, the advisor recommends disabling online compilation ($jit_compile = false$) and dispatching the compiled binary operators to reduce time. After this, the step time decreased to 64.87 ms and the FPS increased to 493.332.

Eliminate synchronization. Secondly, a total of 18 synchronization operations between CPU and NPU were detected, and the two slowest operations accounted for 24.136 ms. The frequent synchronizations are often attributed to certain synchronous operators and environment variable settings. Eventually, the advisor shows that it is due to the frequent use of $tensor.item()$. By replacing this function, we achieved a step time of 54.82 ms and an FPS of 583.695.

Replace affinity APIs. The advisor suggests that some APIs can be replaced with more hardware-friendly ones for Ascend. Specifically, tensor reshape and transpose functions in PyTorch like *view*, *permute*, *transpose*, and *contiguous* can be replaced with $npu_confusion_transpose$ [7]. This optimization brings a step time of 53.45 ms and an FPS of 598.716.

Unbind CPU process. When scaled up to 8-NPU training, we observed too high step time in the early steps of each epoch. *Hermes* revealed that there is an I/O bottleneck caused by a slow dataloader with 2857.845 ms, attributed to the taskset command that binds the CPU. This command is used to bind a process to specific CPU cores, reducing context-switching overhead, and improving cache utilization. However, incorrect CPU binding can cause the dataloader and other processes to be bound to the same core, resulting in memory bandwidth contention. Following the advice of the advisor, we unbind the CPU processes, recovering stabilized 8-NPU training performance. The training time decreased from 11.10 to 5.81 ms (1.91× speedup), and the FPS reached 90% of the GPU's performance (5921), meeting the performance goal.

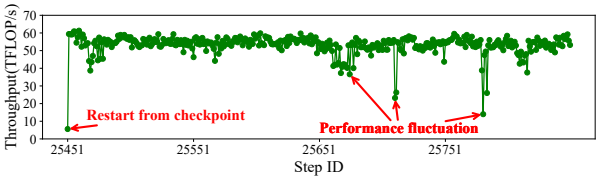
Summary. Table 6 also summarizes the optimization results in deploying other models. Leveraging the optimization suggestions, we have achieved training speedups ranging from 1.08× to 5.34× for these vision, NLP, and recommendation models, demonstrating the effectiveness of *Hermes*.

5.2.3 Performance Fluctuation in Large-scale Training

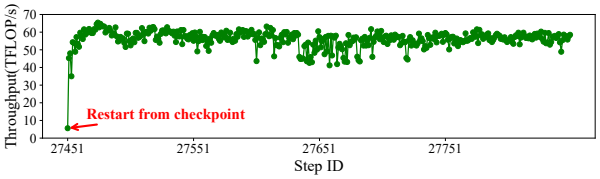
When training a large model, even minor disturbances can lead to significant performance fluctuations, such as the decrease in MFU within MegaScale [27]. However, pinpointing the underlying causes is far from straightforward. Similarly, we frequently encounter performance fluctuations in large-scale training based on Ascend NPUs. Notably, 25% of our analyzed performance fluctuation cases were attributed to Python's garbage collection (GC) [47]. Here, we use a case

Table 6: Model deployment optimization results

Type	Model	Parameter	Optimization Speedup (-: not optimizable)						# of NPUs	Dataset
			I/O	CPU	Para.	Compu.	Comm.	Total		
Vision	ResNet50	25.6M	5.03	-	-	1.02	1.04	5.34	8	ImageNet2012
	VGG16	138.4M	-	-	-	1.08	1.35	1.46		
	MobileNetV1-SSD	4.2M	-	1.37	-	-	-	1.37	1	VOC2012
			1.08	1.91	-	-	-	2.07	8	
NLP	Bert-Large	330M	-	-	-	1.63	1.38	2.49	8	Wiki
	PanGu-α	1.3B	-	-	-	1.18	1.02	1.20		
	GPT3-13B	13B	-	-	1.08	-	-	1.08		
Recommend	DeepFM	16.5M	-	-	-	-	1.08	1.08	8	Criteo
	DLRM	540M	-	-	-	-	1.17	1.17		



(a) Performance before optimization.



(b) Performance after optimization.

Figure 15: Performance fluctuations optimization.

study of 9,000-card MoE model training to share our experience in optimizing garbage collection.

As shown in Figure 15(a), after resuming training from the checkpoint, the throughput gradually increases to about 60 TFLOPs/s. However, during the 400 training steps, there are several significant performance fluctuations, dropping to a low of only 14.0 TFLOPs/s. Each of these drops persists for several steps, severely impacting overall performance. Bottleneck analysis revealed that, within the timeline of the slowest NPU, the non-overlap host time significantly exceeds that of other NPUs. Additionally, *Hermes* offers a GC detection interface to record its occurrences. The results indicated that the number of GC events far exceeded the threshold and performance fluctuations occurred whenever the GC was cleared. If the GCs are not synchronized between each stream, all GCs will execute sequentially within a step, slowing down training.

Although this issue stems from Python's GC mechanism and is currently unresolvable, *Hermes* provides alternative solutions to mitigate it. Firstly, disabling GC (*gc.disable*()) can prevent performance fluctuations but lightly increases training time (based on experience with 7B Llama-2, the cost is about 1%). Secondly, by using the *set_threshold* interface to appropriately increase the threshold, we can reduce the occurrence of fluctuations (with a cost of only 0.02%). Given the relatively low cost of the latter, this approach is commonly adopted in practice. Therefore, we appropriately increased the GC threshold and implemented periodic calls to the *gc.collect*() in the code (typically after saving checkpoints). The results of continued training after optimization, as shown in Figure 15(b), indicate a significant reduction in performance fluctuations. During the same 400 steps, the average throughput improved from 53.33 to 56.23 TFLOPs/s (1.05× speedup).

In addition to software issues, we also detected hardware-induced performance fluctuations when training models in the 18,000-card cluster. Training throughput ranged between 10 and 106 TFLOPs/s, with a high variance of 291, and an average of only 85 TFLOPs/s. Following the recommendations from *Hermes*, we made several optimizations, including switching log writes from remote storage to local storage, repairing four spine-leaf links, and isolating 30 compute nodes (due to port flapping, NPU memory failures, etc.). After these optimizations, training throughput stabilized between 63 and 108 TFLOPs/s, with a variance of 31, and the average improved to 101 TFLOPs/s (1.19× speedup).

6 Limitations and Future Work

We discuss the limitations and future directions of *Hermes*.

More model training techniques. Currently, *Hermes* is mainly used to analyze and optimize the training performance of the Transformer and MoE models. In fact, many new techniques are widely applied in the training of large models such as ChatGPT and Llama, including reinforcement learning (e.g. RLHF [15, 45]) and multimodal techniques [61]. However, *Hermes* currently cannot effectively analyze and optimize the bottlenecks introduced by them. For example, in RLHF, there are potential performance issues in iterative reward modeling, policy optimization, and environment interaction. In multimodal models, cross-modal data alignment (such as image-text fusion) is likely a performance bottleneck. Therefore, we plan to expand *Hermes* to support these emerging

model training technologies.

Diagnosing complex bottlenecks. In complex scenarios, it is still challenging for *Hermes* to diagnose the causes of bottlenecks. For example, in MoE model training, when the AlltoAll communication is severely unbalanced, a simple synchronization analysis is not sufficient to accurately determine whether it is a slow card or slow communication. There are also CPU bottlenecks caused by external environmental changes, which heavily rely on our current troubleshooting experience, currently limited to common monitoring software. As parallel strategies become more complex, including not only DP, PP, and TP but also expert parallelism and context parallelism, how to more accurately identify slow and fast cards becomes a challenge. Therefore, our goal is to improve *Hermes*'s ability to handle these complex situations, provide visualization functions, and more accurately identify bottlenecks.

Evolving cause-optimization mapping. *Hermes* has summarized the causes of bottlenecks and optimization mappings (Table 4) based on 135 actual cases, which already cover most of the bottlenecks and optimizations. However, it is foreseeable that new model training technologies and changing runtime environments will introduce new bottlenecks, and the corresponding new optimization techniques will continue to emerge. The existing mappings of *Hermes* cannot meet the needs and need to be continuously updated. In addition, we plan to move beyond rule-based heuristics by integrating training logs and even LLM-based agents to assist in more accurate root cause analysis and optimization suggestions.

7 Related Work

Performance Profiling. The most convenient way for users to obtain training performance is the built-in profiler of frameworks, such as PyTorch Profiler [50]. These profilers mainly collect information on specific operations within the training framework (e.g., execution time), but their support for hardware profiling is relatively limited (e.g., only GPU utilization and estimated SM efficiency). Furthermore, system-level performance analysis tools, such as Nsight Systems [44] for NVIDIA GPUs, not only provide detailed profiling metrics at the GPU kernel level but also capture the execution and resource utilization of the entire system in real-time, including CPU, GPU, NIC, and storage. However, with the popularity of large-scale training, frequent performance fluctuations [27] and the huge cost of long-term monitoring make it difficult for a single profiling tool to meet demand. In contrast, *Hermes* not only takes into account the architectural features of Ascend but also implements lightweight and online profiling similar to the Llama-3 training [17].

Bottleneck Analysis. After obtaining profiling data, users often need to manually identify bottlenecks based on their own experience. Although there are some automated analysis tools, they only target specific types of bottlenecks. For example, PRESTO [24] only focuses on I/O bottlenecks, SketchDLC [62] only focuses on communication bottlenecks, and some works [19, 34] only focus on RDMA bottlenecks. Even the most powerful Nsight System [44], which provides Expert Systems Analysis to automatically discover performance issues, only covers six rules, such as CUDA synchronous operation and low GPU utilization. In comparison, *Hermes* covers a wide range of common bottlenecks, not only saving a lot of time but also considering the relationships between different bottlenecks to find the real causes.

Training Optimization. Numerous optimizations have been developed to tackle the corresponding training bottlenecks. For example, PRESTO [24] accelerates data preparation by balancing storage capacity and I/O throughput. Mixed precision operation [38] speeds up the computation by reducing the input through compression. Schedulers such as ByteScheduler [46] enhance communication efficiency by controlling tensor splitting and fusion. Hybrid parallelism [57] and auto parallelism [25, 67] achieve a better overlap of computation and communication. When facing bottlenecks, it is also challenging to choose appropriate optimizations. Although tools such as DayDream [70] and dPRO [22] can predict optimization effects based on task dependencies, they are limited to data-parallel scenarios. The Expert Analysis of Nsight System [44] also provides only optimization suggestions for six rules. In contrast, *Hermes* can offer comprehensive optimization suggestions based on the root causes of bottlenecks.

8 Conclusion

In conclusion, *Hermes* addresses the critical issues of performance fluctuation detection, bottleneck analysis, and optimization strategy recommendation. By offering a lightweight profiler, a thorough hierarchical analysis framework, and matching bottleneck causes with optimizations, *Hermes* allows efficient training optimization. Our real-world case studies demonstrate the effectiveness of *Hermes*, leading to substantial improvements in training speed and stability. We believe that our insights from *Hermes* will help the community toward more efficient and stable large-scale model training.

Acknowledgments

The authors thank the shepherd, Yang Wang, and the anonymous reviewers for their insightful comments. This work was supported by the Key Program of the Natural Science Foundation of Jiangsu Province under Grant No. BK20243053, the National Natural Science Foundation of China under Grant Numbers 62325205 and 62172204, the National Natural Science Fund for the Excellent Young Scientists Fund Program (Overseas), the Nanjing University-China Mobile Communications Group Co., Ltd. Joint Institute, and the Postgraduate Research & Practice Innovation Program of Jiangsu Province.

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, page 265–283, USA, 2016. USENIX Association.

[2] Ascend. Ascend pytorch adapter, Jan 2024. https://gitee.com/ascend/pytorch.

[3] Ascend. Cann: Compute architecture for neural networks, 2024. https://www.hiascend.com/en/software/cann.

[4] Ascend. Mindspore modelzoo, 2024. https://gitee.com/mindspore/models/blob/master/README.md.

[5] Ascend. Mindstudio insight, 2024. https://www.hiascend.com/document/detail/en/mindstudio/600/msug/msug_000212.html.

[6] Ascend. Mindstudio training tools, 2024. https://gitee.com/ascend/mstt/tree/master/profiler/msprof_analyze/advisor.

[7] Ascend. torch_npu.npu_confusion_transpose, 2024. https://www.hiascend.com/document/detail/en/canncommercial/700/modeldevpt/ptmigr/ptaoplist_000638.html.

[8] Ascend. torch_npu.optim.npufusedsgd, 2024. https://www.hiascend.com/document/detail/en/canncommercial/700/modeldevpt/ptmigr/ptaoplist_000776.html.

[9] Ascend. Ascend optimization engine. https://www.hiascend.com/document/detail/zh/CANNCommunityEdition/800alpha002/devaids/devtools/aoe/aoeep_16_001.html, dec 2025.

[10] Ascend. Huawei collective communication library, 2025. https://gitee.com/ascend/cann-hccl.

[11] Horovod authors. Tensor fusion. https://horovod.readthedocs.io/en/stable/tensor-fusion_include.html, 2019.

[12] Erik Cambria and Bebo White. Jumping nlp curves: A review of natural language processing research. *IEEE Computational intelligence magazine*, 9(2):48–57, 2014.

[13] Li-Wen Chang, Wenlei Bao, Qi Hou, Chengquan Jiang, Ningxin Zheng, Yinmin Zhong, Xuanrun Zhang, Zuquan Song, Chengji Yao, Ziheng Jiang, Haibin Lin, Xin Jin, and Xin Liu. Flux: Fast software-based communication overlap on gpus through kernel fusion, 2024.

[14] Chang Chen, Xiuhong Li, Qianchao Zhu, Jiangfei Duan, Peng Sun, Xingcheng Zhang, and Chao Yang. Centauri: Enabling efficient scheduling for communication-computation overlap in large model training via communication partitioning. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS '24, page 178–191, New York, NY, USA, 2024. Association for Computing Machinery.

[15] Paul F Christiano, Jan Leike, Tom Brown, Miljan Martic, Shane Legg, and Dario Amodei. Deep reinforcement learning from human preferences. *Advances in neural information processing systems*, 30, 2017.

[16] Chrome. The trace event profiling tool (about:tracing), 2024. https://www.chromium.org/developers/how-tos/trace-event-profiling-tool/.

[17] Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

[18] The Linux Foundation. Prometheus. https://prometheus.io/docs/introduction/overview/, dec 2025.

[19] Adithya Gangidi, Rui Miao, Shengbao Zheng, Sai Jayesh Bondu, Guilherme Goes, Hany Morsy, Rohit Puri, Mohammad Riftadi, Ashmitha Jeevaraj Shetty, Jingyi Yang, et al. Rdma over ethernet for distributed training at meta scale. In *Proceedings of the ACM SIGCOMM 2024 Conference*, pages 57–70, 2024.

[20] Wan-Rong Gao, Jian-Bin Fang, Chun Huang, Chuan-Fu Xu, and Zheng Wang. Wrbench: comparing cache architectures and coherency protocols on armv8 many-core systems. *Journal of Computer Science and Technology*, 38(6):1323–1338, 2023.

[21] Google. Tensorboard, 2024. https://www.tensorflow.org/tensorboard.

[22] Hanpeng Hu, Chenyu Jiang, Yuchen Zhong, Yanghua Peng, Chuan Wu, Yibo Zhu, Haibin Lin, and Chuanxiong Guo. dpro: A generic performance diagnosis and optimization toolkit for expediting distributed dnn training. *Proceedings of Machine Learning and Systems*, 4:623–637, 2022.

[23] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, et al. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *Advances in neural information processing systems*, 32, 2019.

[24] Alexander Isenko, Ruben Mayer, Jeffrey Jedele, and Hans-Arno Jacobsen. Where is my training bottleneck? hidden trade-offs in deep learning preprocessing pipelines. In *Proceedings of the 2022 International Conference on Management of Data*, SIGMOD '22, page 1825–1839, New York, NY, USA, 2022. Association for Computing Machinery.

[25] Zhihao Jia, Matei Zaharia, and Alex Aiken. Beyond data and model parallelism for deep neural networks. In A. Talwalkar, V. Smith, and M. Zaharia, editors, *Proceedings of Machine Learning and Systems*, volume 1, pages 1–13, 2019.

[26] Yimin Jiang, Yibo Zhu, Chang Lan, Bairen Yi, Yong Cui, and Chuanxiong Guo. A unified architecture for accelerating distributed DNN training in heterogeneous GPU/CPU clusters. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 463–479. USENIX Association, November 2020.

[27] Ziheng Jiang, Haibin Lin, Yinmin Zhong, Qi Huang, Yangrui Chen, Zhi Zhang, Yanghua Peng, Xiang Li, Cong Xie, Shibiao Nong, et al. Megascale: Scaling large language model training to more than 10,000 gpus. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 745–760, 2024.

[28] Jonathan Kaldor, Jonathan Mace, Michał Bejda, Edison Gao, Wiktor Kuropatwa, Joe O'Neill, Kian Win Ong, Bill Schaller, Pingjia Shan, Brendan Viscomi, Vinod Venkataraman, Kaushik Veeraraghavan, and Yee Jiun Song. Canopy: An end-to-end performance tracing and analysis system. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 34–50, New York, NY, USA, 2017. Association for Computing Machinery.

[29] James E. Kelley. Critical-path planning and scheduling: Mathematical basis. *Oper. Res.*, 9(3):296–320, jun 1961.

[30] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, and Soumith Chintala. Pytorch distributed: Experiences on accelerating data parallel training. *Proc. VLDB Endow.*, 13(12):3005–3018, aug 2020.

[31] Heng Liao, Jiajin Tu, Jing Xia, Hu Liu, Xiping Zhou, Honghui Yuan, and Yuxing Hu. Ascend: a scalable and unified architecture for ubiquitous deep neural network computing : Industry track paper. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 789–801, 2021.

[32] Junyang Lin, An Yang, Jinze Bai, Chang Zhou, Le Jiang, Xianyan Jia, Ang Wang, Jie Zhang, Yong Li, Wei Lin, Jingren Zhou, and Hongxia Yang. M6-10t: A sharing-delinking paradigm for efficient multi-trillion parameter pretraining, 2021.

[33] Linux. Taskset manual, 2024. https://www.linux.org/docs/man1/taskset.html.

[34] Kefei Liu, Zhuo Jiang, Jiao Zhang, Shixian Guo, Xuan Zhang, Yangyang Bai, Yongbin Dong, Feng Luo, Zhang Zhang, Lei Wang, et al. R-pingmesh: A service-aware roce network monitoring and diagnostic system. In *Proceedings of the ACM SIGCOMM 2024 Conference*, pages 554–567, 2024.

[35] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C Berg. Ssd: Single shot multibox detector. In *Computer Vision–ECCV 2016: 14th European Conference, Amsterdam, The Netherlands, October 11–14, 2016, Proceedings, Part I 14*, pages 21–37. Springer, 2016.

[36] Guandong Lu, Runzhe Chen, Yakai Wang, Yangjie Zhou, Rui Zhang, Zheng Hu, Yanming Miao, Zhifang Cai, Li Li, Jingwen Leng, et al. Distsim: A performance model of large-scale hybrid distributed dnn training. *arXiv preprint arXiv:2306.08423*, 2023.

[37] Kshiteej Mahajan, Ching-Hsiang Chu, Srinivas Sridharan, and Aditya Akella. Better together: Jointly optimizing ML collective scheduling and execution planning using SYNDICATE. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 809–824, Boston, MA, April 2023. USENIX Association.

[38] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, and Hao Wu. Mixed precision training. In *International Conference on Learning Representations*, 2018.

[39] MindSpore. Mindspore, Jan 2024. https://www.mindspore.cn/en.

[40] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, Phillip B. Gibbons, and Matei Zaharia. Pipedream: Generalized pipeline parallelism for dnn training. In *Proceedings of the 27th ACM Symposium on Operating*

Systems Principles, SOSP '19, page 1–15, New York, NY, USA, 2019. Association for Computing Machinery.

[41] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, Amar Phanishayee, and Matei Zaharia. Efficient large-scale language model training on gpu clusters using megatron-lm. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '21, New York, NY, USA, 2021. Association for Computing Machinery.

[42] Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, Xiaodong Wang, Udit Gupta, Carole-Jean Wu, Alisson G. Azzolini, Dmytro Dzhulgakov, Andrey Mallevich, Ilia Cherniavskii, Yinghai Lu, Raghuraman Krishnamoorthi, Ansha Yu, Volodymyr Kondratenko, Stephanie Pereira, Xianjie Chen, Wenlin Chen, Vijay Rao, Bill Jia, Liang Xiong, and Misha Smelyanskiy. Deep learning recommendation model for personalization and recommendation systems, 2019.

[43] Wei Niu, Jiexiong Guan, Yanzhi Wang, Gagan Agrawal, and Bin Ren. Dnnfusion: Accelerating deep neural networks execution with advanced operator fusion. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, page 883–898, New York, NY, USA, 2021. Association for Computing Machinery.

[44] Nvidia. Nvidia nsight systems, 2024. https://developer.nvidia.com/nsight-systems.

[45] Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. Training language models to follow instructions with human feedback. *Advances in neural information processing systems*, 35:27730–27744, 2022.

[46] Yanghua Peng, Yibo Zhu, Yangrui Chen, Yixin Bao, Bairen Yi, Chang Lan, Chuan Wu, and Chuanxiong Guo. A generic communication scheduler for distributed dnn training acceleration. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 16–29, New York, NY, USA, 2019. Association for Computing Machinery.

[47] Python. gc — garbage collector interface, 2024. https://docs.python.org/3/library/gc.html.

[48] Pytorch. Dynamic shapes, 2024. https://pytorch.org/docs/stable/torch.compiler_dynamic_shapes.html.

[49] Pytorch. Jit, 2024. https://residentmario.github.io/pytorch-training-performance-guide/jit.html.

[50] Pytorch. Pytroch profiler, 2024. https://pytorch.org/tutorials/recipes/recipes/profiler_recipe.html.

[51] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. Zero: Memory optimizations toward training trillion parameter models. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '20. IEEE Press, 2020.

[52] Jie Ren, Samyam Rajbhandari, Reza Yazdani Aminabadi, Olatunji Ruwase, Shuangyan Yang, Minjia Zhang, Dong Li, and Yuxiong He. ZeRO-Offload: Democratizing Billion-Scale model training. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 551–564. USENIX Association, July 2021.

[53] Alexander Sergeev and Mike Del Balso. Horovod: fast and easy distributed deep learning in tensorflow. *arXiv preprint arXiv:1802.05799*, 2018.

[54] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

[55] Chunqiang Tang, Thawan Kooburat, Pradeep Venkatachalam, Akshay Chander, Zhe Wen, Aravind Narayanan, Patrick Dowell, and Robert Karl. Holistic configuration management at facebook. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, page 328–343, New York, NY, USA, 2015. Association for Computing Machinery.

[56] Stefan Uhlich, Lukas Mauch, Fabien Cardinaux, Kazuki Yoshiyama, Javier Alonso Garcia, Stephen Tiedemann, Thomas Kemp, and Akira Nakamura. Mixed precision dnns: All you need is a good parametrization. *arXiv preprint arXiv:1905.11452*, 2019.

[57] Minjie Wang, Chien chin Huang, and Jinyang Li. Unifying data, model and hybrid parallelism in deep learning via tensor tiling, 2018.

[58] Shibo Wang, Jinliang Wei, Amit Sabne, Andy Davis, Berkin Ilbeyi, Blake Hechtman, Dehao Chen, Karthik Srinivasa Murthy, Marcello Maggioni, Qiao Zhang, Sameer Kumar, Tongfei Guo, Yuanzhong Xu, and Zongwei Zhou. Overlap communication with dependent computation via decomposition in large deep learning models. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, ASPLOS 2023, page 93–106, New York, NY, USA, 2022. Association for Computing Machinery.

[59] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: An insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4):65–76, apr 2009.

[60] BigScience Workshop, :, Teven Le Scao, Angela Fan, Christopher Akiki, Ellie Pavlick, Suzana Ilić, Daniel Hesslow, Roman Castagné, Alexandra Sasha Luccioni, François Yvon, Matthias Gallé, Jonathan Tow, Alexander M. Rush, Stella Biderman, Albert Webson, Pawan Sasanka Ammanamanchi, Thomas Wang, Benoît Sagot, Niklas Muennighoff, Albert Villanova del Moral, Olatunji Ruwase, Rachel Bawden, Stas Bekman, Angelina McMillan-Major, Iz Beltagy, Huu Nguyen, Lucile Saulnier, Samson Tan, Pedro Ortiz Suarez, Victor Sanh, Hugo Laurençon, Yacine Jernite, Julien Launay, Margaret Mitchell, Colin Raffel, Aaron Gokaslan, Adi Simhi, Aitor Soroa, Alham Fikri Aji, Amit Alfassy, Anna Rogers, Ariel Kreisberg Nitzav, Canwen Xu, Chenghao Mou, Chris Emezue, Christopher Klamm, Colin Leong, Daniel van Strien, David Ifeoluwa Adelani, Dragomir Radev, Eduardo González Ponferrada, Efrat Levkovizh, Ethan Kim, Eyal Bar Natan, Francesco De Toni, Gérard Dupont, Germán Kruszewski, Giada Pistilli, Hady Elsahar, Hamza Benyamina, Hieu Tran, Ian Yu, Idris Abdulmumin, Isaac Johnson, Itziar Gonzalez-Dios, Javier de la Rosa, Jenny Chim, Jesse Dodge, Jian Zhu, Jonathan Chang, Jörg Frohberg, Joseph Tobing, Joydeep Bhattacharjee, Khalid Almubarak, Kimbo Chen, Kyle Lo, Leandro Von Werra, Leon Weber, Long Phan, Loubna Ben allal, Ludovic Tanguy, Manan Dey, Manuel Romero Muñoz, Maraim Masoud, María Grandury, Mario Šaško, Max Huang, Maximin Coavoux, Mayank Singh, Mike Tian-Jian Jiang, Minh Chien Vu, Mohammad A. Jauhar, Mustafa Ghaleb, Nishant Subramani, Nora Kassner, Nurulaqilla Khamis, Olivier Nguyen, Omar Espejel, Ona de Gibert, Paulo Villegas, Peter Henderson, Pierre Colombo, Priscilla Amuok, Quentin Lhoest, Rheza Harliman, Rishi Bommasani, Roberto Luis López, Rui Ribeiro, Salomey Osei, Sampo Pyysalo, Sebastian Nagel, Shamik Bose, Shamsuddeen Hassan Muhammad, Shanya Sharma, Shayne Longpre, Somaieh Nikpoor, Stanislav Silberberg, Suhas Pai, Sydney Zink, Tiago Timponi Torrent, Timo Schick, Tristan Thrush, Valentin Danchev, Vassilina Nikoulina, Veronika Laippala, Violette Lepercq, Vrinda Prabhu, Zaid Alyafeai, Zeerak Talat, Arun Raja, Benjamin Heinzerling, Chenglei Si, Davut Emre Taşar, Elizabeth Salesky, Sabrina J. Mielke, Wilson Y. Lee, Abheesht Sharma, Andrea Santilli, Antoine Chaffin, Arnaud Stiegler, Debajyoti Datta, Eliza Szczechla, Gunjan Chhablani, Han Wang, Harshit Pandey, Hendrik Strobelt, Jason Alan Fries, Jos Rozen, Leo Gao, Lintang Sutawika, M Saiful Bari, Maged S. Al-shaibani, Matteo Manica, Nihal Nayak, Ryan Teehan, Samuel Albanie, Sheng Shen, Srulik Ben-David, Stephen H. Bach, Taewoon Kim, Tali Bers, Thibault Fevry, Trishala Neeraj, Urmish Thakker, Vikas Raunak, Xiangru Tang, Zheng-Xin Yong, Zhiqing Sun, Shaked Brody, Yallow Uri, Hadar Tojarieh, Adam Roberts, Hyung Won Chung, Jaesung Tae, Jason Phang, Ofir Press, Conglong Li, Deepak Narayanan, Hatim Bourfoune, Jared Casper, Jeff Rasley, Max Ryabinin, Mayank Mishra, Minjia Zhang, Mohammad Shoeybi, Myriam Peyrounette, Nicolas Patry, Nouamane Tazi, Omar Sanseviero, Patrick von Platen, Pierre Cornette, Pierre François Lavallée, Rémi Lacroix, Samyam Rajbhandari, Sanchit Gandhi, Shaden Smith, Stéphane Requena, Suraj Patil, Tim Dettmers, Ahmed Baruwa, Amanpreet Singh, Anastasia Cheveleva, Anne-Laure Ligozat, Arjun Subramonian, Aurélie Névéol, Charles Lovering, Dan Garrette, Deepak Tunuguntla, Ehud Reiter, Ekaterina Taktasheva, Ekaterina Voloshina, Eli Bogdanov, Genta Indra Winata, Hailey Schoelkopf, Jan-Christoph Kalo, Jekaterina Novikova, Jessica Zosa Forde, Jordan Clive, Jungo Kasai, Ken Kawamura, Liam Hazan, Marine Carpuat, Miruna Clinciu, Najoung Kim, Newton Cheng, Oleg Serikov, Omer Antverg, Oskar van der Wal, Rui Zhang, Ruochen Zhang, Sebastian Gehrmann, Shachar Mirkin, Shani Pais, Tatiana Shavrina, Thomas Scialom, Tian Yun, Tomasz Limisiewicz, Verena Rieser, Vitaly Protasov, Vladislav Mikhailov, Yada Pruksachatkun, Yonatan Belinkov, Zachary Bamberger, Zdeněk Kasner, Alice Rueda, Amanda Pestana, Amir Feizpour, Ammar Khan, Amy Faranak, Ana Santos, Anthony Hevia, Antigona Unldreaj, Arash Aghagol, Arezoo Abdollahi, Aycha Tammour, Azadeh HajiHosseini, Bahareh Behroozi, Benjamin Ajibade, Bharat Saxena, Carlos Muñoz Ferrandis, Daniel McDuff, Danish Contractor, David Lansky, Davis David, Douwe Kiela, Duong A. Nguyen, Edward Tan, Emi Baylor, Ezinwanne Ozoani, Fatima Mirza, Frankline Ononiwu, Habib Rezanejad, Hessie Jones, Indrani Bhattacharya, Irene Solaiman, Irina Sedenko, Isar Nejadgholi, Jesse Passmore, Josh Seltzer, Julio Bonis Sanz, Livia Dutra, Mairon Samagaio, Maraim Elbadri, Margot Mieskes, Marissa Gerchick, Martha Akinlolu, Michael McKenna, Mike Qiu, Muhammed Ghauri, Mykola Burynok, Nafis Abrar, Nazneen Rajani, Nour Elkott, Nour Fahmy, Olanrewaju Samuel, Ran An, Rasmus Kromann, Ryan Hao, Samira Alizadeh, Sarmad Shubber, Silas Wang, Sourav Roy, Sylvain Viguier, Thanh Le, Tobi Oyebade, Trieu Le, Yoyo Yang, Zach Nguyen, Abhinav Ramesh Kashyap, Alfredo Palasciano, Alison Callahan, Anima Shukla, Antonio Miranda-Escalada, Ayush Singh, Benjamin Beilharz, Bo Wang, Caio Brito, Chenxi Zhou, Chirag Jain, Chuxin Xu, Clémentine Fourrier, Daniel León Periñán, Daniel Molano, Dian Yu, Enrique Manjavacas, Fabio Barth, Florian Fuhrimann, Gabriel Altay,

Giyaseddin Bayrak, Gully Burns, Helena U. Vrabec, Imane Bello, Ishani Dash, Jihyun Kang, John Giorgi, Jonas Golde, Jose David Posada, Karthik Rangasai Sivaraman, Lokesh Bulchandani, Lu Liu, Luisa Shinzato, Madeleine Hahn de Bykhovetz, Maiko Takeuchi, Marc Pàmies, Maria A Castillo, Marianna Nezhurina, Mario Sänger, Matthias Samwald, Michael Cullan, Michael Weinberg, Michiel De Wolf, Mina Mihaljcic, Minna Liu, Moritz Freidank, Myungsun Kang, Natasha Seelam, Nathan Dahlberg, Nicholas Michio Broad, Nikolaus Muellner, Pascale Fung, Patrick Haller, Ramya Chandrasekhar, Renata Eisenberg, Robert Martin, Rodrigo Canalli, Rosaline Su, Ruisi Su, Samuel Cahyawijaya, Samuele Garda, Shlok S Deshmukh, Shubhanshu Mishra, Sid Kiblawi, Simon Ott, Sinee Sang-aroonsiri, Srishti Kumar, Stefan Schweter, Sushil Bharati, Tanmay Laud, Théo Gigant, Tomoya Kainuma, Wojciech Kusa, Yanis Labrak, Yash Shailesh Bajaj, Yash Venkatraman, Yifan Xu, Yingxin Xu, Yu Xu, Zhe Tan, Zhongli Xie, Zifan Ye, Mathilde Bras, Younes Belkada, and Thomas Wolf. Bloom: A 176b-parameter open-access multilingual language model, 2023.

[61] Peng Xu, Xiatian Zhu, and David A Clifton. Multimodal learning with transformers: A survey. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 45(10):12113–12132, 2023.

[62] Yemao Xu, Dezun Dong, Weixia Xu, and Xiangke Liao. Sketchdlc: A sketch on distributed deep learning communication via trace capturing. *ACM Trans. Archit. Code Optim.*, 16(2), apr 2019.

[63] Yuhui Xu, Yongzhuang Wang, Aojun Zhou, Weiyao Lin, and Hongkai Xiong. Deep neural network compression with single and multiple level quantization. In *Proceedings of the AAAI conference on artificial intelligence*, volume 32, 2018.

[64] Charlene Yang, Thorsten Kurth, and Samuel Williams. Hierarchical roofline analysis for gpus: Accelerating performance optimization for the nersc-9 perlmutter system. *Concurrency and Computation: Practice and Experience*, 32, 2019.

[65] Wei Zeng, Xiaozhe Ren, Teng Su, Hui Wang, Yi Liao, Zhiwei Wang, Xin Jiang, ZhenZhang Yang, Kaisheng Wang, Xiaoda Zhang, Chen Li, Ziyan Gong, Yifan Yao, Xinjing Huang, Jun Wang, Jianfeng Yu, Qi Guo, Yue Yu, Yan Zhang, Jin Wang, Hengtao Tao, Dasen Yan, Zexuan Yi, Fang Peng, Fangqing Jiang, Han Zhang, Lingfeng Deng, Yehong Zhang, Zhe Lin, Chao Zhang, Shaojie Zhang, Mingyue Guo, Shanzhi Gu, Gaojun Fan, Yaowei Wang, Xuefeng Jin, Qun Liu, and Yonghong Tian. Pangu: Large-scale autoregressive pretrained chinese language models with auto-parallel computation, 2021.

[66] Mark Zhao, Niket Agarwal, Aarti Basant, Buğra Gedik, Satadru Pan, Mustafa Ozdal, Rakesh Komuravelli, Jerry Pan, Tianshu Bao, Haowei Lu, Sundaram Narayanan, Jack Langman, Kevin Wilfong, Harsha Rastogi, Carole-Jean Wu, Christos Kozyrakis, and Parik Pol. Understanding data storage and ingestion for large-scale deep recommendation model training: Industrial product. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ISCA '22, page 1042–1057, New York, NY, USA, 2022. Association for Computing Machinery.

[67] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P. Xing, Joseph E. Gonzalez, and Ion Stoica. Alpa: Automating inter- and Intra-Operator parallelism for distributed deep learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 559–578, Carlsbad, CA, July 2022. USENIX Association.

[68] Zhen Zheng, Xuanda Yang, Pengzhan Zhao, Guoping Long, Kai Zhu, Feiwen Zhu, Wenyi Zhao, Xiaoyong Liu, Jun Yang, Jidong Zhai, Shuaiwen Leon Song, and Wei Lin. Astitch: enabling a new multi-dimensional optimization space for memory-intensive ML training and inference on modern SIMT architectures. In Babak Falsafi, Michael Ferdman, Shan Lu, and Thomas F. Wenisch, editors, *ASPLOS '22: 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, 28 February 2022 - 4 March 2022*, pages 359–373. ACM, 2022.

[69] Yuhang Zhou, Zhibin Wang, Guyue Liu, Shipeng Li, Xi Lin, Zibo Wang, Yongzhong Wang, Fuchun Wei, Jingyi Zhang, Zhiheng Hu, Yanlin Liu, Chunsheng Li, Ziyang Zhang, Yaoyuan Wang, Bin Zhou, Wanchun Dou, Guihai Chen, and Chen Tian. Squeezing operator performance potential for the ascend architecture. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS '25, page 1156–1171, New York, NY, USA, 2025. Association for Computing Machinery.

[70] Hongyu Zhu, Amar Phanishayee, and Gennady Pekhimenko. Daydream: Accurately estimating the efficacy of optimizations for DNN training. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 337–352. USENIX Association, July 2020.

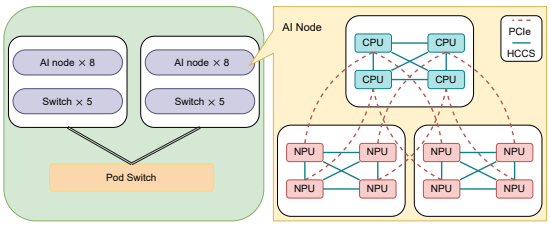


Figure 16: AI computing node topology.

Table 7: Per-node specification.

CPU Compute (TOPs)	1.96 (FP64) / 3.92 (FP32)
NPU Compute (TFLOPs)	2048 (FP16) / 4096 (INT8)
HBM	256 GB
DDR	2 TB
HCCS bandwidth	32 GB/s (uni-directional)
PCIe bandwidth	30 GB/s (uni-directional)
Network bandwidth	8 × 100 Gbps

A Environment details

Figure 16 shows the topology of the Ascend AI computing cluster in our experiments. Each cluster contains 16 NPU Pods, which are interconnected through high-speed aggregation switches. Each NPU pod comprises 8 AI compute nodes interconnected via high-speed access switches. In the single-node topology, every four NPUs are interconnected via HCCS, forming an intra-node cluster with a unidirectional bandwidth of 30 GB/s. The intra-node cluster and NPUs are connected to the CPUs through PCIe 4.0 with a unidirectional bandwidth of 32 GB/s. Each NPU has a 100G RoCE v2 direct outboard network port to establish interconnections with NPUs in other nodes.

Table 7 summarizes the aggregated capabilities of a single node with 4 CPUs and 8 NPUs. The 8 NPUs can provide 2048 FLOPs of half-precision peak computing power, 4096 TOPs of integer-precision peak computing power, and 256 GB of HBM. The four CPUs offer a total of 1.96 TOPs of single-precision peak computing power and 3.92 TOPs of double-precision peak computing power and share 2 TB of DDR4.0 general-purpose memory.

B Roofline Analysis

We define compute Arithmetic Utilization $U_{\text{arithmetic}}$(Memory Bandwidth Utilization U_{memory}) to measure the ratio of actual computational performance P_{actual} (bandwidth B_{actual}) to peak performance P_{peak} (bandwidth B_{peak}). A utilization threshold $U_{\text{threshold}}$ is first established based on practical experience to delineate underutilization. This leads to the establishment of performance threshold $P_{\text{threshold}} = U_{\text{threshold}} * P_{\text{peak}}$ and bandwidth threshold $B_{\text{threshold}} = U_{\text{threshold}} * B_{\text{peak}}$. We then introduce the following classification function.

$$f(P_{\text{actual}}, B_{\text{actual}}) = \begin{cases} \text{memory bound}, & \text{if } AI_{\text{actual}} \le AI_{\text{necessary}} \\ & \text{and } B_{\text{actual}} \ge B_{\text{threshold}}, \\ \text{compute bound}, & \text{if } AI_{\text{actual}} > AI_{\text{necessary}} \\ & \text{and } P_{\text{actual}} \ge P_{\text{threshold}}, \\ \text{underutilization}, & \text{otherwise.} \end{cases} \tag{3}$$

Following the roofline model, we use $AI_{\text{necessary}} = P_{\text{peak}}/B_{\text{peak}}$ in ridge point to classify the *memory bound* and *compute bound*. Operators that fail to meet either memory or compute thresholds fall into the underutilization category.

Referring back to Figure 6, the height of the line reflects the compute performance, while the slope of the line reflects the memory bandwidth. To visualize thresholds, two dashed lines can be added: the left line for $B_{\text{threshold}}$ and the right line for $P_{\text{threshold}}$. The area between these two threshold lines is shaded blue in the figure, marking the *underutilization* zone.

C Case Study of Computation Optimization

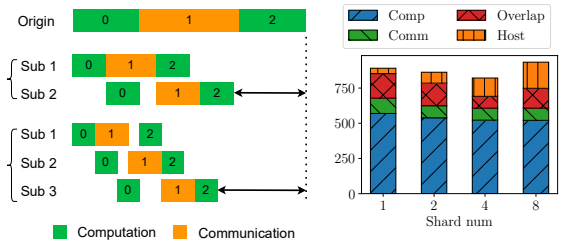
In 100B PanGu-α, the top 10 time-consuming computation operators mainly include MatMul/BatchMatMul, format conversion operators (TransData and Cast), and element-wise operators (Mul and Add). The roofline analysis shows that 61.48% of the bottlenecks were underutilized, while 34.02% were memory-bound, with compute-bound only 4.50%.

Optimization: For the underutilized operators, we chose the following optimization. 1) *FastGeLU.* We replace the inefficient GeLU operator with FastGeLU, which is a *high-performance operator*. 2) *Operator Fusion.* We fuse inefficient element-wise operators such as Mul and Add into the LayerNorm operator, which is suitable for parallel acceleration. We can also perform fusion on time-consuming MatMul operators, such as FastGeLU with MatMul or BatchMatMul with Add. 3) *Format Conversion:* Ascend's matrix compute unit (Cube [31]) only supports private formats (Appendix F), while tensor inputs often do not, leading to frequent format conversions during training. So we defaulted to using the private format to reduce the need for format conversion operators.

Results. After the optimizations, the computation time per step decreased from 72.31 s to 25.16 s. Specifically, regarding the distribution of computation bottlenecks, the ratio of underutilized operators decreased to 46.32%, and most of them shifted to the memory-bound.

D Case Study of Parallel Optimization

Taking the GPT3-13B model with 8-card 8-way tensor parallel training as an example, as shown in Figure 17(a), the single step time is 890.92 ms, 'Host', 'Comp' and 'Comm'



(a) Multi-shard parallelism. (b) Results on GPT3.

Figure 17: New optimization for parallel bottleneck.

are non-overlapping parts. The multi-component parallel analysis shows that the overlap time between computation and communication is only 38.149 ms, suggesting a parallel bottleneck.

Optimization: Inspired by finer-grained parallelism [14, 58], we propose the multi-shard parallel strategy to divide the computation and communication tasks within the same NPU, further enhancing the overlap. According to Figure 17(a), the computation and communication tasks are sequential and independently executed in the origin training, resulting in only one component being utilized while the other is idle. Our optimization tries to parallelize the computation and communication components by splitting the task along the *batch_size* dimension into two shorter tasks. Within the same device, one task focuses on communication, while the other performs computation. Moreover, increasing the number of partitions can theoretically achieve higher parallelism, but it also introduces additional overhead, which is a trade-off. Notice that multi-shard parallelism can be used as a standalone parallel strategy or combined with others like tensor parallelism.

Results: During tensor parallelism, AllReduce (AllGather and ReduceScatter) communication is typically involved before and after the computation of the self-attention and linear layers [54], which is consistent with multi-shard parallelism. As shown in Figure 17(a), we evaluated with shard numbers 2, 4, and 8. When num=2 or 4, the overlap time ratio increases, resulting in more overlapped computation and communication, and the step time decreases to 860.46 ms (1.04×) and 821.375 ms (1.08×), respectively. Surprisingly, when num=8, although the overlap time continues to increase, the step time increases instead of decreasing, reaching 933.968 ms. The reasons include the total computation time exceeding the computation time of the original task after partitioning, and the additional communication overhead. Therefore, we should better keep the number of partitions within a small range.

E Adaptive Gradient Fusion

Actually, the dual problem of fusion is how to segment a sequence of gradient synchronization operators. The origin is equal to that each operator constitutes a segment. Formally, we have a sequence of operator $O = [o_1, o_2, \ldots, o_n]$. For the operator o_i, we use v_i to denote the volume of gradients. As the gradients for the synchronization operator o_i will be generated from the corresponding computation operator, we use $d_{compute,i}$ to denote the corresponding duration time of computation. The segmentation is defined as $S = [\mathcal{S}_1, \mathcal{S}_2, \ldots, \mathcal{S}_n]$ where $\mathcal{S}_i = [o_l, o_{l+1}, o_r]$ denotes a segment of the operator sequence. The optimization problem can be formulated as:

$$\begin{aligned} \arg\min_{S} \quad & e_{comm,m} \\ \text{s.t.} \quad & \mathcal{S}_1 \cup \mathcal{S}_2 \cup \cdots \cup \mathcal{S}_m = O, \\ & \forall 1 \le i < j \le m, \quad \mathcal{S}_i \cap \mathcal{S}_j = \emptyset, \end{aligned} \tag{4}$$

where $e_{comm,m}$ denotes that the communication end time of m-th fused gradients. The end time of i-th fused gradients can be obtained from the following recursive equation:

$$\begin{aligned} e_{compute,1} &= \sum_{o_i \in \mathcal{S}_1} d_{compute,i} \\ e_{comm,1} &= e_{compute,1} + f\left(\sum_{o_i \in \mathcal{S}_1} v_i\right), \\ e_{compute,k} &= e_{compute,k-1} + \sum_{o_i \in \mathcal{S}_k} d_{compute,i} \\ e_{comm,k} &= \max\left(e_{compute,k}, e_{comm,k-1}\right) + f\left(\sum_{o_i \in \mathcal{S}_k} v_i\right), \end{aligned} \tag{5}$$

where $f : \mathbb{R} \to \mathbb{R}$ maps volume to communication time according to our benchmark.

Forward search fusion is illustrated in Figure 10. Once the gradient computation is completed, the current AllReduce operator starts executing. At the same time, the next gradient computation continues until it is completed, triggering the next AllReduce. The algorithm aims to merge as many AllReduce operators as possible without increasing the overall communication time. The process begins by attempting to fuse the first and second AllReduce operators. It calculates the end time of the fused AllReduce and compares it with the start time of the next one. If the end time does not exceed the start time, the fusion is successful, and the fused operator is included in the strategy. Conversely, when attempting to fuse the third and fourth AllReduce operators, the end time of the fused operator exceeds the start time of the next AllReduce. Therefore, the fusion is abandoned, and the current operator is added to the strategy. This process is repeated, and ultimately, all the fused AllReduce operators are obtained.

F Private Data format

ND and NZ formats are common data formats in Ascend chips. The former is the arbitrary format that supports ex-officio Tensor data storage. It defaults to the NCHW format in Pytorch and MindSpore, while it defaults to the NHWC

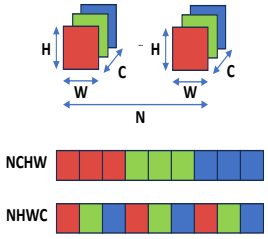


Figure 18: ND format.

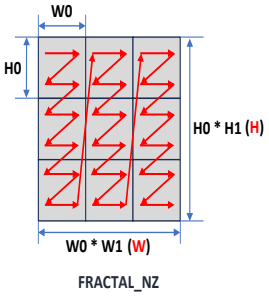


Figure 19: NZ format.

format in Tensorflow, where N is the batch size, C is the number of input channels, and H and W are the sizes of the input feature maps. Figure 18 shows a schematic of this data format and the mapping of its logical arrangement to its physical arrangement in memory.

The NZ format (*FRACTAL_NZ*) is an Ascend-specific fractal format, such as the data storage of featuremap. The data format, as shown in Figure 19 of the output matrix, is $NW1H1H0W0$ during cube unit calculation. The whole matrix is divided into $(H1 * W1)$ fractals, which are arranged according to column-major and have the shape of N, and $(H0 * W0)$ elements inside each fractal, which are arranged according to row-major and have the shape of z. Considering the format of the data layout, the $NW1H1H0W0$ data format is called the Nz format, where H0 and W0 indicate the size of a fractal.

CAFault: Enhance Fault Injection Technique in Practical Distributed Systems via Abundant Fault-Dependent Configurations.

Yuanliang Chen
KLISS, BNRist, School of Software
Tsinghua University, China

Fuchen Ma *
KLISS, BNRist, School of Software
Tsinghua University, China

Yuanhang Zhou
KLISS, BNRist, School of Software
Tsinghua University, China

Zhen Yan
KLISS, BNRist, School of Software
Tsinghua University, China

Yu Jiang *
KLISS, BNRist, School of Software
Tsinghua University, China

Abstract

To ensure high reliability and availability, distributed systems are designed to be resilient to various faults in complex environments. Fault injection techniques are commonly used to test whether a distributed system can correctly handle different potential faults. However, existing fault injection testing is typically performed under a fixed default configuration, overlooking the impact of varying configurations (which can differ in real-world applications) on testing execution paths. This results in many vulnerabilities being overlooked.

In this work, we introduce CAFault (**C**onfiguration **A**ware **Fault**), a general testing framework for enhancing existing fault injection techniques via abundant fault-dependent configurations. Considering the vast combinatorial search space between fault inputs and configuration inputs, CAFault first constructs a fault-dependent model (FDModel) to prune the test input space and generate high-quality configurations. Second, to effectively explore the fault input space under each configuration, CAFault introduces fault-handling guided fuzzing, which constantly detects bugs hidden in deep paths. We implemented and evaluated CAFault on four widely used distributed systems, including HDFS, ZooKeeper, MySQL-Cluster, and IPFS. Compared with the state-of-the-art fault injection tools CrashFuzz, Mallory, and Chronos, CAFault covers 31.5%, 29.3%, and 81.5% more fault tolerance logic. Furthermore, CAFault has detected 16 serious previously unknown bugs.

1 Introduction

In a distributed environment, various faults can occur, such as network delays, packet loss, hardware failures, etc. To handle these faults and enhance system availability, distributed systems are designed with different kinds of fault tolerance mechanisms, such as replication [9], consensus protocols [11], and failover strategies [47]. In a practical distributed system, fault tolerance mechanisms guarantee one of the most fundamental properties called availability [5], ensuring that distributed systems can continuously provide functional services without interruption, even when faults occur.

Due to the complexity of fault tolerance mechanisms in distributed systems, it is challenging to avoid incorrect handling or implementation bugs in these fault tolerance logics. These bugs are referred to as **fault handling bugs**. Since fault tolerance mechanisms play a critical role in distributed systems, such bugs can have severe consequences, leading to service unavailability, data loss, and even compromising system security. To detect these bugs, several fault injection tools [6,17,46,56,59] have been developed to simulate and inject various faults into systems. These tools have successfully uncovered numerous bugs to date.

However, existing fault injection tools typically test distributed systems in fixed default configurations, ignoring how different configurations can impact the execution of fault tolerance mechanisms. In real-world applications, configurations can vary significantly, and existing tools overlook many execution paths, missing hidden bugs that might arise under different configurations. For example, in MySQL-Cluster [12], the default configuration for the data consensus mechanism is set to 'RAFT' [53], which ensures consistency across nodes. However, MySQL-Cluster also offers different consensus options, such as 'Paxos' [27] and 'Quorum-based' [35] mechanisms. The choice of consensus configuration can significantly impact the fault tolerance logic, such as how the system handles network partitions, node failures, and data consistency during recovery, potentially leading to different behaviors and vulnerabilities depending on the selected configuration.

To improve fault handling bug detection in distributed systems, an intuitive approach is to conduct fault injection testing under different configurations. However, since distributed systems typically have numerous configuration parameters, the configuration input space is often large [68], denoted as M. Similarly, the fault injection parameters, such as the time, location, and sequence length of the fault, create a vast input space [22], denoted as N. Consequently, exploring the combination of these two input spaces results in a total search space of $M * N$, which is significantly enormous.

*Fuchen Ma and Yu Jiang are the corresponding authors.

To effectively explore such a huge input space, there are two main challenges: (1) **The first challenge** is automatically analyzing the implicit dependencies between fault and configuration inputs to prune the configuration input space M. Many configuration items in distributed systems, such as log level settings, are unrelated to fault handling and can be excluded from testing. However, the vast number of configuration options and their complex semantics make it difficult to automatically identify these dependencies. Existing configuration testing tools [41, 42, 61, 65] rely on manual modeling to construct dependencies between configuration and workload inputs, which is labor-intensive and lacks scalability. (2) **The second challenge** lies in optimizing and pruning the exploration of the fault injection input space N. In each configuration input, fault injection can occur at any time and at any execution location, making the exploration space N infinite. Existing state-of-the-art (SOTA) fault injection tools typically use code coverage [17] or runtime behavior-guided [46] fuzzing strategies to explore fault input generation. However, these methods are time-consuming and tend to explore code unrelated to fault-handling logic, which result in ignoring bugs hidden in deep paths, leading to ineffective testing.

To address these challenges, we propose CAFault, a comprehensive testing framework aimed at detecting fault handling bugs in distributed systems across different configuration setups. First, CAFault introduces an FDModel that automatically identifies implicit dependencies between fault and configuration inputs, helping to prune and optimize the configuration input space. FDModel monitors the runtime behavior of the distributed system under test, observing which configuration items influence fault handling logic and how faults impact the system status. By tracking coverage changes and correlating them with configuration changes, the FDModel is dynamically constructed and updated throughout the testing process. Second, to effectively explore the fault input space and uncover bugs hidden in deep paths for each configuration input, CAFault employs a fault-handling guided fuzzing strategy. It performs static analysis to identify fault-handling code in the system and then leverages coverage data from this identified code to guide fault input generation during fuzzing. In this way, CAFault continuously conducts efficient fault injection testing with abundant fault-dependent configurations, effectively exercising fault tolerance mechanisms and detecting bugs in distributed systems.

We implemented CAFault and evaluated its effectiveness on four widely-used distributed systems: ZooKeeper [36], MySQL-Cluster [12], Hadoop Distributed File System (HDFS) [4], and IPFS [32]. In comparison to other state-of-art fault injection tools, e.g., CrashFuzz [17], Mallory [46], and Chronos [6], CAFault excelled in exposing more bugs and covering 31.5%, 29.3%, and 81.5% more fault tolerance logic, respectively. CAFault found 16 bugs in total, with 4 in ZooKeeper, 6 in MySQL-Cluster, 3 in HDFS, and 3 in IPFS. We also enhanced existing fault injection tools with the FDModel in CAFault. The results show that FDModel significantly improves testing performance, detects 6 more bugs, and covers 25% more fault tolerance logic on average.

In summary, we make three key contributions:

- We propose the FDModel that automatically analyzes implicit dependencies between fault and configuration inputs. Enhanced with this model, the performance of existing fault injection techniques has been significantly improved.
- We introduce fault-handling guided fuzzing to prune fault input space and effectively exercise fault tolerance mechanisms in distributed systems.
- We implement and evaluate CAFault on four widely used distributed systems. We will open-source CAFault[1].

2 Background of Fault Tolerance Mechanism

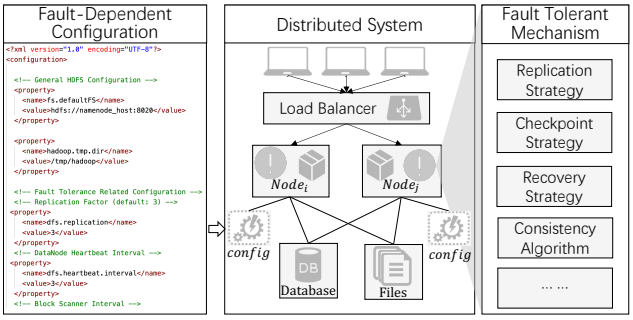


Figure 1: Typical fault tolerance mechanisms in Distributed Systems, that are affected by various configuration setups.

Fault tolerance mechanisms in distributed systems are essential for ensuring the reliability and availability of services in the presence of failures. Due to the complexity of distributed interactions, distributed systems are inherently more vulnerable to faults than standalone systems, with potential failures including network issues, hardware malfunctions, or software bugs. Fault tolerance mechanisms aim to detect, tolerate, and recover from these failures to ensure continuous operation. As shown in Figure 1, Mainstream fault tolerance techniques primarily include redundancy, checkpointing, recovery, and consistency strategies [38]. Replication strategy [26] involves adding extra resources to improve system reliability and fault tolerance, including hardware (e.g., redundant servers), data (e.g., replication across nodes), and computational replication (e.g., task replication or load balancing). Checkpointing [14] is a technique where the system periodically saves its state to allow recovery to a known good state after a failure. Recovery mechanism [60] can be cold recovery, where the system restarts from the last checkpoint and reloads data, or hot recovery, where the system resumes from a more recent state to minimize downtime and data loss. Consistency algorithms [19], such as Paxos and Raft, ensure that distributed systems maintain data consistency across nodes

[1] CAFault:https://anonymous.4open.science/r/CAFault-43A8/

despite faults, addressing issues like network partitions and concurrent updates that could cause data inconsistencies.

Fault-Dependent Configuration: System configuration is crucial to ensure that a distributed system operates reliably and efficiently under varying conditions [68]. It provides the flexibility for the system to adapt to different requirements. In distributed systems, configuration inputs are the parameters or settings that define how the system behaves [62]. These inputs can be divided into two main types: (1) *General configurations*, such as network settings and resource allocation, which are not related to fault tolerance mechanisms. (2) *Fault-dependent configurations*, which determine how the system responds to failures. These include settings like whether a service should be replicated, the frequency of recovery triggers, and thresholds for fault detection. The fault tolerance behavior of the system can vary significantly depending on these configurations, as they control critical aspects like timeout intervals, redundancy levels, and recovery strategies. For example, in a high-replication configuration, the fault handling logic may prioritize data consistency and integrity, triggering immediate recovery processes to synchronize all replicas. In contrast, a low-replication configuration might focus on minimizing downtime, quickly recovering a failed node without waiting for full synchronization across replicas [69]. Therefore, fault-dependent configuration inputs have a significant impact on fault injection testing. Unfortunately, existing fault injection tools overlook this aspect.

3 Motivation Example

Some bugs in fault tolerance mechanisms are difficult to detect because they can only be triggered under specific configuration settings, yet they can lead to severe consequences in distributed systems. One such example is a fault-handling bug in HDFS [8], caused by incorrect type sorting, specifically the failure to properly handle null values. This bug leads to HDFS node crashes, resulting in service unavailability for all applications relying on the system. Figure 2 illustrates the key steps to trigger this bug, while Figure 3 presents the core code snippet of it. In an HDFS cluster, the NameNode manages nodes, while the DataNode stores data. Once the HDFS cluster loads the configuration, the NameNode sets up the LoadBalancer ('balance.start=true' means that load balancing is enabled automatically) and heartbeat mechanism based on the configuration settings. When a DataNode fails to respond to heartbeat packets due to various faults (e.g., persistent network issues), the NameNode considers the DataNode to be offline. As a result, the LoadBalancer detects a storage imbalance and initiates the load reallocation process. The DataNodeManager then sorts the DataNodes based on their actual load. Since 'read.considerStorageType' is set to True, the DataNodeManager first sorts the DataNodes according to their storage types. However, if the previously offline DataNode has just restarted and reconnected, and at that moment the DataNodeManager is fetching its storage type, and the initialization thread responsible for that storage type has not yet completed, a null value will be returned. Subsequently, this bug is triggered because the default Comparator.comparing (line 5) cannot handle comparisons involving null values, causing the NameNode to crash. This fault handling bug is fixed by overriding the comparator and adding handling for null value comparisons, as shown in lines 6-14.

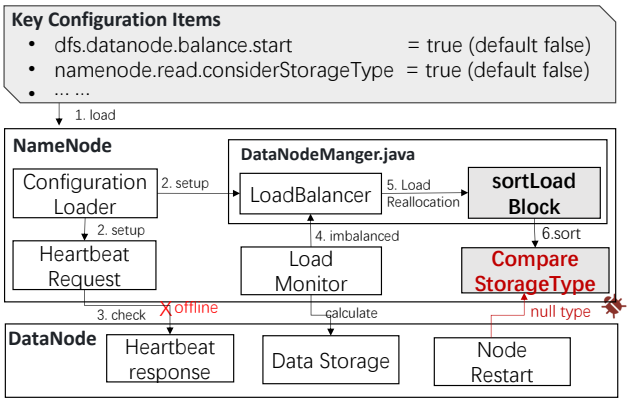


Figure 2: The HDFS-17098 breaks down the NameNode in HDFS, resulting in service unavailable.

```
1  private Consumer<...> sortLoadBlock() {
2     Consumer<...> loadSort = null;
3     if (readConsiderStorageType) {
4       Comparator<DatanodeInfoWithStorage> comp =
5  -       Comparator.comparing(getStorageType);
6  +       Comparator.comparing(getStorageType,
7  +           (s1, s2) -> {
8  +              if (s1 == null) {
9  +                return (s2 == null) ? 0 : -1;
10 +              } else if (s2 == null) {
11 +                return 1;
12 +              } else {
13 +                return s2.compareTo(s1);
14                }));
15      loadSort=list.Collections.sort(list, comp);
16    }
17 }
```

Figure 3: The core code snippet of HDFS-17098. Missing Null Type handling in'sortLoadBlock()'.

Fault tolerance mechanisms are commonly used in distributed systems, and bugs in their implementation are inevitable. Bugs in one node may affect the whole distributed system, thus causing severe consequences such as service hangs or node crashes. We can draw **three important lessons** from this case: 1) The execution logic of fault tolerance mechanisms is influenced by configuration inputs, a factor that existing fault injection tools often overlook. To address this, we propose CAFault, which focuses on enhancing these tools with high-quality configurations. 2) The dependency between configuration items and fault inputs is implicit. In this case, the activation of 'balance.start' and 'considerStorageType' affects the fault handling logic during the node reconnection process, yet they do not appear to have a direct dependency

relationship. If we naively enumerate and explore all combinations of configurations and fault inputs, it would lead to an explosion of the input space. To solve this problem, CAFault dynamically learns, optimizes, and refines the implicit dependencies between configurations and fault inputs by leveraging the runtime information of the distributed system under test. 3) Some fault handling bugs are hidden in deep paths, and triggering them requires a combination of multiple faults. In this case, triggering this bug requires at least one network failure causing a heartbeat timeout, which triggers the load balancing process, as well as a node failure, followed by the successful automatic restart of the node at the right moment. To handle this issue, CAFault employs fault-handling guided fuzzing, dynamically selecting high-quality fault combinations to explore as much fault tolerance logic as possible.

4 CAFault Design

Design goal: A practical configuration-aware fault injection framework should have the following properties.

- *General:* CAFault is designed to find fault handling bugs for most practical distributed systems, from distributed file systems, e.g., HDFS [3], to distributed configuration services, e.g., ZooKeeper [36]. From distributed database systems, e.g., MySQL-Cluster [12], to decentralized file systems, e.g., IPFS [32]. The tool can be deployed to different distributed systems with minor adjustments.
- *Automatic:* The entire testing process is automated, requiring minimal or no manual effort.
- *Efficient:* CAFault is able to constantly exercise the fault tolerance mechanisms and effectively detect more bugs in real-world distributed systems compared to SOTA tools.

4.1 CAFault Workflow

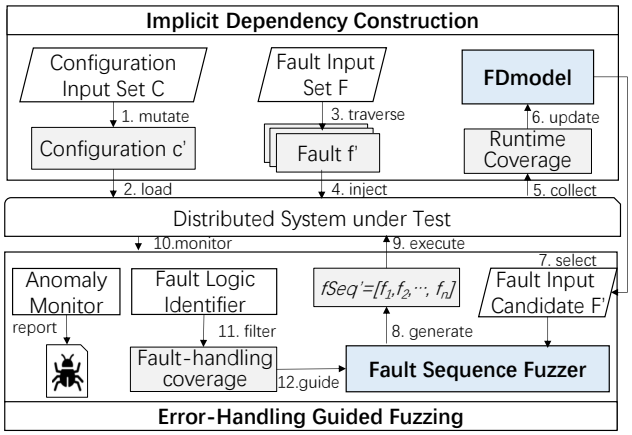


Figure 4: The workflow of CAFault. It includes two main phases: (1) Implicit Dependency Construction and (2) Fault-Handling Guided Fuzzing.

Figure 4 illustrates the workflow of CAFault, which consists of two main phases. The first phase focuses on the construction of the FDModel (fault-dependent model): (1) CAFault selects a configuration from the Configuration Input Set (initially containing the system's default configurations) and performs random mutations to generate a new configuration, c'. (2) The distributed system under test then loads the configuration c'. (3) CAFault iterates through each fault operation in the Fault Input Set (including IO timeout, node crash, etc.). (4) CAFault sequentially injects each fault input F into the system. (5) For each executed fault input F, corresponding coverage information is collected. (6) Based on real-time coverage changes, the FDModel is dynamically updated and refined. In this phase, if c' leads to new coverage improvements in the fault tolerance mechanisms, it is regarded as a high-quality fault-dependent configuration and added to Set C for further exploration. Note that this phase does not require precise attribution of coverage data to individual fault operations, only whether the coverage change needs to be observed. Finally, based on c', CAFault proceeds to the second phase.

The second phase is the fuzzing test process, consisting of the following steps: (7) CAFault uses the FDModel to select fault operations that are dependent on configuration c', creating the fault candidate set F'. (8) The Fault Sequence Fuzzer mutates and generates fault operation sequences $fSeq'$ based on the set F'. (9) CAFault then injects and executes $fSeq'$ in the distributed system under test. (10) Subsequently, CAFault monitors the runtime states of the system in real-time. (11) Finally, the Fault Logic Identifier analyzes the runtime states, filtering out fault-handling coverage information. Simultaneously, the Anomaly Monitor identifies if there are unexpected errors. The bugs are reported once they are detected. (12) Fault sequences that lead to new fault-handling coverage or expose new bugs are prioritized by the Fuzzer to guide the generation of subsequent fault sequences. Importantly, CAFault treats each fault sequence as a single fuzzing unit, collecting feedback at the sequence level rather than for individual faults. As a result, it does not require attributing precise feedback to each fault; instead, it evaluates the overall effectiveness of the entire sequence in exercising fault-handling logic. CAFault proceeds to the next fuzzing iteration (from step 8 to step 12) of the testing process until coverage converges (where newly generated fault sequences no longer yield additional fault-handling code coverage). Afterward, CAFault returns to the first phase, continuously alternating between exploring configuration inputs and generating fault inputs, until termination.

4.2 FDModel Updating

Both configuration inputs and fault inputs influence the execution logic of a distributed system's fault tolerance mechanisms. However, their dependency relationships are often implicit and difficult to uncover directly. CAFault leverages dynamic

runtime information to continuously learn these implicit dependencies, enabling the ongoing updating and optimization of the FDModel. A heuristic insight of this step is that, under the same fault input f_i, if changing a configuration item $item_j$ triggers a change in the execution coverage data, then we consider there to be an implicit execution dependency between item $item_j$ and fault f_i.

Definition of Implicit Dependency: Implicit dependency refers to long and complex control-flow and data-flow relationships between configuration items and fault-handling logic that are difficult to detect. First, we formally define the process of testing a distributed system as $\phi = \{C, F, Cov\}$. Specifically, $C = \{c_1, c_2, c_3, \ldots, c_n\}$ represents the set of configuration inputs to be loaded into the system. $F = \{f_1, f_2, f_3, ..., f_m\}$ represents the set of fault inputs to be injected into the system under test. $Cov = \{cov_{1,1}, \ldots, cov_{i,j}, \ldots, cov_{n,m}\}$ represent the coverage data, and $cov_{i,j}$ means represents the coverage data generated by executing configuration c_i and fault input f_j. For each c in C, $c = \{item_1, item_2, item_3, \ldots, item_l\}$ represents the actual settings of all items in c. Let $diff(c_i, c_j) = \{item_1, item_2, \ldots, item_k\}$ represent the differences between configuration c_i and configuration c_j, where the values of these k items are different. $\forall f_i \in F$, if $diff(c_a, c_b) = \{item_j\}$ and $cov_{a,i} \neq cov_{b,i}$, then we define that item $item_j$ has an implicit execution dependency with fault f_i, denoted as $d_{(item_j, f_i)}$.

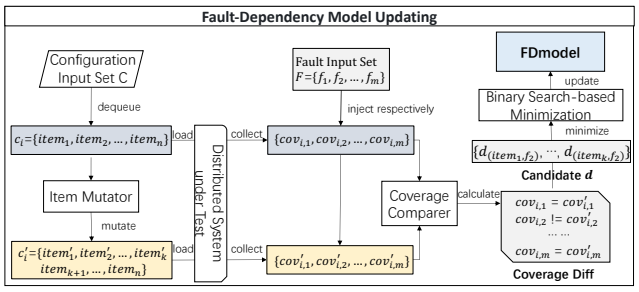


Figure 5: The process of Fault-Dependency Model updating. CAFault continuously learns implicit dependencies and refines the FDModel by analyzing coverage differences.

Figure 5 provides a detailed overview of the FModel update process. CAFault first dequeues a configuration c from the Configuration Input Set C. Then, an item mutator is employed to randomly mutate k items ($k \geq 1$ and $k \leq$ number of all items) based on the selected configuration, generating a new configuration c'. We assume that there are k items $\{item_1, item_2, \ldots, item_k\}$ that are mutated and differ between c and c'. CAFault loads configuration c, and then injects each fault f from the Fault Input Set F respectively, recording the corresponding coverage in fault tolerance mechanisms (will be introduced in section 4.3), denoted as $Cov = \{cov_{i,1}, cov_{i,2}, \ldots, cov_{i,m}\}$. Next, CAFault reloads c' and repeats the previous step, obtaining the coverage information $Cov' = \{cov'_{i,1}, cov'_{i,2}, \ldots, cov'_{i,m}\}$. Then CAFault utilizes a Coverage Comparer to calculate the coverage difference between Cov and Cov'. As shown in Figure 5, if $cov_{i,2} \neq cov'_{i,2}$, it means that changing these k items under the fault input f_2 affects the fault tolerance execution logic of the system under test. Therefore, these k items $\{item_1, item_2, \ldots, item_k\}$ may have an implicit execution dependency with fault f_2, resulting in a candidate dependency d. Considering that not every item in these k items will affect the execution logic of f_2, in order to precisely analyze the dependency between these k items and f_2, we use a binary search-based minimization algorithm to try to extract the smallest subset of items that can trigger the coverage difference. Finally, CAFault applies the minimized dependency d to update the FDModel.

Item Mutation: when mutating configuration items in distributed systems, we apply different strategies based on the type of the item. For boolean configuration items, we mutate their values by performing a logical NOT operation. For enum-type items, we randomly select one value from the available options. For numeric configuration items, we mutate them by adding or subtracting a random value, ensuring that the new value remains within a valid range. For string-type items, we mutate them according to their semantics. If the item represents a time value, we generate a random time in the correct format. For items representing IP addresses or file paths, we skip mutation operations, as arbitrary modifications to these values could disrupt the network topology and prevent the distributed system from functioning properly.

Algorithm 1: Binary Search-based Minimization.

```
Input  : f: fault input
         d: dependency require to be minimized
fn binaryMinize(d, f):
    cov' = execute (d, f) ;
    if cov'==cov then
        d.removeAll () ;
    end
    else
        left = 0, right = len(d) - 1 ;
        mid = (left + right) // 2 ;
        if left < right then
            left_item = d[left:mid] ;
            right_item = d[mid:right] ;
            d_min.add (binaryMinize(left_item,f));
            d_min.add (binaryMinize(left_item,f));
        end
    end
```

Binary Search-based Minimization: Algorithm 1 illustrates the procedure for candidate dependency minimization. The inputs to the minimization algorithm consist of fault injection f and the set of fault-dependent configuration items d to be minimized. The function 'execute(d, f)' refers to the process of executing fault injection f on the configuration c with only the items in set d mutated, while the other items remain unchanged. Coverage is used to determine whether

each change in configuration items leads to a change in coverage. Specifically, the algorithm first splits the candidate item set into two parts: $left_{item}$ and $right_{item}$. It then recursively explores each subset of items (Lines 7-14). If the coverage produced by executing a particular item set is the same as the coverage of the original configuration c, it is considered that all items in that set have no execution dependency on fault f, and these items are removed (Lines 3-5).

4.3 Fault-Handling Guided Fuzzing

The fault injection input space remains vast for each specific configuration. A commonly used method, such as CrashFuzz, uses code coverage-guided fuzzing to explore fault input generation. However, an increase in code coverage does not necessarily equate to an increase in fault tolerance logic coverage, leading to the exploration of many non-fault-handling code paths. An intuitive insight of CAFault is to directly use code in fault tolerance mechanisms as feedback to guide the exploration of fault injection generation. To achieve this, we first need to automatically extract the code related to fault tolerance mechanisms from the coverage data.

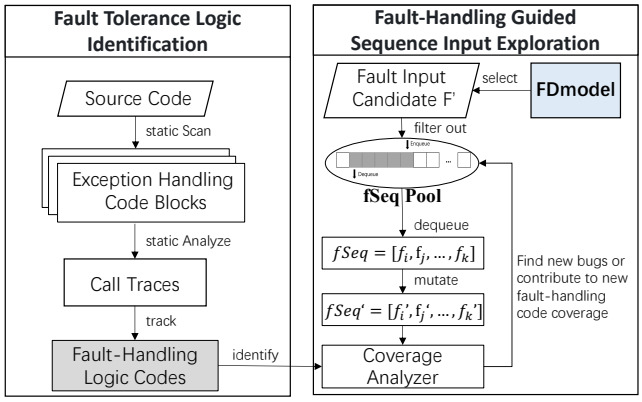


Figure 6: The process of Fault-handling code guided fuzzing. CAFault first constructs a fault-handling code identifier and then dynamically analyzes coverage based on it.

Figure 6 provides a detailed overview of the fault-handling guided fault input exploration process. Before testing begins, CAFault performs static analysis to identify and mark the code related to fault tolerance mechanisms, outputting the fault-handling code. During the testing process, CAFault first uses the FDModel along with the current configuration c of the system under test to select fault operations related to it, placing them into a candidate set F'. Then the fSeq pool is filtered to remove those sequences that include a fault f which does not belong to F'. During coverage analysis, CAFault calculates the fault-handling coverage based on the code sections marked by the Fault Mechanism Logic Identifier, using it as feedback to guide the fuzzing process. Once the fault-handling coverage converges, it indicates that the fault exploration for the current configuration c is complete. CAFault then switches back to the first phase, generating and loading a new high-quality configuration.

Fault Tolerance Logic Identifier. Considering that in most distributed systems, their fault tolerance mechanisms typically involve the capture and handling of various exceptions, the first step is to statically scan the source code of the system under test to identify all exception-handling code blocks. In Java language, this step is achieved by recognizing 'try', 'catch', and 'finally' blocks. In C++ language, this is done by identifying 'try', 'catch', and 'throw' statements. In Go language, this is achieved by identifying 'defer', 'panic', and 'recover' statements. Subsequently, for each exception handling code block, CAFault performs static analysis of its call traces. By tracking these call traces, CAFault identifies and marks codes related to the fault tolerance mechanism, generating a fault-handling code set for subsequent coverage analysis.

Algorithm 2: Fault-Handling Guided Fuzzing.

Input : $DSUT$: Distributed System under Test
c: current configuration loaded in the DSUT
Output : B: fault handling bugs

1 $DSUT$.load (c);
2 B={} $fseqPool$={};
3 $CovAnalyzer$ = setupCoverAnalyzer();
4 $fseqPool$.enqueue (randomInit ())
5 **while** *true* **do**
6 $fSeq = fseqPool$.dequeue() ;
7 $fSeq^{'}$ = mutate($fSeq$) ;
8 $CAFault$.inject($fSeq^{'}$);
9 **async:**
10 $Cover = DSUT$.execute($fSeq^{'}$) ;
11 $Cover_{fault}$ = idenitfy($Cover$) ;
12 **end async**
13 Bug_{new} = checkAnomaly();
14 B_n.append($newBugs$);
15 **if** ($Cover_{fault}$.isNew ()) *or* (Bug_{new} != *NULL*) **then**
16 $fseqPool$.enqueue($fSeq^{'}$);
17 **end**
18 **end**

Fault Sequence Exploration: Algorithm 2 illustrates the fault-handling guided fuzzing process. Before the fuzzing process begins, CAFault first sets up the Distributed System Under Test (DSUT) and loads the configuration c. Then, CAFault initializes a coverage analyzer to collect and calculate fault-handling coverage in real time. Next, CAFault initializes the fault sequence pool $fSeqPool$. If this is the first time entering the fuzzing phase, CAFault randomly selects some faults from the candidate set F' as the initial fault sequence. If it is not the first time entering the fuzzing phase, CAFault loads the fault sequences from the previous fuzzing phase's fSeqPool, as shown in Lines 1-4. Then, the fuzzing process starts.

In each fuzzing iteration, CAFault first dequeues a fault sequence $fSeq$ from the pool and applies mutation strategies commonly used in existing fault injection tools to mutate it into a new sequence $fSeq'$. The mutated fault sequence $fSeq'$ is then injected into the DSUT by CAFault (Lines 6-8). The $fSeq$ is executed asynchronously by the DSUT, and its fault-handling coverage information is analyzed in real time, as shown in Lines 9-12. Meanwhile, an anomaly detector monitors the system's runtime status and reports any new bugs if a fault-handling bug is detected (Lines 13-14). If the $fSeq'$ contributes new fault-handling coverage or leads to the discovery of new bugs, it is regarded as an interesting seed and is stored back in the fSeqPool to guide the next fuzzing iteration. In this way, CAFault continuously generates fault sequences, exploring as much fault tolerance logic as possible.

5 Implementation

We implemented CAFault in the four widely used distributed systems: HDFS, ZooKeeper, MySQL, and IPFS. The reasons we choose them are listed below:

System Popularity: HDFS, a core component of the Apache Hadoop project, is one of the most widely used distributed file systems for data storage [48]. MySQL Cluster is a scalable, real-time, ACID-compliant transactional database, favored for its low cost and multi-master architecture [40]. ZooKeeper is a popular distributed coordination service, enabling processes to synchronize through a shared namespace with high throughput and low latency [36]. IPFS is a widely used peer-to-peer file-sharing and distributed storage system that enables content addressing and efficient data retrieval [2].

Platform Diversity: These distributed systems come from different organizations and are implemented in different programming languages. HDFS and ZooKeeper are developed by Apache Software Foundation in Java language. MySQL-Cluster is developed by MySQL AB in C++ language. IPFS is developed by IPFS Org in the Go language. Implementation and evaluation of these distributed systems can demonstrate that CAFault is a cross-platform and language-independent testing framework with high generality.

Figure 7 presents the components of CAFault, which can be divided into three core parts. The first part is the Configuration Generator, which is implemented for synthesizing high-quality configurations that impact fault tolerance mechanisms in distributed systems under test. The second part is the Fault Injector, which effectively explores fault inputs under each fault-dependent configuration setup. The third part is the Interaction Adaptor, designed to interact with the target system, including standardizing the configuration input format of the distributed systems under test, sending workloads to them, and detecting anomalies within the systems. The rest of the section describes notable implementation details.

Coverage Collector: To collect runtime code coverage information, language-specific instrumentation tools are required for the distributed systems under test. For Java programs, we use Javacoco [33]. For C/C++ programs, we use gcov [18]. For Go programs, we use gtest [20].

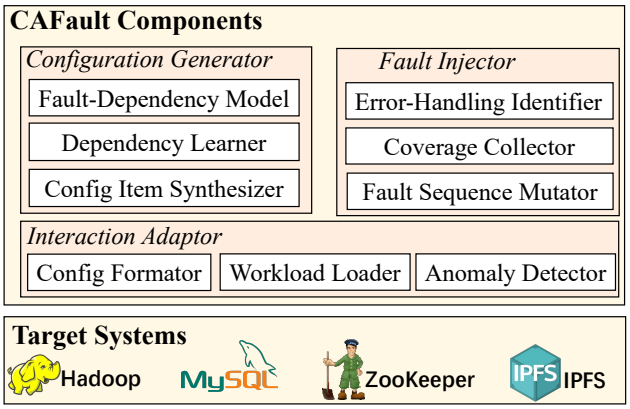


Figure 7: Components of CAFault, including Configuration Generator, Fault Injector, and Interaction Adaptor.

Workload Loader: In distributed system testing, workload refers to the set of tasks, requests, or operations sent to the system to simulate real-world usage. For HDFS, the workload is derived from Intel HiBench [29], a widely used benchmark suite for testing big data processing systems. For MySQL Cluster, the workload is generated using SQLancer [58], a popular SQL generator for testing the robustness and correctness of database systems. For ZooKeeper, the workload is loaded from Apache JMeter [25], a commonly used functional testing tool for performance and stress testing distributed systems, including ZooKeeper. For IPFS, we use ipfs-benchmark [31] to simulate multiple client requests and generate workloads.

Configuration Parameter Mutator: CAFault considers all exposed configuration items of each target system and mutates them using type-aware strategies (e.g., boolean negation, enum substitution, numeric shifting), consistent with existing configuration fuzzing tools such as ECFuzz [41]. The detailed mutation strategy is introduced in Section 4.2.

Fault Type and Detector: Due to the diversity of fault tolerance mechanisms in distributed systems, there are many types of fault handling bugs. Currently, CAFault adopts a set of commonly used faults, including delay, crash, restart, and packet loss, which is consistent with those used by SOTA fault injection tools (i.e., Chronos, Mallory, and CrashFuzz). We also integrate several mainstream bug detectors: For crash recovery bugs [16], we use the detector from CrashFuzz. For timeout bugs [10], we leverage the monitor from Chronos. For other common bugs (including memory vulnerability, inconsistency, etc.), we use three detectors (i.e., AddressSanitizer, log checker, and consistency checker) from Mallory.

Bug Reproducing and Analyzing: When reproducing bugs in distributed systems, especially those involving non-determinism, we follow the strategy adopted by previous fault injection tools such as Phoenix [45]. Specifically, when a bug is detected, we precisely log the execution context of each injected fault, including the fault type, injection timing,

Table 1: 16 new fault handling bugs detected by the tools within 48 hours. CAFault found all 16 bugs, including 3 in HDFS, 6 in MySQL-Cluster, 4 in ZooKeeper, and 3 in IPFS. In comparison, other SOTA tools detected fewer bugs: CrashFuzz found 3 bugs, Mallory found 4 bugs, and Chronos found 1 bug, respectively.

#	Platform	Version	The Root Cause Analysis	Identifier
1	HDFS	3.4.1	NameNode in Quorum-based Mode breaks down due to NullPointer in NameNodeRpcServer after multiple request timeout.	HDFS-202410312
2	HDFS	3.4.1	DataNode fails to recover caused by incorrect connect retries in the node restart process when using default configuration.	HDFS-202410318
3	HDFS	3.4.1	File fetching repeatedly hangs due to NullPointerException in BlockPlacementPolicy when setting dfs.replication=1	HDFS-202410319
4	MySQL-Cluster	8.4	Data loss in NDBD and failure to recover caused by incorrect index deletion when setting HAProxy.balance=roundrobin.	MySQL-S1825461
5	MySQL-Cluster	8.4	Bufferoverflow in sql_opt_exec_shared due to incorrect handling in data resynchronization when setting sync_binlog=0.	MySQL-S1825462
6	MySQL-Cluster	8.4	NDBD fails to recover caused by SEGV in Item_subselect which crashes the server node when using default configuration.	MySQL-S1827319
7	MySQL-Cluster	8.4	Continuous data write timeouts due to incorrect thread allocation in perfschema/pfs.cc When setting RedoBufferSize 1MB.	MySQL-S1827320
8	MySQL-Cluster	8.4	Data inconsistency among ndbds caused by incorrect STATUS in ndb_mgmd restarting when setting NoOfReplicas to 10.	MySQL-S1827321
9	MySQL-Cluster	8.4	SQL Query hangs and fails to recover due to wrong connection retry in conn_handler when using default configuration.	MySQL-S1830147
10	ZooKeeper	3.9	Leader node disconnects other nodes and fails to recover when meeting network faults in fast leader election mode.	ZooKeeper-2024532
11	ZooKeeper	3.9	Data corruption: Zookeeper client erroneously handling absolute path changes when resetting ZOOCFG.dataDir	ZooKeeper-2024550
12	ZooKeeper	3.9	NullPointerException in inputStream field crashes nodes and stops them restarting when using default configuration.	ZooKeeper-2024551
13	ZooKeeper	3.9	Request timeout due to missing ZooDefs.OpCode in RequestMetricsCollector when setting tickTime to 100	ZooKeeper-2024547
14	IPFS	0.33	Inconsistent repo size caused by incorrect calculation of resyncing repository when setting a small ipfs.cache.	IPFS-10252
15	IPFS	0.33	Resources accessing hangs incorrect IPNS address resolution after nodes restart when using the FUSE mounting mode.	IPFS-10573
16	IPFS	0.33	Kubo daemon crashes and fails to recover caused by incorrect pointer access under the default recovery configuration.	IPFS-10217

and executed code location at the time of injection. During the reproduction phase, we strictly replay these faults with the same context to maximize consistency with the original execution. To further mitigate the effects of non-determinism, we re-execute the fault sequence multiple times.

Adaptation to New Distributed Systems: The effort of adapting CAFault to other DFSes could be small. Modules in CAFault are well-encapsulated and loosely coupled. Hence, when adapting CAFault to a new distributed system under test, developers only need to implement three interfaces related to a specific target system. (1) The first interface is 'config.format()', which converts the configuration input format of the target system to CAFault's standard format (XML [37]) and vice versa. Currently, we support several mainstream configuration file formats, such as JSON [44], YAML [1], INI [15], and TOML [63]. If the distributed system uses one of these common file formats, no adaptation is needed. Otherwise, it is necessary to manually implement this format interface, though this situation is rare. (2) The second interface is 'Workloading()', which is used to load the workload required for testing the distributed system under test. Fortunately, most commonly used distributed systems have their own workload generation tools for testing and evaluation. As previously mentioned, we can directly integrate and use these existing workloads through this interface. (3) The third interface is 'statusMonitor()', which is responsible for monitoring key statuses and identifying bugs in the distributed system.

6 Evaluation

To evaluate the effectiveness of CAFault, we compared it with three SOTA fault injection tools: CrashFuzz, Mallory, and Chronos on four widely used distributed systems. These tools were chosen because they represent recent advances published in top-tier conferences. Each of them has been evaluated against a wide range of prior fault injection tools (e.g., Jepsen [34], ChaosMonkey [50], etc.) and has demonstrated superior performance in terms of bug detection and code coverage. We ran each distributed system in a cluster of 20 virtual nodes isolated by Docker [13]. Each Docker has a 2.25 GHz 6-core CPU, 16 GB of RAM, and a 480 GB SATA SSD. They all connect to each other with a 10 Gbps network bandwidth setup. They ran Ubuntu 20.04.2 with Linux kernel version 4.4.0. All Docker containers run in a physical machine, which is a 64-bit machine with 128 CPU cores (AMD EPYC 7742 64-Core Processor), and 512 GB main memory. All the experiments are conducted several times with the same workloads, and the average values are used in this paper. We designed experiments to address the following research questions:

- **RQ1:** Is CAFault effective in finding fault handling bugs of real-world distributed systems?
- **RQ2:** Can CAFault cover more fault tolerance logic in distributed systems compared to other SOTA tools?
- **RQ3:** How effectively does the Fault-Dependency Model enhance testing performance?
- **RQ4:** To what extent does fault-handling guided fuzzing improve the performance of testing?
- **RQ5:** What is the accuracy of CAFault?

6.1 Bugs in Real-World Distributed Systems

We applied CAFault to the latest versions of four distributed systems (HDFS 3.4.1, MySQL-Cluster v8.4, ZooKeeper v3.9, and IPFS v0.33) to evaluate fault handling bug detection. For comparison, we also ran CrashFuzz, Mallory, and Chronos on the same target systems using the same experimental setup. Furthermore, since these existing tools only perform testing on the fixed and default configurations of the distributed systems under test, to provide a fair comparison and validate whether FDModel can enhance the testing performance

of existing fault injection techniques, we also integrate FDModel into these tools, which we refer to as CrashFuzz$^+$, Mallory$^+$, and Chronos$^+$. Specifically, we first loaded fault-dependent configurations from the FDModel generated by CAFault. Then, we applied the existing fault injection techniques (i.e., CrashFuzz, Mallory, and Chronos) to each configuration to explore the fault input space. Once the fault injection test coverage converged for the current configuration, we moved to the next configuration (as described in Section 4.1). Each experiment is conducted for 48 hours. In total, CAFault identified 16 fault handling bugs on four target distributed systems with 3 in HDFS, 6 in MySQL-Cluster, 4 in ZooKeeper, and 3 in IPFS. The detailed information on these fault handling bugs is presented in Table 1.

As shown in Table 1, all 16 identified fault handling bugs have been confirmed and fixed by the corresponding vendors. Among these, except for 5 that were found solely through the default configuration of the system under test, the majority of the bugs (11/16) required testing under various configuration item settings with corresponding fault injection to be triggered. Specifically, most of the fault handling bugs (8/16) caused server nodes to crash, and due to errors in the fault tolerance mechanism, these distributed nodes were unable to recover automatically. Three bugs were caused by errors during the data synchronization process, resulting in data inconsistency across distributed nodes and even the loss of critical data. The remaining 5 bugs caused the distributed services to hang for an extended period, affecting system availability. Some of these fault handling bugs can lead to serious consequences. Take bugs #1, #4 and #10, and #16 for example, these bugs could allow attackers to deliberately crash or hang specific target nodes, disrupting their ability to handle requests or impeding the data synchronization process through the execution of specific fault operations. Such malicious activities could directly result in the loss of critical data or cause the outage of essential cloud services and ultimately lead to significant economic losses.

Table 2: Bugs found by CAFault and other SOTA methods. Existing fault injection tools detect no more than 5 bugs, while CAFault detects 16 unknown bugs.

Tool Name	Number	Bugs ID #
CAFault	16	#1 - 16
CrashFuzz	3	#2, 6, 12
CrashFuzz$^+$	9	#2, 3, 5, 6, 8, 11-14
Mallory	4	#2, 6, 12, 16
Mallory$^+$	12	#2, 3-9, 11-14, 16
Chronos	1	#9
Chronos$^+$	5	#3, 7, 9, 13, 15

Comparison with existing methods: In our 48-hour experiments, CrashFuzz only found 3 bugs, including bugs #2, #6, and #12. Mallory has successfully found 4 bugs, including bugs #2, #6, #12, and #16. Chronos only found 1 bug (#9). However, the remaining 11 bugs were not detected by these tools because these fault-handling bugs require specific fault-dependent configurations to be triggered. Therefore, with the enhancement of FDModel, CrashFuzz$^+$, Mallory$^+$, and Chronos$^+$ successfully detected 8 additional bugs. However, 3 bugs (#1, #4, and #10) were still not detected. This is because they are hidden not only in various specific fault-dependent configurations but also in deep execution paths. Triggering them requires conducting multiple fault interactions across different phases first. With the help of the fault-handling guided fuzzing strategy, CAFault successfully explored the fault tolerance logic hidden in deep paths and detected all 16 bugs, proving the effectiveness of CAFault in detecting fault handling bugs in real-world distributed systems, which adequately answers **RQ1**. Compared with other SOTA tools, CAFault found all the bugs that other tools found.

6.1.1 Case Study

Now we use one case to illustrate how the fault handling bugs detected by CAFault affect the whole distributed system, and how CAFault detects them. This case is the bug #1 listed in Table 1. This bug is a severe node crash, and attackers may utilize it to cause arbitrary distributed nodes to break down by conducting a certain delay strategy. It is caused by an implementation bug that incorrectly manipulates a NullPointer. It is found in version 3.4.1 of HDFS. It can only be triggered when running NameNode in Quorum mode. The code snippet in Figure 8 describes the detailed information.

```
1  private void tryReconnect() {
2      Preconditions.checkState(proxy == null);
3      proxy = createProxy();
4      ...
5      if (quorumEnabled) {
6        address = getConnect(NodeID, server);
7  +     Preconditions.checkArgument(detailedConnetTime
       != null, "Quorum mode enabled, detailed connect
       time metric +should not be null!");
8        detailedConnetTime.add(proxy, address);
9      }
10     updateProcessingDetails(Timing.LOCKWAIT, now -
           startNanos);
11 }
```

Figure 8: A NullPointer bug that crashes HDFS NameNode.

Root Cause: In HDFS, NameNode is designed for managing the filesystem metadata and keeping track of the locations of blocks across the cluster. The NameNode can be configured to run in various modes. To support the HDFS High Availability (HA) feature, the Quorum Journal Manager (QJM) is introduced to share edit logs between the Active and Standby NameNodes [24]. As shown in Figure 8, when heartbeat checks between distributed NameNode nodes fail due to various faults (e.g., network delays), the 'tryReconnect()' function is called to attempt re-establishing the NameNode quorum. In most cases, this code runs normally. However,

when another reconnect thread for the current NameNode is being re-spawned (due to multiple reconnect attempts exceeding the max_try setting), the 'detailedConnectTime' in that thread might not have been properly re-initialized yet. If the 'tryReconnect()' function is called in the current thread before this re-initialization, it triggers a null pointer issue, causing the node to crash and preventing it from recovering automatically. The developer has fixed this fault handling bug by adding a NullPointer checker (lines 7).

This crash bug was introduced when the quorum mode was added in HDFS v2.0.1. However, it remained hidden in the daily usage of HDFS until it was detected for the first time by CAFault. This is mainly because most of the previous fault injection testing tools were only run under HDFS's default configuration (i.e., Standalone Mode), which failed to trigger this bug. Additionally, even though existing tools (CrashFuzz$^+$, Mallory$^+$, and Chronos$^+$) were enhanced with our FDModel, they still did not detect it during our 48-hour experiment. This is because the bug is hidden deep within the fault interaction path: in addition to needing to run in quorum mode, it also requires that at least two connections on the NameNode be in a disconnected state, with one connection exceeding the maximum retry limit, while another connection attempts to reconnect at the same time. Such a scenario is quite rare. Fortunately, with the fault-handling guided fuzzing strategy, CAFault generates a wide range of fault inputs, enabling it to explore various fault-handling logic in the distributed system. This ultimately triggers the fault handling bug hidden in the deep path.

6.2 Fault Tolerance Logic Coverage

To evaluate the effectiveness of CAFault in covering fault tolerance logic in distributed systems, we set up a network for each target system and compared CAFault with other SOTA tools under the same experimental conditions. We measured the coverage of fault-handling code (which represents the fault tolerance mechanisms in the distributed systems under test, as detailed in Section 4.3) over a 48-hour testing period. The statistics are shown in Table 3. A 48-hour duration is commonly used in prior fault injection studies, offering a fair basis for comparison. Moreover, for most tools, code coverage tends to converge within this timeframe, with negligible gains observed beyond that point, as shown in Figure 9. In conclusion, CAFault consistently outperforms other SOTA tools across all four target distributed systems. Compared to CrashFuzz, Mallory, and Chronos, CAFault covers 31.5%, 29.3%, and 81.5% more fault-handling code, respectively, on average. The statistics adequately answer **RQ2**

Compared to their original versions (CrashFuzz, Mallory, and Chronos), the FDModel enhanced tools (CrashFuzz$^+$, Mallory$^+$, and Chronos$^+$) consistently achieve better coverage across all four target systems. Overall, with the help of FDModel, they improved fault tolerance logic coverage by

Table 3: fault-handling code coverage on four target systems in 48 hours. CAFault covers 81.52%, 31.55%, and 29.31% more fault tolerance logics compared with other SOTA tools.

	HDFS	MySQL-Cluster	ZooKeeper	IPFS	Improvement
CrashFuzz	9965	14624	4869	2393	-
CrashFuzz$^+$	13550	17982	6221	2969	↑ 27.69%
Mallory	10058	14951	5132	2361	-
Mallory$^+$	13012	18975	6065	2885	↑ 24.16%
Chronos	7776	10583	3656	1564	-
Chronos$^+$	9650	12953	4412	1962	↑ 23.15%
CAFault	**13684**	**19162**	**6254**	**3097**	↑ **29.31% - 81.52%**

27.69%, 24.16%, and 23.15%, respectively. The key reason for this improvement is that FDModel generates various high-quality, fault-dependent configuration inputs, which enrich the testing scenarios of existing fault injection tools and enhance testing efficiency. Among these existing fault injection tools, Chronos achieves the lowest coverage. This is because Chronos primarily focuses on timeout-related mechanisms in the distributed system under test, neglecting other fault-handling logic, such as crash recovery or failover, which results in lower fault-handling code coverage. In comparison to the FDModel enhanced tools (CrashFuzz$^+$, Mallory$^+$, and Chronos$^+$), CAFault consistently outperforms them across all four target systems. The main reason is that, with the help of fault-handling guided fuzzing,CAFault effectively explores the fault input space for each configuration, thus improving the efficiency of testing fault tolerance mechanisms.

To track the trends of coverage growth over time, we record the fault-handling code coverage every minute over 24 hours, as shown in figure 9. The data shows that the coverage of existing fault injection tools grows significantly during the first 300 minutes on all four target DFSes. The data shows that existing fault injection tools experience significant coverage growth within the first 5 hours across all four target distributed systems. After approximately 10 hours, their coverage gradually converges (only less than 1% branch coverage improvement is observed). In contrast, CAFault exhibits a gradual and steady increase in coverage over the 48-hour testing period, surpassing other state-of-the-art tools after around 8 hours. Specifically, during the first 8 hours, CAFault's test coverage is lower than that of these SOTA tools. This is mainly because CAFault continuously analyzes runtime coverage data during testing, particularly in the early phase, and updates the FDModel accordingly, which introduces some overhead. Additionally, the system under test incurs extra overhead when loading and running the high-quality configurations generated by CAFault. However, once the FDModel is completed (around 8 hours), CAFault's test coverage steadily surpasses that of other SOTA tools and remains consistently higher.

6.3 Effectiveness of Fault-Dependent Model

To evaluate the effectiveness of the FDModel, we also conducted the experiment that compares CAFault with *CAFault*$^-$, a version of CAFault that disables the FDModel and randomly

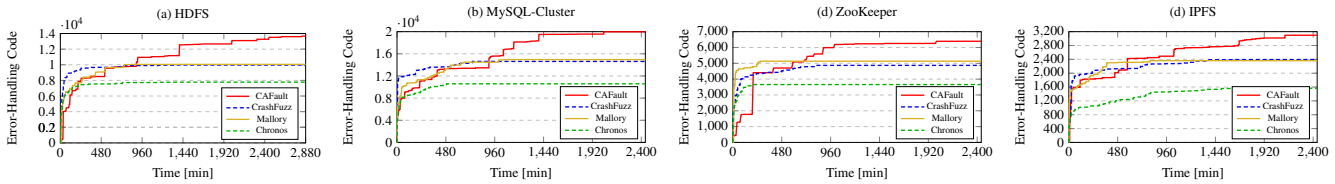


Figure 9: Coverage trends evaluated for CAFault, CrashFuzz, Mallory, and Chronos. Compared with them, CAFault with FDModel and the fault-handling guided fuzzing shows better fault tolerance logic coverage on all the target distributed systems.

generates configuration inputs instead. We collected the fault-handling code coverage and the number of bugs in 48 hours on all four target distributed systems.

Table 4: Comparison of $CAFault^-$ and CAFault on four target systems in 48 hours. CAFault with FDModel detects 128.5% more bugs and covers 29.19% more fault-handling codes.

	Number of Bugs		Fault Handling Coverage	
	$CAFault^-$	CAFault	$CAFault^-$	CAFault
HDFS	1	3	10443	13684
MySQL-Cluster	4	6	14996	19162
ZooKeeper	1	4	4902	6254
IPFS	1	3	2376	3097
Improvement	-	↑ 128.5%	-	↑ 29.19%

As shown in Table 5, with the help of FDModel, CAFault detects all 16 bugs within 48 hours, whereas $CAFault^-$ only detects 7 of them. Specifically, compared to $CAFault^-$, CAFault consistently achieves higher fault-handling code coverage across all four target distributed systems. In total, CAFault covers 9480 more fault-handling codes, resulting in a 29.19% improvement in fault tolerance logic coverage. This demonstrates that the FDModel significantly enhances both fault-handling code coverage and bug detection.

Take Bug #1 listed in Table 1 as an example. CAFault successfully detected this deep fault-handling bug within 48 hours, whereas CAFault$^-$ failed to do so. The key reason lies in the execution dependency between specific configuration and fault inputs required to trigger the bug – specifically, running in quorum mode while experiencing frequent network delays and reconnect attempts. Without the configuration-fault dependency information provided by the FDModel, CAFault$^-$ can only randomly generate combinations of configurations and fault inputs. Given the vast input space, it becomes highly unlikely for CAFault$^-$ to reach the precise conditions needed to expose this bug within the testing time window. Thus, we can conclude that FDModel effectively improves testing performance, which adequately answers **RQ3**.

6.4 Effectiveness of Fault-Handling Feedback

To evaluate the effectiveness of the fault-handling guided fuzzing strategy, we also conducted an experiment that compares CAFault with $CAFault_r$, a version of CAFault that randomly injects faults, and $CAFault_c$, a version of CAFault that utilizes traditional code coverage guided fuzzing. We collected the fault-handling code coverage and the number of bugs in 48 hours on all four distributed systems.

Table 5: Comparison of $CAFault_r$, $CAFault_c$ and CAFault. CAFault with fault-handling guided fuzzing detects 33.3%-77.8% more bugs and covers 11.46%-18.04% more fault tolerance Mechanism codes.

	Number of Bugs			Fault Handling Coverage		
	$CAFault_r$	$CAFault_c$	CAFault	$CAFault_r$	$CAFault_c$	CAFault
HDFS	2	2	3	11607	12316	13684
MySQL-Cluster	3	5	6	15958	17183	19162
ZooKeeper	2	2	4	5402	5657	6254
IPFS	2	3	3	2615	2749	3097

As shown in Table 5, with the help of the fault-handling guided fuzzing algorithm, CAFault detects all 16 bugs within 48 hours, while $CAFault_r$ and $CAFault_c$ detect only 9 and 12 bugs, respectively. Moreover, compared to these methods, CAFault consistently achieves higher fault tolerance logic coverage across all four target distributed systems, covering 18.04% and 11.46% more fault-handling codes in total. Besides, we also use Bug #1 in Table 1 as an illustrative example. Only CAFault successfully detected it, while $CAFault_r$ and $CAFault_c$ failed. This is mainly because the bug is deeply hidden and requires the execution of multiple complex exception-handling steps – specifically, a rare scenario involving simultaneous reconnect attempts under multiple disconnected NameNode connections. With the help of the fault-handling guided fuzzing strategy, CAFault efficiently explores fault-handling logic and ultimately triggers this bug. Therefore, we can conclude that fault-handling guided fuzzing significantly improves both fault tolerance logic coverage and bug detection, leading to a substantial enhancement in testing performance. This effectively addresses **RQ4**.

6.5 Accuracy of CAFault

Due to CAFault's alternating two-phase design, where the first phase uses a fixed set of fault operations to infer configuration-fault dependencies, some dependencies may be missed, particularly when certain configuration items only influence fault-handling behavior under complex fault combinations. To assess potential false negatives, we collected 10 historical bugs from the four target distributed systems, each of which was originally triggered under non-default configurations and specific fault conditions. We configured the systems to the corresponding historical versions and executed CAFault on each

for 48 hours. As shown in Table 6, CAFault successfully reproduced 9 out of the 10 bugs. The one missed case involved a configuration that only exhibited dependency behavior when subjected to a specific combination of multiple faults, a type of complex dependency that CAFault's current design does not yet support. Addressing such complex multifault dependencies to improve FDModel deserves future exploration.

Table 6: Historical bugs reproduced by CAFault.

HDFS	MySQL-Cluster	ZooKeeper	IPFS	Total
2/2	3/4	2/2	2/2	9/10

Considering that FDModel's dynamic dependency analysis is time-consuming, each test requires launching a cluster, running workloads, injecting faults, and cleaning up, efficiency becomes a key concern. A straightforward alternative is to use static analysis to pre-establish configuration-fault dependencies. Specifically, we first construct the Abstract Syntax Tree (AST) of the system's source code and analyze the reachability between nodes where configuration items are accessed and where fault-handling logic resides. If an execution path connects these two nodes, it is considered that an implicit dependency exists between them, thereby forming a coarse-grained FDModel. We then manually evaluated the accuracy of this static approach. However, it produced a high false positive rate of approximately 67%, mainly due to the extensive use of multithreading and asynchronous execution in distributed systems, which static analysis struggles to handle effectively. Thus, we use the dynamic analysis approach to construct the FDModel to avoid such false positives.

7 Discussion

Limitation of FDModel. Exploring both the fault and configuration input space inevitably introduces additional computational complexity. CAFault addresses this by dynamically learning the dependencies between fault inputs and configuration items using runtime coverage feedback to build an FDModel, which helps prune the vast input space. With the help of FDModel, CAFault successfully covers 29.19% more fault-handling codes and detects 128.5% more bugs in fault tolerance mechanisms. However, this dynamic learning process introduces runtime overhead, slowing down the initial testing speed, as observed in the coverage trends in Figure 9.

To mitigate this, future work may incorporate static analysis to quickly construct an initial FDModel by examining the reachability between configuration item nodes and fault-handling code in the source code. While static analysis is efficient, it often introduces a significant number of false positives. Based on our preliminary evaluation, approximately 67% of the inferred dependencies were incorrect, primarily because static analysis cannot accurately capture dynamic execution contexts such as multi-threading and asynchronous events common in distributed systems. How to effectively utilize CAFault's dynamic analysis to refine the initial model and reduce false dependencies remains an important direction for future research. Similarly, leveraging SMT solvers to infer configuration-fault dependencies is also a potential direction. If configuration parameters and control-flow paths can be symbolically represented, SMT solvers could help precompute feasible dependency paths, accelerating FDModel construction. While challenges remain in modeling accuracy and scalability for large codebases of distributed systems, integrating SMT into CAFault deserves future exploration.

Speculate on potential bugs. Given the vast configuration input space, being able to speculate on the potential range of undiscovered faults in unexplored configurations can help effectively prune the search space. While it is difficult to precisely quantify these faults, informed estimation is possible. A natural assumption is that fault-handling bugs are roughly evenly distributed across the fault-handling code. Based on this, we can estimate the bug exposure potential of an unexplored configuration by analyzing how much unique fault-handling logic it is expected to cover. This can be achieved through static or dynamic profiling of the fault-handling code paths activated by each configuration. Configurations likely to trigger more diverse fault-handling behaviors can then be prioritized during testing. We leave the design of such estimation models and prioritization strategies as an important direction for future work.

More bug types support. Currently, CAFault focuses on detecting bugs related to fault tolerance mechanisms, such as crash recovery failures and timeouts, by generating abundant high-quality configurations. However, distributed systems are also prone to many other types of bugs, such as fail-slow [23, 43, 54], race condition [55] bugs, load imbalance bug [7], etc. These tools share a common limitation: they typically test only the default configuration, missing many bugs.

Take load imbalance bugs as an example. These faults occur in distributed systems when uneven workload or data distribution degrades performance, wastes resources, and undermines reliability.. Themis [7] detects them by calculating the load differences between distributed nodes as an evaluation metric. A key insight of CAFault is its ability to generate high-quality configurations related to target bugs, thereby enriching the test scenarios of existing testing tools and improving their test coverage and bug detection capabilities. This approach is orthogonal to the above work. CAFault can first filter out configuration items related to system load, mutate them to generate high-quality configurations, and then use these load-dependent configurations to detect imbalance bugs effectively.

However, different from the fault handling bugs (e.g., crash, recover fail, or hang), which have precise and deterministic definitions, load imbalance bugs are harder to define accurately. 'Imbalance' is a qualitative rather than a quantitative concept, making it difficult to establish a precise definition of what constitutes an imbalance status. As a result, testing for

imbalance bugs often leads to false positives. To address this issue, a reliable and precise oracle for load imbalance bugs needs to be explored in future research.

8 Related Work

Fault injection technology: Fault injection is a widely-used technique for identifying and mitigating potential system failures by intentionally introducing faults or errors into a system [64]. In distributed system testing, there are three main types of fault injection [28]: (1) Implementation-level model checking: Typical tools such as MODIST [66] and SAMC [39] model fault operations (e.g., network delays, disk failures) and abstract system states (e.g., data storage, replication). They then simulate failures and enumerate sequences of nondeterministic events to detect bugs. However, these approaches suffer from the state space explosion problem, making it difficult to explore the vast and complex state spaces in real-world distributed systems. (2) Run-time fault injection: Techniques like chaos engineering inject random faults into a live system. Chaos Monkey [50] and Simian Army [51] from Netflix first simulate network faults in complex cloud environments. Alibaba's ChaosBlade [30] extends support to more fault injections, such as network disruptions and CPU scheduling issues, for comprehensive distributed system testing. Jepsen [34] tests distributed data management systems by performing fault injections using manually written test cases. However, these methods often lack fine-grained control and fail to account for the runtime context. (3) Compile-time fault injection [21, 49]: injecting predefined faults into the source code and testing how the system handles them during execution. CrashFuzz [17] utilizes code coverage guided fuzzing to dynamically optimizes the generation of fault injection inputs for cloud system testing. Mallory [46] constructs a "happen-before" graph to describe the causal relationships between behaviors, and then guides the generation of fuzz fault inputs based on key behavioral states. Chronos [6] proposes deep-priority guided fuzzing to explore fault inputs and detects timeout bugs hidden in deep paths. However, existing fault injection techniques overlook the impact of different configuration inputs on fault tolerance testing in distributed systems, leading to missed some bugs.

Configuration Testing technology: Configuration testing detects bugs in the configuration handling logic of distributed systems by generating a large number of high-quality configuration inputs [67]. Typical tools such as TEA-Cloud [52] and ChatT [57] focus on testing the system's configuration logic for computational resources, including the number of physical machines, CPU usage, network bandwidth, and disk storage capacity. They generate configuration files that cover various computational resource configurations through manually defined generation rules and evaluate system performance under different configurations to uncover potential bugs. Ctests [61] models the dependencies between configurations manually and generates multiple different configuration combinations for testing inputs based on this model using mutation techniques. ConfTest [42] analyzes the system's configuration options, syntax, and semantic constraints to create a detailed configuration model to identify configuration-related logical errors. ECFuzz [41] employs a multi-dimensional configuration generation strategy that formulates diverse mutation strategies based on parameter dependencies, and in each test round, selects and combines multiple parameters from the candidate set. However, existing configuration testing primarily models the dependencies between configuration and workload inputs, neglecting the relationship between configuration and fault inputs. Additionally, their modeling approach relies on manual effort, with accuracy depending on the domain knowledge of the modelers, making it both resource-intensive and time-consuming.

Main Difference: Different from the above work, CAFault focuses on enhancing existing fault injection techniques via abundant fault-dependent configurations to detect more fault handling bugs, thereby improving the resilience and stability of the system. To prune the input space and generate high-quality configurations, CAFault harnesses an FDModel to describe the implicit dependencies between configuration and fault inputs, dynamically optimizing and updating the model based on the runtime state of the system under test. To effectively explore fault injection inputs for each configuration setup, CAFault employs fault-handling guided fuzzing to continuously exercise the fault tolerance mechanisms in distributed systems. Moreover, due to the scalability of the testing framework, CAFault can be quickly adapted to test other distributed systems.

9 Conclusion

In this paper, we propose CAFault, a general testing framework for enhancing existing fault injection techniques with abundant fault-dependent configurations. CAFault first introduces the FDModel to construct implicit dependencies between fault and configuration inputs for pruning the test input space. Then, CAFault harnesses a fault-handling guided fuzzing strategy to effectively explore the fault input space under each high-quality configuration input. We implemented and evaluated CAFault on four widely used distributed systems. Compared with the SOTA tools, crashFuzz, Mallory, and CAFault, CAFault covers 31.5%, 29.3%, and 81.5% more fault tolerance logic. Our future work will consider enhancing CAFault by supporting more types of bug.

Acknowledgements

This research is sponsored in part by the National Key Research and Development Project (No. 2022YFB3104000), and NSFC Program (No. U2441238, 62021002).

References

[1] Oren Ben-Kiki, Clark Evans, and Brian Ingerson. Yaml ainâĂŹt markup language (yamlâĎć) version 1.1. *Working Draft 2008*, 5(11), 2009.

[2] Juan Benet. Ipfs-content addressed, versioned, p2p file system. *arXiv preprint arXiv:1407.3561*, 2014.

[3] Dhruba Borthakur. The hadoop distributed file system: Architecture and design. *Hadoop Project Website*, 11(2007):21, 2007.

[4] Dhruba Borthakur et al. Hdfs architecture guide. *Hadoop apache project*, 53(1-13):2, 2008.

[5] Eric A Brewer. Towards robust distributed systems. In *PODC*, volume 7, pages 343477–343502. Portland, OR, 2000.

[6] Yuanliang Chen, Fuchen Ma, Yuanhang Zhou, Ming Gu, Qing Liao, and Yu Jiang. Chronos: Finding timeout bugs in practical distributed systems by deep-priority fuzzing with transient delay. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 1939–1955. IEEE, 2024.

[7] Yuanliang Chen, Fuchen Ma, Yuanhang Zhou, Zhen Yan, Qing Liao, and Yu Jiang. Themis: Finding imbalance failures in distributed file systems via a load variance model. 2025.

[8] ConfX. Datanodemanager does not handle null storage type properly. https://issues.apache.org/jira/projects/HDFS/issues/HDFS-17098, 2017. Accessed at December 29, 2024.

[9] Flavin Cristian. Understanding fault-tolerant distributed systems. *Communications of the ACM*, 34(2):56–78, 1991.

[10] Ting Dai, Jingzhu He, Xiaohui Gu, and Shan Lu. Understanding real-world timeout problems in cloud server systems. In *2018 IEEE International Conference on Cloud Engineering (IC2E)*, pages 1–11. IEEE, 2018.

[11] Huynh Tu Dang, Pietro Bressana, Han Wang, Ki Suh Lee, Noa Zilberman, Hakim Weatherspoon, Marco Canini, Fernando Pedone, and Robert Soulé. P4xos: Consensus as a network service. *IEEE/ACM Transactions on Networking*, 28(4):1726–1738, 2020.

[12] Alex Davies and Harrison Fisk. *MySQL clustering*. Sams Publishing, 2006.

[13] Inc Docker. Docker. *línea].[Junio de 2017]. Disponible en: https://www. docker. com/what-docker*, 2020.

[14] Ifeanyi P Egwutuoha, David Levy, Bran Selic, and Shiping Chen. A survey of fault tolerance mechanisms and checkpoint/restart implementations for high performance computing systems. *The Journal of Supercomputing*, 65:1302–1326, 2013.

[15] FileFormat. Ini-initiation file format. https://docs.fileformat.com/system/ini/, 2025.

[16] Yu Gao, Wensheng Dou, Feng Qin, Chushu Gao, Dong Wang, Jun Wei, Ruirui Huang, Li Zhou, and Yongming Wu. An empirical study on crash recovery bugs in large-scale distributed systems. In *Proceedings of the 2018 26th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*, pages 539–550, 2018.

[17] Yu Gao, Wensheng Dou, Dong Wang, Wenhan Feng, Jun Wei, Hua Zhong, and Tao Huang. Coverage guided fault injection for cloud systems. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 2211–2223. IEEE, 2023.

[18] Gcov. A test coverage program. https://gcc.gnu.org/onlinedocs/gcc/Gcov.html, 2023.

[19] Victor BF Gomes, Martin Kleppmann, Dominic P Mulligan, and Alastair R Beresford. Verifying strong eventual consistency in distributed systems. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):1–28, 2017.

[20] Googletest. Googletest coverage. https://github.com/google/googletest, 2023.

[21] Joseph Gravellier, Jean-Max Dutertre, Yannick Teglia, and Philippe Loubet Moundi. Faultline: Software-based fault injection on memory transfers. In *2021 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 46–55. IEEE, 2021.

[22] Haryadi S Gunawi, Thanh Do, Pallavi Joshi, Peter Alvaro, Joseph M Hellerstein, Andrea C Arpaci-Dusseau, Remzi H Arpaci-Dusseau, Koushik Sen, and Dhruba Borthakur. {FATE} and {DESTINI}: A framework for cloud recovery testing. In *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11)*, 2011.

[23] Haryadi S Gunawi, Riza O Suminto, Russell Sears, Casey Golliher, Swaminathan Sundararaman, Xing Lin, Tim Emami, Weiguang Sheng, Nematollah Bidokhti, Caitie McCaffrey, et al. Fail-slow at scale: Evidence of hardware performance faults in large production systems. *ACM Transactions on Storage (TOS)*, 14(3):1–26, 2018.

[24] Apache Hadoop. Hdfs high availability using the quorum journal manager. https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/HDFSHighAvailabilityWithQJM.html, 2025. Accessed at December 29, 2024.

[25] Emily H Halili. *Apache JMeter*. 2008.

[26] Abdelsalam A Helal, Abdelsalam A Heddaya, and Bharat B Bhargava. *Replication techniques in distributed systems*, volume 4. Springer Science & Business Media, 2005.

[27] Heidi Howard and Richard Mortier. Paxos vs raft: Have we reached consensus on distributed consensus? In *Proceedings of the 7th Workshop on Principles and Practice of Consistency for Distributed Data*, pages 1–9, 2020.

[28] Mei-Chen Hsueh, Timothy K Tsai, and Ravishankar K Iyer. Fault injection techniques and tools. *Computer*, 30(4):75–82, 1997.

[29] Intel. Hibench suite. https://github.com/Intel-bigdata/HiBench, 2023. Accessed at March 30, 2023.

[30] Chaosblade Io. Chaos blade. https://netflix.github.io/chaosmonkey/, 2022. Accessed at December 29, 2024.

[31] IPFS. Ipfs benchmarks. https://github.com/ipfs/benchmarks, 2023.

[32] IPFS. Ipfs: An open system to manage data without a central server. https://ipfs.tech/, 2024. Accessed at December 6, 2024.

[33] Javacoco. Java code coverage library. https://github.com/jacoco/jacoco, 2023. Accessed on March 31, 2023.

[34] Jepsen. Distributed systems safety research. http://jepsen.io/, 2023.

[35] Yuh-Jzer Joung. Quorum-based algorithms for group mutual exclusion. *IEEE transactions on parallel and distributed systems*, 14(5):463–476, 2003.

[36] Flavio Junqueira and Benjamin Reed. *ZooKeeper: distributed process coordination*. " O'Reilly Media, Inc.", 2013.

[37] Simon St Laurent. *XML: a primer*. IDG Books Worldwide, Inc., 1997.

[38] Abdeldjalil Ledmi, Hakim Bendjenna, and Sofiane Mounine Hemam. Fault tolerance in distributed systems: A survey. In *2018 3rd International Conference on Pattern Analysis and Intelligent Systems (PAIS)*, pages 1–5. IEEE, 2018.

[39] Tanakorn Leesatapornwongsa, Mingzhe Hao, Pallavi Joshi, Jeffrey F Lukman, and Haryadi S Gunawi. {SAMC}:{Semantic-Aware} model checking for fast discovery of deep bugs in cloud systems. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 399–414, 2014.

[40] Arjen Lentz and MySQL AB. An introduction to mysql cluster architecture and use. *MySQL AB, November*, 2006.

[41] Junqiang Li, Senyi Li, Keyao Li, Falin Luo, Hongfang Yu, Shanshan Li, and Xiang Li. Ecfuzz: Effective configuration fuzzing for large-scale systems. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, pages 1–12, 2024.

[42] Wang Li, Shanshan Li, Xiangke Liao, Xiangyang Xu, Shulin Zhou, and Zhouyang Jia. Conftest: Generating comprehensive misconfiguration for system reaction ability evaluation. In *Proceedings of the 21st International Conference on Evaluation and Assessment in Software Engineering*, pages 88–97, 2017.

[43] Ruiming Lu, Erci Xu, Yiming Zhang, Fengyi Zhu, Zhaosheng Zhu, Mengtian Wang, Zongpeng Zhu, Guangtao Xue, Jiwu Shu, Minglu Li, et al. Perseus: A {Fail-Slow} detection framework for cloud storage systems. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*, pages 49–64, 2023.

[44] Teng Lv, Ping Yan, and Weimin He. Survey on json data modelling. In *Journal of Physics: Conference Series*, volume 1069, page 012101. IOP Publishing, 2018.

[45] Fuchen Ma, Yuanliang Chen, Yuanhang Zhou, Jingxuan Sun, Zhuo Su, Yu Jiang, Jiaguang Sun, and Huizhong Li. Phoenix: Detect and locate resilience issues in blockchain via context-sensitive chaos. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, pages 1182–1196, 2023.

[46] Ruijie Meng, George Pîrlea, Abhik Roychoudhury, and Ilya Sergey. Greybox fuzzing of distributed systems. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, pages 1615–1629, 2023.

[47] Bashir Mohammed, Mariam Kiran, Kabiru M Maiyama, Mumtaz M Kamala, and Irfan-Ullah Awan. Failover strategy for fault tolerance in cloud computing environment. *Software: Practice and Experience*, 47(9):1243–1274, 2017.

[48] J Yates Monteith, John D McGregor, and John E Ingram. Hadoop and its evolving ecosystem. In *5th International Workshop on Software Ecosystems (IWSECO 2013)*, volume 50, page 74. Citeseer, 2013.

[49] Kit Murdock, David Oswald, Flavio D Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. Plundervolt: Software-based fault injection attacks against intel sgx. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1466–1482. IEEE, 2020.

[50] Netflix. Chaos monkey. https://netflix.github.io/chaosmonkey/, 2022. Accessed at December 29, 2024.

[51] Army Netflix. Simian army. https://github.com/Netflix/SimianArmy, 2022. Accessed at December 29, 2024.

[52] Alberto Núñez, Pablo C Cañizares, Manuel Núñez, and Robert M Hierons. Tea-cloud: A formal framework for testing cloud computing systems. *IEEE Transactions on Reliability*, 70(1):261–284, 2020.

[53] Diego Ongaro and John Ousterhout. The raft consensus algorithm. *Lecture Notes CS*, 190:2022, 2015.

[54] Biswaranjan Panda, Deepthi Srinivasan, Huan Ke, Karan Gupta, Vinayak Khot, and Haryadi S Gunawi. Iaso: A fail-slow detection and mitigation framework for distributed storage services. In *USENIX Annual Technical Conference*, pages 47–62, 2019.

[55] João Carlos Pereira, Nuno Machado, and Jorge Sousa Pinto. Testing for race conditions in distributed systems via smt solving. In *International Conference on Tests and Proofs*, pages 122–140. Springer, 2020.

[56] Cuong Pham, Long Wang, Byung Chul Tak, Salman Baset, Chunqiang Tang, Zbigniew Kalbarczyk, and Ravishankar K Iyer. Failure diagnosis for distributed systems using targeted fault injection. *IEEE Transactions on Parallel and Distributed Systems*, 28(2):503–516, 2016.

[57] Alif Akbar Pranata, Olivier Barais, Johann Bourcier, and Ludovic Noirie. Chat: Evaluation of reconfigurable distributed network systems using metamorphic testing. In *2021 IEEE Global Communications Conference (GLOBECOM)*, pages 1–6. IEEE, 2021.

[58] Manuel Rigger and Zhendong Su. Testing database engines via pivoted query synthesis. In *OSDI*, volume 20, pages 667–682, 2020.

[59] Nadir K Salih, D Satyanarayana, Abdullah Said Alkalbani, and R Gopal. A survey on software/hardware fault injection tools and techniques. In *2022 IEEE Symposium on Industrial Electronics & Applications (ISIEA)*, pages 1–7. IEEE, 2022.

[60] James PG Sterbenz, David Hutchison, Egemen K Çetinkaya, Abdul Jabbar, Justin P Rohrer, Marcus Schöller, and Paul Smith. Resilience and survivability in communication networks: Strategies, principles, and survey of disciplines. *Computer networks*, 54(8):1245–1265, 2010.

[61] Xudong Sun, Runxiang Cheng, Jianyan Chen, Elaine Ang, Owolabi Legunsen, and Tianyin Xu. Testing configuration changes in context to prevent production failures. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 735–751, 2020.

[62] Amir Taherkordi, Feroz Zahid, Yiannis Verginadis, and Geir Horn. Future cloud systems design: challenges and research directions. *IEEE Access*, 6:74120–74150, 2018.

[63] TOML. A config file format for humant. https://toml.io/en/, 2025.

[64] Jeffrey M Voas and Gary McGraw. *Software fault injection: inoculating programs against errors*. John Wiley & Sons, Inc., 1997.

[65] Shuai Wang, Xinyu Lian, Qingyu Li, Darko Marinov, and Tianyin Xu. Ctest4j: A practical configuration testing framework for java. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, pages 562–566, 2024.

[66] Junfeng Yang, Tisheng Chen, Ming Wu, Zhilei Xu, Xuezheng Liu, Haoxiang Lin, Mao Yang, Fan Long, Lintao Zhang, and Lidong Zhou. Modist: Transparent model checking of unmodified distributed systems. In *NSDI 2009*, pages 213–228, 2009.

[67] Zuoning Yin, Xiao Ma, Jing Zheng, Yuanyuan Zhou, Lakshmi N Bairavasundaram, and Shankar Pasupathy. An empirical study on configuration errors in commercial and open source systems. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 159–172, 2011.

[68] Yuanliang Zhang, Haochen He, Owolabi Legunsen, Shanshan Li, Wei Dong, and Tianyin Xu. An evolutionary study of configuration design and implementation in cloud systems. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 188–200. IEEE, 2021.

[69] Ming Zhong, Kai Shen, and Joel Seiferas. Replication degree customization for high availability. *ACM SIGOPS Operating Systems Review*, 42(4):55–68, 2008.

Revealing Floating-Point Accumulation Orders in Software/Hardware Implementations

Peichen Xie
Microsoft Research

Yanjie Gao
Microsoft Research

Yang Wang
Microsoft Research

Jilong Xue
Microsoft Research

Abstract

Accumulation-based operations, such as summation and matrix multiplication, are fundamental to numerous computational domains. However, their accumulation orders are often undocumented in existing software and hardware implementations, making it difficult for developers to ensure consistent results across systems. To address this issue, we introduce FPRev, a diagnostic tool designed to reveal the accumulation order in the software and hardware implementations through numerical testing. With FPRev, developers can identify and compare accumulation orders, enabling developers to create reproducible software and verify implementation equivalence.

FPRev is a testing-based tool that non-intrusively reveals the accumulation order by analyzing the outputs of the tested implementation for distinct specially designed inputs. Employing FPRev, we showcase the accumulation orders of popular libraries (such as NumPy and PyTorch) on CPUs and GPUs (including GPUs with specialized matrix accelerators such as Tensor Cores). We also validate the efficiency of FPRev through extensive experiments. FPRev exhibits a lower time complexity compared to the basic solution. FPRev is open-sourced at https://github.com/peichenxie/FPRev.

1 Introduction

Today, floating-point computations are ubiquitous, with accumulation-based operations (AccumOps) such as summation, dot products, matrix-vector multiplications, and matrix multiplications playing fundamental roles in various domains. However, no general specification dictates the accumulation orders of AccumOps. Without well-defined requirements, AccumOps are implemented differently across software and hardware, leading to inconsistencies due to the non-associativity of floating-point addition [35]. For example, the half-precision (float16) sum of 0.5, 512, and 512.5 depends on the accumulation order: $(0.5+512)+512.5=1025$, while $0.5+(512+512.5)=1024$. Consequently, varying AccumOp implementations yield different results, complicating reproducibility in software development.

Numerical reproducibility is critical in scientific computing [12, 28, 33], high-performance computing [35], database management system [22], deep learning [4, 31], etc. Particularly, software without verified numerical reproducibility is deemed risky or disqualified when applied to safety-critical or rigorous scenarios like aerospace or banking, where even minor inconsistencies in data are unacceptable. Unfortunately, with the rapid evolution of heterogeneous hardware and the fast iteration of diverse software stacks, reproducibility of AccumOps has become increasingly challenging. Existing implementations rarely disclose their accumulation orders, hindering reproducible AccumOp development.

We propose FPRev, a diagnostic tool to help developers identify how AccumOps are implemented in software and hardware. FPRev reveals the accumulation order of an AccumOp implementation (AccumImpl) through numerical testing. This enables developers to reproduce an AccumImpl on a new system by using the revealed accumulation order as a specification and verify equivalence between two implementations by comparing their accumulation orders.

As a case study, we use FPRev to analyze popular numerical libraries on diverse hardware, uncovering their undocumented and undisclosed accumulation orders. On different CPUs, we apply FPRev to the NumPy library [11]. On different GPUs, we apply FPRev to the PyTorch library [27]. The results indicate that NumPy's summation functions are implemented equivalently across CPUs, and the same holds for PyTorch's summation functions across GPUs. However, other AccumOps relying on BLAS (Basic Linear Algebra Subprograms) backends like Intel MKL [15], OpenBLAS [26], and NVIDIA cuBLAS [24] exhibit non-reproducible behavior.

FPRev also visualizes the order with the summation tree, i.e., a full binary tree representing how an AccumImpl performs summation, to guide develeopment. For example, Figure 1 illustrates NumPy's summation of 32 single-precision (float32) numbers. It divides the 32 numbers into 8 ways, accumulates the summands with a stride of 8 on each way, and sums up the 8 ways together using pairwise summation. This 8-way accumulation order is friendly to CPU's SIMD instruc-

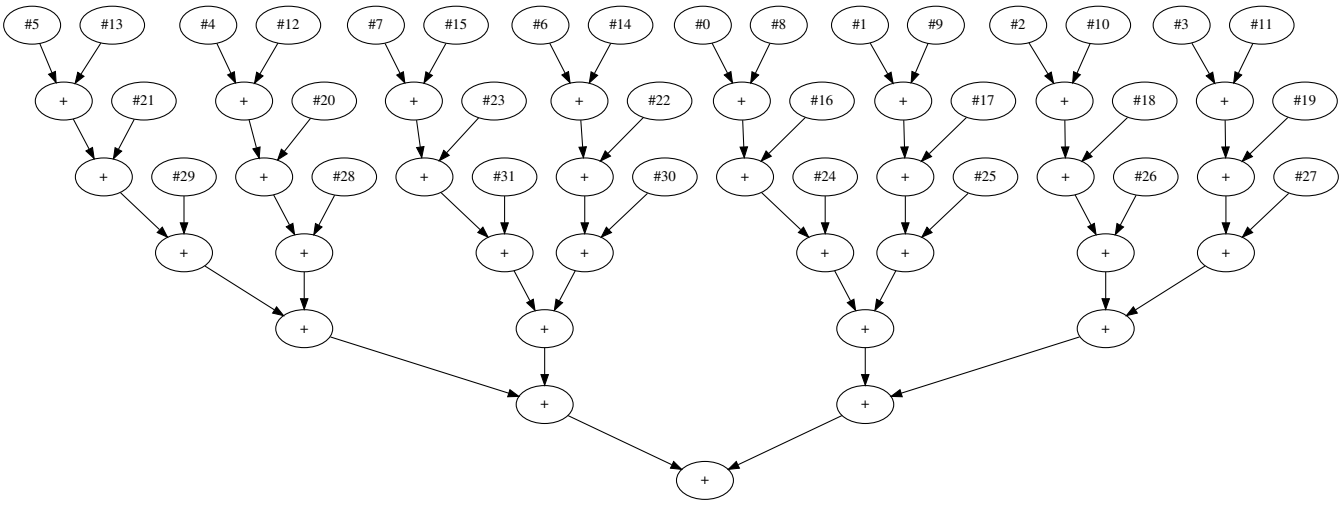


Figure 1: Visualizing the accumulation order of Numpy's summation function for $n = 32$ single-precision numbers with a summation tree. The numbers on the leaf nodes denote the indexes in the input.

tions. With this information, it is easy to replicate NumPy's numerical behavior in a new implementation.

Design overview. Determining the accumulation order of an AccumImpl is a challenging task. Static methods, such as analyzing the source code, are cumbersome and inapplicable to black-box implementations and compiler optimization. Dynamic methods, like scrutinizing the runtime traces, lack an automatic tool to analyze the traces. In addition, many software or hardware implementations are parallel, making the analysis more challenging.

We address these challenges through non-intrusive testing. Recall the example where $(0.5+512)+512.5 = 1025$ and $0.5+(512+512.5) = 1024$ in half precision. Different accumulation orders yield distinct results, making it possible to deduce the order from numerical outputs. However, the number of all possible accumulation orders is exponential, making the time complexity of the naive brute-force solution (NaiveSol) impractical.

To achieve practical time complexity, we propose a basic solution called BasicFPRev that uses specially designed inputs to facilitate the distinguishing process. We take the summation function for example. First, BasicFPRev set all n summands to 1.0. Then, two of the summands are replaced by a very large number (denoted by M) and its negative ($-M$), where M satisfies $(n-2)+M = M$. The summation output corresponds to an integer between 0 and $n-2$, depending on when M or $-M$ cancel each other during the accumulation. Specifically, when $\pm M$ is added to other numbers, it remains $\pm M$; when M is added to $-M$, it results in 0; after that, the remaining summands are accumulated without rounding errors because they are all 1.0. Therefore, the output equals the number of summands accumulated after $M+(-M)$.

BasicFPRev leverages this information to construct the summation tree. Using i and j to denote the indexes of M or $-M$ in the input, we note that the operation $M+(-M)$ corresponds to the lowest common ancestor (LCA) of node $\#i$ and $\#j$ in the summation tree, and the number of leaf nodes under the LCA equals n minus the summation output. Based on this finding, BasicFPRev enumerates i and j, collects the output for the corresponding input, and infers the size of subtree rooted at the LCA of node $\#i$ and $\#j$. BasicFPRev then constructs the summation tree bottom-up, starting with subtrees of two leaf nodes and progressively building larger subtrees until the entire tree is generated.

Based on BasicFPRev, FPRev further reduces time complexity by eliminating redundancy, and adds support for matrix accelerators such as Tensor Cores on recent NVIDIA GPUs [19]. Matrix accelerators are specialized hardware units on GPUs for high-performance matrix multiplication, but they perform non-standard multi-term fused summations [9]. FPRev models their accumulation orders using a multiway tree, where a node with multiple children represents a multi-term fused summation for a group of summands. The summands are aligned and truncated before they are accumulated, as if they are added in finite-precision fixed-point arithmetic.

FPRev has a time complexity of $\Omega(nt(n))$ and $O(n^2t(n))$, where $t(n)$ is the time complexity of the tested AccumImpl. This shows a significant improvement over the $O(4^n/n^{3/2} \cdot t(n))$ complexity of NaiveSol and the $\Theta(n^2t(n))$ complexity of BasicFPRev. Experimental results confirm FPRev's efficiency and scalability across diverse AccumOp implementations on three CPUs and three GPUs with distinct architectures.

In summary, the contributions of this paper include:

1. Design and development of FPRev: we introduce FPRev, a diagnostic tool that non-intrusively reveals the accumulation order of accumulation-based operations implemented in different software and hardware, enabling

developers to verify equivalence and maintain numerical reproducibility between implementations.
2. Empirical analysis of popular implementations: We demonstrate FPRev's capabilities by analyzing accumulation orders in popular libraries (e.g., NumPy and PyTorch) across CPUs and GPUs, providing reproducibility insights in backend implementations.
3. Algorithmic innovation: we describe the algorithm of FPRev for revealing accumulation orders and constructing summation trees, refine the algorithm to reduce time complexity, and extend it to handle modern GPU matrix accelerators (e.g., NVIDIA Tensor Cores), modeling their multi-term fused summation using multiway trees.
4. Performance evaluation: we evaluate FPRev's efficiency through comprehensive experiments, test diverse AccumOps implementations on various CPUs and GPUs, and demonstrate significant performance improvements over naive and basic solutions.

2 Related work

2.1 AccumOp implementations

2.1.1 On canonical CPUs and GPUs

Accumulation-based operations (AccumOps) are implemented diversely on modern systems. On most CPUs and GPUs, implementations use standard IEEE-754 addition or fused multiply-add (FMA) arithmetic [14] to accumulate floating-point numbers. However, they may perform accumulations in different orders without explicitly disclosing those orders.

First, there is diverse numerical software, including BLAS libraries such as Intel MKL [15] and NVIDIA cuBLAS [24], Python libraries such as NumPy [11] and PyTorch [27], and domain-specific compilers such as Numba [17] and Triton [34]. These libraries are developed without a unified specification, making it difficult to guarantee consistent accumulation orders.

Second, the same software may behave differently across different hardware. Different CPUs and GPUs vary in architecture, number of cores, SIMD width, cache size, etc. Consequently, for performance optimization, software may adjust the accumulation order based on the specific hardware characteristic. Specifically, library developers implement various techniques (e.g., different configurations of loop unrolling, block partitioning, cache optimization, and vectorization) for performance tuning, resulting in different accumulation orders. Additionally, auto-tuners (e.g., Triton [34] and TVM [5]) are often used to search for optimal configurations, given the complexity of performance factors such as instruction pipelining and dynamic frequency scaling.

Although order-independent algorithms have been proposed [6–8], which ensure consistent results regardless of the accumulation order, they are highly inefficient and thus rarely used in industry.

Our tool FPRev supports the AccumOp implementations on canonical CPUs and GPUs and can reveal their undisclosed accumulation orders.

2.1.2 On matrix accelerators

Matrix accelerators [19, 29] are specialized hardware components in modern GPUs designed for high-performance matrix multiplication. Developers can implement matrix multiplications using the APIs of matrix accelerators [23]. However, the numerical behavior and accumulation order of the APIs are undocumented and inconsistent across different GPUs.

FPRev supports the AccumOp implementations based on matrix accelerators and can reveal their undisclosed accumulation orders.

2.2 Revelation of numerical behaviors

FPRev achieves non-intrusive revelation of accumulation orders through numerical testing. Prior works [9, 18] have also employed numerical testing to study the numerical behavior of matrix accelerators. They design "corner cases" as test inputs and analyze the numerical behavior based on the outputs. They find that for float64 on NVIDIA Tensor Cores and AMD Matrix Cores, the matrix multiplication instruction is based on a chain of standard FMA arithmetic. In contrast, other instructions use a non-standard arithmetic where multiple terms (the exact number depends on the hardware) are accumulated after alignment and truncation.

FPRev is a general tool that applies to AccumOp implementations, including those based on matrix accelerators, while prior works focus exclusively on specific hardware.

2.3 Numerical reproducibility engineering

The inconsistent AccumOp implementations pose significant issues in numerical reproducibility [2, 35]. To help developers debug the issues, several testing-based tools have been proposed. For example, Varity [16] uses randomized testing to verify equivalence between implementations. Tools like pLiner [10] and its follow-up [20] employ differential testing to pinpoint non-reproducible parts of a program. In contrast, FPRev uses a deterministic testing method to identify the accumulation order of AccumOps.

Early works [12, 33] have emphasized the importance of reproducible AccumOps for ensuring numerical stability but lack practical solutions. A preliminary approach [1] adopts the aforementioned order-independent algorithm [7] to ensure reproducibility, but suffers from its inefficiency. In contrast, FPRev offers a more practical solution: replicating the accumulation order of existing efficient implementations.

3 Problem statement

3.1 Motivation

Accumulation-based operations (AccumOps) are fundamental in floating-point computing, but most implementations do not specify their accumulation orders. This lack of transparency motivates us to design a tool for revealing the accumulation order.

An example application of the tool is in developing reproducible AccumOps, which are key to software reproducibility and service consistency [12, 25, 33]. Developers must ensure that the accumulation order remains consistent across systems to maintain the reproducibility of AccumOps. If developers can determine the accumulation order, they can use it as a specification to guide their development process. In addition, when porting software to a new system, developers need a rigorous way to verify the equivalence of AccumOps between two systems. This can be achieved by comparing the accumulation orders of the AccumOps implemented on two systems.

3.2 Problem definition

For an AccumOp implementation (AccumImpl), we aim to design a diagnostic tool to reveal its accumulation order. For brevity, we focus on the summation function in the following discussion, since other AccumOps can be abstracted as calls to the summation function with the intermediate results as inputs. For example, dot product $\mathbf{x} \cdot \mathbf{y}$ can be treated as $\sum_{i=0}^{n-1} x_i y_i$. Thus, solutions for the summation function can be naturally applied to other AccumOps.

We formulate the summation operation as follows. The floating-point addition is performed $n-1$ times in a predetermined order to calculate the sum of n floating-point numbers. We assume that the accumulation order is unknown but is uniquely determined by the given implementation on specific hardware. Therefore, randomized implementations and those where the order depends on the values of the summands are out of scope[1].

The problem is how to reveal the accumulation order in an summation implementation, denoted by SUMIMPL. Specifically, the input of our revelation algorithm is SUMIMPL and the number of summands n. The output of the algorithm is the accumulation order of SUMIMPL.

Strictly speaking, the accumulation order is represented by a computational graph called the summation tree, which is a rooted full binary tree with n leaf nodes and $n-1$ inner nodes. Each addition operation corresponds to an inner node, which represents the sum of this operation. The two children of the node represent the two summands of this operation. For example, Figure 1 depicts the accumulation order of Numpy's summation function for $n = 32$ single-precision numbers.

[1]While randomized or input-dependent summation algorithms are possible, they tend to be inefficient. Similarly, algorithms using AtomicAdd are affected by thread scheduling, but we did not find popular libraries implementing such algorithms.

3.3 Inefficiency of the naive solution

We now introduce a naive solution (NaiveSol) to the problem, which is based on brute-force search. We design a recursive algorithm to enumerate every possible accumulation orders. For each order, we verify its correctness through randomized testing. Specifically, we generate multiple random inputs, compute the sums in the current order, and compare the results with those from SUMIMPL. If the results match, we accept the order.

The time complexity of NaiveSol is $O(4^n/n^{3/2} \cdot t(n))$, as the number of all possible orders is the $(n-1)$-th Catalan number $C_{n-1} = \frac{(2n-2)!}{n!(n-1)!} = O(4^n/n^{3/2})$. Here, $t(n)$ represents the time complexity of SUMIMPL. In addition to being inefficient, NaiveSol is not fully reliable because different orders can produce the same output for certain inputs. Although the probability is low and reliability can be improved by increasing the number of test inputs, a deterministic solution with full reliability is preferable, as we will achieve next.

4 Basic polynomial-time solution

The exponential complexity of the naive solution is highly impractical. To address the issue, we present our basic solution called BasicFPRev for revealing the accumulation order, which reduces the time complexity to polynomial. We design an algorithm to determine the accumulation order from the numerical results of the tested summation implementation (SUMIMPL) for specially designed testing inputs. The following parts detail the three steps of the algorithm.

4.1 Step 1: designing testing inputs

To facilitate distinguishing the accumulation order, we leverage the swamping phenomenon of floating-point addition [13]. When two floating-point numbers differing by many orders of magnitude are added, the smaller number is swamped and makes no contribution to the sum. For example, $2^{24}+1$ equals 2^{24} in single-precision (float32) arithmetic.

To induce and utilize this phenomenon, we construct various "masked all-one arrays" as testing inputs. Specifically, let n denote the number of summands, and let SUMIMPL represent an summation implementation with a predetermined but unknown accumulation order. Let M be a very large floating-point number that readily induces the swamping phenomenon. For example, we set $M = 2^{127}$ for float32 or $M = 2^{1023}$ for float64. Then, we define a masked all-one

array $A^{i,j}$ as $A^{i,j} = (A_0^{i,j}, A_1^{i,j}, ..., A_{n-1}^{i,j})$ such that

$$A_k^{i,j} = \begin{cases} M & \text{if } k = i \\ -M & \text{if } k = j \\ 1.0 & \text{otherwise} \end{cases}$$

where i and j denote the indexes of M and $-M$ in the array. In $A^{i,j}$, there exist exactly one M and one $-M$, with all other elements being 1.0.

We use $\pm M$ as masks. Specifically, $M + \sigma = M$ and $-M + \sigma = -M$ hold for $0 \le \sigma \le n-2$ in floating-point arithmetic, if $n \ll M$. Therefore, in SUMIMPL($A^{i,j}$), adding any summand or intermediate sum (except M and $-M$ themselves) to $\pm M$ yields $\pm M$. In other words, M and $-M$ can mask the summands or intermediate sums added to them.

As a result, the output of SUMIMPL($A^{i,j}$) depends on the accumulation order and we can distinguish the accumulation order from the output. For example, given $n = 3$ and $A^{0,1} = (M, -M, 1)$, sequential summation $M + (-M) + 1$ corresponds to 1, stride summation $M + 1 + (-M)$ corresponds to 0, and reverse summation $1 + (-M) + M$ corresponds to 0. If the output equals 1, then we can infer the accumulation order is sequential summation. If the output equals 0, we can determine the exact accumulation order by further testing with $A^{0,2}$ as the input.

4.2 Step 2: analyzing the accumulation order from the outputs

To analyze the accumulation order, we call SUMIMPL with $n(n-1)/2$ inputs, i.e., $A^{i,j}$ for $0 \le i < j < n$. Each output reveals information about the accumulation order. Specifically, since $\pm M$ mask the summands or intermediate sums added to them, these numbers make no contribution to the sum. In contrast, only those summands not masked by $\pm M$ contribute to the sum. Therefore, the output equals the sum of these summands. Since each of the summands equals 1.0, the output equals the number of the summands not masked by $\pm M$:

$$n^{i,j}_{\text{not masked}} = \text{SUMIMPL}(A^{i,j}).$$

Then, we can also obtain the number of the summands masked by $\pm M$ by calculating $n^{i,j}_{\text{masked}} = n - 2 - n^{i,j}_{\text{not masked}}$.

How does this information relate to the order, or specifically, the summation tree? Recall that i and j denote the positions of the masks, represented by node #i and #j in the summation tree. **We note that the neutralization of the two masks (i.e., the addition operation $M + (-M) = 0$) corresponds to the lowest common ancestor (LCA) of node #i and #j** . Then, observing the subtree rooted at the LCA, we find that all the summands masked by $\pm M$ are in the subtree, and all the summands not masked by $\pm M$ are out of the subtree. Therefore, the number of leaf nodes in the subtree (representing the size of the subtree) equals $n - n^{i,j}_{\text{not masked}}$, denoted by

$$l^{i,j} = n - n^{i,j}_{\text{not masked}} = n - \text{SUMIMPL}(A^{i,j}).$$

For brevity, we use $l^{i,j}$ to denote "the number of leaf nodes in the subtree rooted at the LCA of node #i and #j " in the rest of the paper.

Take Algorithm 1 as an example SUMIMPL, whose accumulation order is depicted in Figure 2.[2] If $i = 2$ and $j = 4$, then the array $A^{2,4}$ is $(1, 1, M, 1, -M, 1, 1, 1)$. Computing the sum of $A^{2,4}$ with the example SUMIMPL, the 3rd summand and the intermediate sum of the 0th and 1st summands are masked by M (the 2nd summand); the 5th summand is masked by $-M$ (the 4th summand). Therefore, in total, $n^{2,4}_{\text{masked}} = 4$. In contrast, the 6th and 7th summands and their intermediate sum are not added to M or $-M$, so $n^{2,4}_{\text{not masked}} = \text{SUMIMPL}(A^{2,4}) = 2$.

Algorithm 1 An example summation implementation.

```
float sum = 0;
for (int i=0; i<8; i+=2)
    sum += a[i] + a[i+1];
```

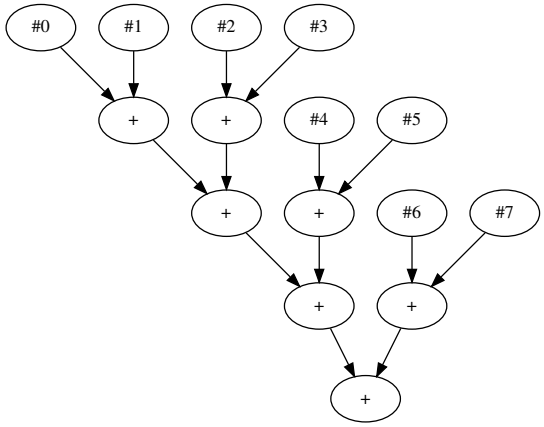


Figure 2: The summation tree of Algorithm 1. The numbers on the leaf nodes denote the indexes in the input.

The LCA of node #2 and #4 is the grandparent node of node #4, as shown in 2. It corresponds to the neutralization of the 2nd summand M and the 4th summand $-M$, i.e., $M + (-M) = 0$. Within the subtree rooted there, there are node #0, #1, #2, #3, #4, and #5 (6 leaf nodes in total), corresponding to the two masks and the summands masked by them. In contrast, node #6 and #7 (2 leaf nodes in total), which correspond to the summands not masked, are out of the subtree. Therefore, $l^{2,4} = 8 - 2 = 6$. Table 1 shows more examples of the output of SUMIMPL($A^{i,j}$) and $l^{i,j}$ for Algorithm 1.

[2] We compile it with different compiler versions and optimization flags, and note that they do not change the accumulation order, even if they might in theory. We suspect such changes can be found on more complex math expressions instead of summations.

Table 1: The order-related information $l^{i,j}$ inferred from the outputs of Algorithm 1 with different masked all-one arrays $A^{i,j}$ as inputs.

i	j	Input: $A^{i,j}$	Output	$l^{i,j}$
0	1	$(M,-M,1,1,1,1,1,1)$	6	2
0	2	$(M,1,-M,1,1,1,1,1)$	4	4
0	3	$(M,1,1,-M,1,1,1,1)$	4	4
0	4	$(M,1,1,1,-M,1,1,1)$	2	6
0	5	$(M,1,1,1,1,-M,1,1)$	2	6
0	6	$(M,1,1,1,1,1,-M,1)$	0	8
0	7	$(M,1,1,1,1,1,1,-M)$	0	8
		...		
2	3	$(1,1,M,-M,1,1,1,1)$	6	2
2	4	$(1,1,M,1,-M,1,1,1)$	2	6
		...		

4.3 Step 3: generating the summation tree

With the information $L=\{(l^{i,j},i,j)\}$ derived from the outputs, where $l^{i,j}$ represents the size of the subtree rooted at the LCA of node #i and #j, generating the summation tree from L is a tree algorithm problem.

Our solution employs a bottom-up approach to construct the tree. First, we sort L in ascending order. For each $l^{i,j}$, we locate the root of the existing subtree containing node #i and the root of the existing subtree containing node #j, and merge them by creating a new parent node for them. By repeating this process, we construct the entire summation tree, progressing from small subtrees to larger ones.

For example, consider the order-related information $L=\{(l^{i,j},i,j)\}$ shown in Table 1. To generate the summation tree, we start by initializing the tree with eight disjoint nodes labeled 0 to 7. Then, examining the smallest value in L, we have $l^{0,1}=2$. This implies that the subtree rooted at the LCA of node #0 and #1 should have 2 leaf nodes. Since the summation tree is a full binary tree, node #0 and #1 are exactly the two children of the root of this subtree. Therefore, we add a new node to the tree, label it with n plus the label of its left child (i.e., $n+0$ in this example), and add two edges from the two leaf nodes to the new node.

For $l^{2,3}=l^{4,5}=l^{6,7}=2$, similarly, we can construct the subtree containing node #2 and #3, the subtree containing node #4 and #5, and the subtree containing node #6 and #7. Now, we have four subtrees of size 2, where the size of a subtree is represented by the number of leaf nodes in it.

Next, the smallest unexamined value in L is $l^{0,2}=4$. This implies that the subtree rooted at the LCA of node #0 and #2 should have 4 leaf nodes. We note that node #0 and #2 are currently in two different subtrees (each has 2 leaf nodes), so we should merge the two subtrees. Therefore, we find the current roots of the subtrees containing node #0 and #2 respectively, i.e., node #i' and #j' where $i'=n+0$ and $j'=n+2$. Subsequently, we add a new node as the parent of node #i' and #j', label it with n plus the label of its left child, i.e., $n+i'=2n+0$, and add two edges from node #i' and #j' to the new node.

For $l^{0,3}=4$, we find that node #0 and #3 are already in the same subtree with 4 leaf nodes, so we just skip it. The same process is applied to $l^{1,2}=l^{1,3}=4$. Now, we have a subtree with 4 leaf nodes and two subtrees with 2.

The next smallest unexamined value in L is $l^{0,4}=6$, which implies that the subtree rooted at the LCA of node #0 and #4 should have 6 leaf nodes. Similarly, node #0 and #4 are currently in two different subtrees (one has 4 leaf nodes and the other has 2), so we should merge the two subtrees. Therefore, following the similar process, we find the current roots of the trees containing node #0 and #4 respectively, i.e., node #i' and #j' where $i'=2n+0$ and $j'=n+4$. Subsequently, we create a new node as their parent (labelled as $n+i'=3n+0$), and add two edges from node #i' and #j' to it.

For $l^{0,5}=6$, we find that node #0 and #5 are already in the same subtree with 6 leaf nodes, so we skip it. The same process is applied to $i\in\{1,2,3\}$ and $j\in\{4,5\}$. Now, we have a subtree of with 6 leaf nodes and a subtree of with 2.

Finally, in the similar way, the next smallest unexamined value in L is $l^{0,6}=8$, which implies that the subtree rooted at the LCA of node #0 and #6 should have8 leaf nodes. Since node #0 and #6 are currently in two different subtrees (one has 6 leaf nodes and the other has 2), we should merge the two subtrees. After we add the parent node of the current roots of the two subtrees and add the corresponding edges, the entire summation tree is generated.

To generalize and formulate the algorithm, we present BasicFPRev in Algorithm 2, where the GENERATETREE function encapsulates Step 3. It first initializes T with n disjoint nodes and no edges. Then, for each $l^{i,j}$, it finds the roots of the existing subtrees containing node #i and #j . If they are identical, then the i-node #i and #j are already in the same subtree. Otherwise, it combines them. The FindRoot function can be implemented by the disjoint-set data structure, resulting in an amortized time complexity of $O(\alpha(n))$ [32], where $\alpha(n)$ is the inverse Ackermann function.

4.4 Time complexity and correctness analysis

Let $t(n)$ denote the time complexity of SUMIMPL. The computation of L has a time complexity of $\Theta(n^2t(n))$. In GENERATETREE, the time complexity of sorting $n(n-1)/2$ elements is $\Theta(n^2\log n^2)=\Theta(n^2\log n)$. Therefore, the time complexity of GENERATETREE is $\Theta(n^2\log n+n^2\alpha(n))=\Theta(n^2\log n)$. Thus, the overall time complexity of BasicFPRev is $\Theta(n^2t(n))+\Theta(n^2\log n)=\Theta(n^2t(n))$.

The correctness of BasicFPRev is inherent in its design and can be proven as follows. For a given implementation SUMIMPL and n, we use T to denote the real summation tree and define $T'=\text{BASICFPREV}(\text{SUMIMPL},n)$. Assuming $T\neq T'$, there must exist i and j such that $l_T^{i,j}\neq l_{T'}^{i,j}$. Now we con-

Algorithm 2 BasicFPRev: our basic solution for revealing the accumulation order.

Require: Implementation SUMIMPL and the number of summands n
Ensure: Summation tree of SUMIMPL

```
function BASICFPREV(SUMIMPL, n)
    L ← ∅
    for i ← 0 to n − 1 do
        for j ← i + 1 to n − 1 do
            A^{i,j} ← (1, 1, ..., 1); A^{i,j}_i ← M; A^{i,j}_j ← −M
            l^{i,j} ← n − SUMIMPL(A^{i,j})
            L ← L ∪ {(l^{i,j}, i, j)}
    function GENERATETREE(L)
        T ← ∅
        for (l^{i,j}, i, j) ∈ L in ascending order do
            i' ← T.FindRoot(i)
            j' ← T.FindRoot(j)
            if i' ≠ j' then
                k ← i' + n                ▷ assign a new label
                T ← T ∪ {(i', k), (j', k)}
        return T
    return GENERATETREE(L)
```

struct $A^{i,j}$ and compute its sum in the order T and T' respectively, resulting in s and s'. Then, $s = n - l_T^{i,j} \neq s' = n - l_{T'}^{i,j}$. However, since $s = \text{SUMIMPL}(A^{i,j})$, then $l_{T'}^{i,j} = n - s' \neq n - s = n - \text{SUMIMPL}(A^{i,j})$. This contradicts the statement $l^{i,j} \leftarrow n - \text{SUMIMPL}(A^{i,j})$ in Algorithm 2. Therefore, the assumption $T \neq T'$ is false, so $T = T'$.

5 Algorithm improvement in FPRev

This section introduces the full version of our tool FPRev, which is evolved from our basic solution BasicFPRev detailed in Section 4. First, we refine the algorithm to reduce its time complexity. Second, based on the refined algorithm, we add support for multi-term fused summation, which is used by matrix accelerators, and finalize FPRev.

5.1 Reducing time complexity

5.1.1 Removing redundancy

By analyzing BasicFPRev, we observe that Algorithm 2 requires $n(n-1)/2$ different $(l^{i,j}, i, j)$ tuples, even though many values of $l^{i,j}$ are identical. However, only $n-1$ new nodes and $2(n-1)$ new edges are constructed. Since computing multiple $l^{i,j}$ by calling SUMIMPL is the primary source of the method's time complexity, reducing redundancy in $l^{i,j}$ (i.e., the cases corresponding to $i' = j'$ in Algorithm 2) can significantly improve efficiency.

To achieve this, we calculate $l^{i,j}$ on demand. Specifically, we do not calculate all $l^{i,j}$ ahead of the tree generation. Instead, we directly start to generate the summation tree, and calculate $l^{i,j}$ when needed. Following the bottom-up idea, we still construct subtrees from leaf to root.

Step 1. We use the set $I = \{0, 1, ..., n-1\}$ to denote the labels of the leaf nodes for which we are going to construct a summation tree. Let i represent the leaf node with the smallest label in I. The sibling node of node #i is either a leaf node or an inner node. If it is a leaf node, there exists a unique j such that $l^{i,j} = 2$. Otherwise, if it is an inner node, then $l^{i,j} > 2$ for all j such that $j \neq i$. Therefore, to distinguish the two cases, we need to calculate $l^{i,j}$ for all other js, denoted by the set $L_i = \{l^{i,j} : j \in I - \{i\}\}$. We examine the minimum value in L_i, which is denoted by $l = \min(L_i)$.

If l equals 2, let j be the one that satisfies $l^{i,j} = 2$. Then, node #j is the sibling node of node #i, so we add a new node to the tree, and add two edges from node #i and #j to the new node. Now, the currently constructed subtree has 2 leaf nodes.

Otherwise, if l is larger than 2, the sibling node of node #i must be an inner node. The subtree rooted at this inner node must have $l-1$ leaf nodes. Let $J_l = \{j : j \in I - \{i\} \wedge l^{i,j} = l\}$. Then, the number of members of J_l must be $l-1$, and the members of J_l are exactly the leaf nodes of this subtree. This can be proven by contradiction. Now, constructing this subtree is a subproblem for the set J_l. Suppose that we have constructed this subtree by a recursive algorithm. We shall add a new node to the tree, and add edges from node #i and the root node of this subtree to the new node. Now, the currently constructed subtree has l leaf nodes.

Summarizing the two cases, we can treat both cases as the same pattern: finding $J_l = \{j : j \in I - \{i\} \wedge l^{i,j} = l\}$ and solving the subproblem for J_l. The first case ($|J_l| = 1$) just leads to the stop condition of the recursion ($|I| = 1$).

Step 2. Now we have constructed a subtree with l leaf nodes. Let r be the root of this subtree. Similarly, to find the sibling node of r, we examine the minimum value in the rest of L_i, which is denoted by l' here. Then, we solve the subproblem for $J_{l'} = \{j : j \in I - \{i\} \wedge l^{i,j} = l'\}$, and get a subtree whose leaf nodes are $J_{l'}$. The root of the subtree, whether a leaf node or an inner node, is the sibling node of r. Therefore, we shall add a new node to the tree, and add edges from r and the root node of the subtree to the new node. Now, the currently constructed subtree has l' leaf nodes.

Remaining steps. We repeat the above step until all values in L_i are examined and the entire tree is constructed. We implement this method with a recursive algorithm, as shown in Algorithm 3.

5.1.2 Demonstration with example

Consider the example SUMIMPL in Algorithm 1, whose summation tree is illustrated in Figure 2. We call Algorithm 3 with this SUMIMPL and $n = 8$. First, the set of leaf nodes is

Algorithm 3 Refinement of BasicFPRev (Algorithm 2).

Require: Implementation SUMIMPL and the number of summands n
Ensure: Summation tree of SUMIMPL

```
function BASICFPREVREFINED(SUMIMPL, n)
    function BUILDSUBTREE(I)
        T ← ∅
        if |I| = 1 then                          ▷ stop condition
            return T
        i ← min(I); L_i ← ∅
        for j ∈ I − {i} do                       ▷ calculate l^{i,j} on demand
            A^{i,j} ← (1,1,...,1); A_i^{i,j} ← M; A_j^{i,j} ← −M
            l^{i,j} ← n − SUMIMPL(A^{i,j})
            L_i ← L_i ∪ {l^{i,j}}
        r ← i                                    ▷ current root of the subtree
        for l ∈ L_i in ascending order do
            J_l ← {j : j ∈ I − {i} ∧ l^{i,j} = l}
            T' ← BUILDSUBTREE(J_l)
            T ← T ∪ T'
            T ← T ∪ {(r, r+n), (GetRoot(T'), r+n)}
            r ← r+n
        return T
    return BUILDSUBTREE({0,1,...,n−1})
```

$I = \{0,1,...,7\}$, where the smallest label is $i = 0$. Next, the set $L_i = \{l^{i,j} : j \in I - \{i\}\} = \{2,4,4,6,6,8,8\} = \{2,4,6,8\}$ is computed. Examining the smallest value in L_i, we have $l = 2$ and $J_l = \{j : j \in I - \{i\} \wedge l^{i,j} = l\} = \{1\}$. Therefore, BUILDSUBTREE($\{1\}$) is called, reaching the stop condition. Then, the subtree with node #0 and #1 as its leaf nodes is constructed. The root of this subtree is denoted by r.

Next, examining the smallest value in the rest of L_i, we have $l = 4$ and $J_l = \{j : j \in I - \{i\} \wedge l^{i,j} = l\} = \{2,3\}$. Therefore, BUILDSUBTREE($\{2,3\}$) is called, where we have $I = \{2,3\}$, $i = 2$, and $L_i = \{2\}$, and BUILDSUBTREE($\{3\}$) is called there. BUILDSUBTREE($\{2,3\}$) returns the subtree with node #2 and #3 as its leaf nodes. We then designate its root as the sibling node of r, and construct the parent node of this root and r. Then, the subtree with node #0, #1, #2, and #3 as its leaf nodes is constructed. r is updated by the root of this subtree.

The next smallest value is $l = 6$. We have $J_l = \{4,5\}$. Similarly, BUILDSUBTREE($\{4,5\}$) is called, and it returns the subtree with node #4 and #5 as its leaf nodes. We merge its root with r, and construct the subtree with node #0, #1, #2, #3, #4, and #5 as its leaf nodes. r is updated by the root of this subtree.

Finally, $l = 8$ and $J_l = \{6,7\}$. BUILDSUBTREE($\{6,7\}$) is called, and it returns the subtree with node #6 and #7 as its leaf nodes. We merge its root with r. Then the entire tree is constructed.

5.1.3 Time complexity

The time complexity of Algorithm 3 is $O(n^2 t(n))$ and $\Omega(nt(n))$. The worst-case scenario occurs when adding n summands in the right-to-left order. In this case, BUILDSUBTREE will be invoked with all suffixes of $\{0,1,...,n-1\}$, and $l^{i,j}$ for all $0 \le i < j < n$ will be calculated. The worst-case time complexity is $\Theta(n^2 t(n))$. In practice, this order is cache-unfriendly, and thus no library uses it.

The best-case scenario corresponds to the sequential summation, where the summation tree will be constructed in one pass, and only $l^{0,j}$ for all $0 < j < n$ will be calculated. The best-case time complexity is $\Theta(nt(n))$. In practice, many libraries use similar orders, because these orders are cache-friendly and efficient.

5.2 Adding support for matrix accelerators

5.2.1 Multi-term fused summation

Matrix accelerators such as NVIDIA Tensor Cores are specialized hardware components in modern GPUs. Matrix accelerators enable assembly instructions that take a matrix $A = (a_{ij})_{M\times K}$, a matrix $B = (b_{ij})_{K\times N}$, and a matrix $C = (c_{ij})_{M\times N}$ as input, and produce a matrix $D = (d_{ij})_{M\times N}$ such that $D = A \times B + C$. The data types of A and B are identical. The data types of C and D are also identical, and their precision is no lower than the precision of A and B.

The numerical behavior of these assembly instructions remains undisclosed. Specifically, the computation of $d_{ij} = c_{ij} + \sum_{k=0}^{K-1} a_{ik} b_{kj}$ is executed in an undocumented way. Through delicate numerical experiments, prior works [9,18] have found that for double-precision input, the computation is executed in a chain of standard FMAs; for low-precision input (specifically, when the precision of A and B is lower than float32), the computation of $d_{ij} = c_{ij} + \sum_{k=0}^{K-1} a_{ik} b_{kj}$ is executed based on multi-term fused summation:

- The products are computed exactly, and the results are maintained in full precision without rounding after the multiplication.
- The summation of a group of summands is performed in a fixed-point manner. Specifically, the significands are aligned to the largest exponent of the summands, and then truncated to 24+ bits (i.e., no less than the precision of float32). The number of bits and the truncation method vary depending on the GPU architecture.
- Then, the sum is converted to the floating-point number in the output data type of the instruction.

Note that the size of the group w, the width of the accumulator, and the detailed conversion method vary depending on the GPU architecture. In addition, previous works do not target the high-level APIs and libraries, so the accumulation orders of them remains unknown.

Our proposed solutions can work for standard FMAs. However, multi-term fused summation requires a new method, be-

cause it is executed in a non-standard, IEEE-754-incompliant way. Specifically, in multi-term fused summation, w summands (e.g., $x_0 = c$, and $x_i = a_{i-1}b_{i-1}$ for $1 \le i < w$) are summed in a fixed-point manner, thus making the result independent of the summation order. To represent this operation in the summation tree, we should use a node with w children instead of a node with two children. Therefore, the summation tree should be an w-way tree.

5.2.2 Constructing the multiway summation tree

To adapt to the multiway tree, we first revisit BasicFPRev in Section 4. The first two steps still work because we find that the key equation $l^{i,j} = n - \text{SUMIMPL}(A^{i,j})$ remains valid in multi-term fused summation. Thus, the values of $l^{i,j}$ can be obtained in the same way, and we only need to redesign the tree construction algorithm in the third step.

Then, we revisit the tree construction algorithm in Algorithm 3. In BUILDSUBTREE(I), we calculate $l^{i,j}$ for a fixed i and all $j \in I - \{i\}$, enumerate them in ascending order, and maintain r as the root of the largest constructed subtree containing node #i. For some $l \in L_i = \{l^{i,j} : j \in I - \{i\}\}$ and $J_l = \{j : j \in I - \{i\} \wedge l^{i,j} = l\}$, the return value of BUILDSUBTREE(J_l) is the subtree with J_l as its leaf nodes. The root of this subtree must be the sibling node of r, so we can create a new node as their parent node and update r. However, this relation is not always true for the multiway tree.

In addition to being sibling node, the root of the subtree may also be the parent node of r in the multiway tree. For example, suppose a 5-way tree with leaf nodes $I = \{0,1,2,3,4\}$ as the children of the root. Then, when $r = 0$, $l = 5$, and $J_l = \{1,2,3,4\}$, solving the subproblem for J_l should return a partial subtree with J_l as its leaves. The root node of the subtree is the parent node of r.

To distinguish the two cases, we observe the return value of BUILDSUBTREE(J_l), denoted by T', and the complete subtree rooted at the root of T', denoted by T_c. In the first case, the root of T' should be the sibling of r, and $T' = T_c$. In the second case, the root of T' should be the parent of r, and $T' \subset T_c$. Therefore, we can compare the size of T' (denoted by $n^{T'}_{\text{leaves}}$) with the size of T_c (denoted by $n^{T_c}_{\text{leaves}}$). We note that $n^{T'}_{\text{leaves}} = |J_l|$, and $n^{T_c}_{\text{leaves}} = \max\{l^{j,k} : j,k \in J_l\} = \max(L_{\min(J_l)})$. Therefore, if $\max(L_{\min(J_l)}) = |J_l|$, then the root of T' should be the sibling of r, so we should create a new node as their parent node and update r with the index of this new node. Otherwise, $\max(L_{\min(J_l)}) > |J_l|$, so the root of T' should be the parent of r, and thus we should add an edge from r to the root of T', and update r with the root.

Through this modification, the multiway tree can be correctly constructed. We elaborate on the above process in Algorithm 4, i.e., the algorithm of FPRev. It has the same time complexity as Algorithm 3 (note that Algorithm 3 just corresponds to the case where $\max(L_{\min(J_l)}) = |J_l|$), and supports multi-term fused summation.

Algorithm 4 The algorithm of FPRev.

Require: Implementation SUMIMPL and the number of summands n
Ensure: Summation tree of SUMIMPL

```
function FPREV(SUMIMPL,n)
    function BUILDSUBTREE(I)
        T ← ∅
        if |I| = 1 then                                   ▷ stop condition
            return (T, 1)
        i ← min(I); L_i ← ∅
        for j ∈ I − {i} do                  ▷ calculate l^{i,j} on demand
            A^{i,j} ← (1,1,...,1); A^{i,j}_i ← M; A^{i,j}_j ← −M
            l^{i,j} ← n − SUMIMPL(A^{i,j})
            L_i ← L_i ∪ {l^{i,j}}
        r ← i                                  ▷ current root of the subtree
        for l ∈ L_i in ascending order do
            J_l ← {j : j ∈ I − {i} ∧ l^{i,j} = l}
            (T', n^{T_c}_leaves) ← BUILDSUBTREE(J)
            T ← T ∪ T'
            if |J_l| = n^{T_c}_leaves then                ▷ T' = T_c
                T ← T ∪ {(r, r+n), (GetRoot(T'), r+n)}
                r ← r+n
            else                                          ▷ T' ⊂ T_c
                T ← T ∪ {(r, GetRoot(T'))}
                r ← GetRoot(T')
        return (T, max(L_i))
    (T, n^{T_c}_leaves) ← BUILDSUBTREE({0,1,...,n−1})
    return T
```

5.3 Time complexity and correctness

Following the same analysis in Section 5.1.3, the time complexity of FPRev is $O(n^2t(n))$ and $\Omega(nt(n))$, and the probability of the worst-case time complexity $O(n^2t(n))$ is low. The correctness of it is also guaranteed by design and can be proven following the same process in Section 4.4.

6 Case study

In this section, we apply FPRev to two prevalent numerical libraries: NumPy [11], the most popular Python library for numerical computing on CPUs, and PyTorch [27], a very popular Python library for numerical computing on GPUs. We successfully identify and analyze the undocumented accumulation orders in these libraries.

6.1 NumPy's implementation on CPUs

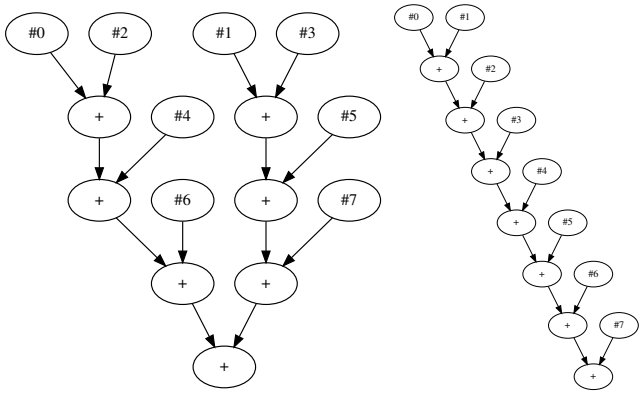
We use FPRev to test NumPy (version 1.26) on three CPUs:

- CPU-1: Intel Xeon E5-2690 v4 (24 v-cores)
- CPU-2: AMD EPYC 7V13 (24 v-cores)
- CPU-3: Intel Xeon Silver 4210 (40 v-cores)

Summation. On these CPUs, we find that NumPy implements identical accumulation order for the summation function in single precision. Therefore, Numpy's summation is verified to be reproducible across these systems, and can be used in software requiring numerical reproducibility.

The accumulation order is sequential for $n < 8$. For $8 \leq n \leq 128$, NumPy implements an eight-way summation. Each way i sums up $a_i, a_{i+8}, a_{i+16}, \ldots$ sequentially, and the sums of eight ways are summed using pairwise summation. For example, Figure 1 shows the accumulation order for $n = 32$. This accumulation order implies that developers can leverage the eight-way SIMD instructions in the CPU to accelerate computation. For $n > 128$, NumPy increases the number of ways, thus leveraging multi-threading for large-scale summation.

Other AccumOps. We also test NumPy's dot product, matrix-vector multiplication, and matrix multiplication functions in single precision. We observe discrepancies in the accumulation order across the tested CPUs. For example, Figure 3 shows the accumulation orders of Numpy's $n \times n$ matrix-vector multiplication for $n = 8$ on the CPUs. We note that on CPU-1 and CPU-2, the 32 products of each output element are accumulated using 2-way summation, whereas on CPU-3, which has more cores than CPU-1 and CPU-2, the products are accumulated sequentially.



(a) On Intel Xeon E5-2690 v4 and AMD EPYC 7V13 (24 v-cores).

(b) On Intel Xeon Silver 4210 (40 v-cores).

Figure 3: The accumulation orders of NumPy's 8×8 matrix-vector multiplication on different CPUs.

In summary, NumPy's summation function is safe for developing reproducible software for these CPUs, while other AccumOps of NumPy should not be used in software requiring numerical reproducibility.

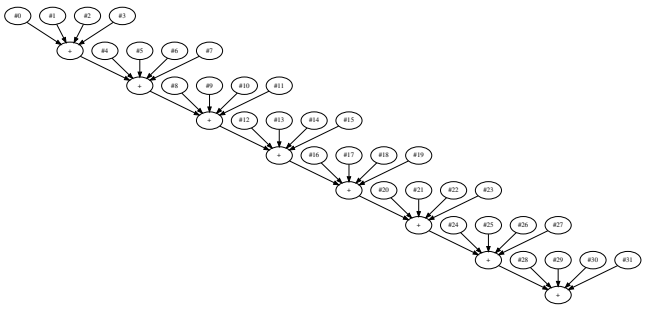
6.2 PyTorch's implementation on GPUs

We use FPRev to test PyTorch (version 2.3) on three GPUs:

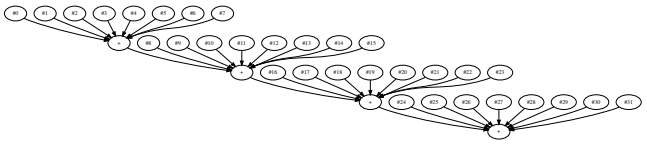
- GPU-1: NVIDIA V100 (5120 CUDA cores)
- GPU-2: NVIDIA A100 (6912 CUDA cores)
- GPU-3: NVIDIA H100 (16896 CUDA cores)

On these GPUs, we observe findings similar to those for NumPy: PyTorch implements identical accumulation orders for the summation function in single precision but not for the BLAS operations. Therefore, PyTorch's summation function is safe for developing reproducible software for these GPUs, while other AccumOps of PyTorch should not be used in software requiring numerical reproducibility.

Matrix multiplication on Tensor Cores. To enable Tensor Core computation, we apply FPRev to half-precision matrix multiplication in PyTorch, which is implemented using the cuBLAS backend. The results show that the summation tree is a 5-way tree on NVIDIA V100, a 9-way tree on A100, and a 17-way tree on H100, corroborating the conclusion in [9, 18], which states that the Tensor Cores on NVIDIA Volta, Ampere, and Hopper architectures use (4+1)-, (8+1)-, and (16+1)-term fused summation respectively. For example, Figure 4 shows the summation trees for $n = 32$ on these devices.



(a) On NVIDIA V100.



(b) On NVIDIA A100.

(c) On NVIDIA H100.

Figure 4: The accumulation orders of PyTorch's half-precision $32 \times 32 \times 32$ matrix multiplication on Tensor Cores.

We also examine the SASS assembly instructions they use, and observe that V100 uses the `HMMA.884` instruction, and both A100 and H100 use the `HMMA.16816` instruction. **Interestingly, an `HMMA.16816` instruction on A100 indicates the shape of the inputs where the accumulation dimension $K = 16$, but it is implemented through (8+1)-term fused summation by the A100 Tensor Core hardware.**

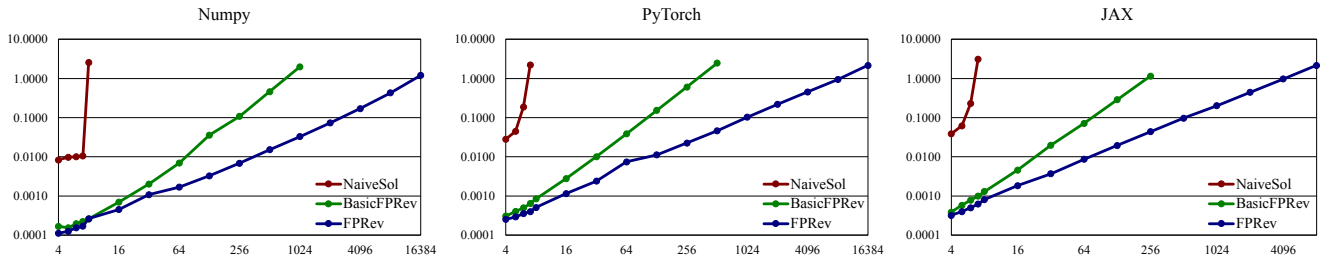


Figure 5: Execution time of applying NaiveSol, BasicFPRev, and FPRev to the summation functions in NumPy, PyTorch, and JAX. The vertical axis represents execution time in seconds. The horizontal axis represents the number of summands n.

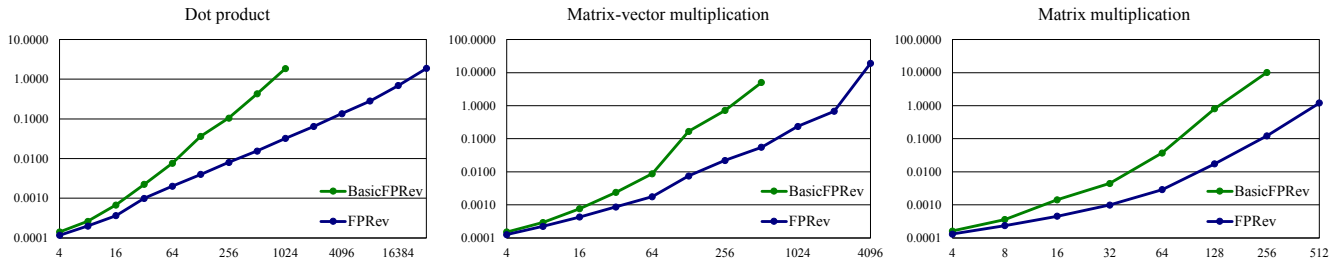


Figure 6: Execution time of applying BasicFPRev and FPRev to the dor product, matrix-vector multiplication, and matrix multiplication functions in NumPy. The vertical axis represents execution time in seconds. The horizontal axis represents the number of summands n.

7 Performance evaluation

7.1 Experiment design

In this section, we evaluate the efficiency of FPRev. Specifically, we aim to answer the following research questions (RQs):

- RQ1: how efficient is FPRev when applied to different libraries?
- RQ2: how efficient is FPRev when applied to different operations?
- RQ3: how efficient is FPRev on different CPUs and GPUs?

To answer the RQs, we measure the execution time (wall-clock time) of applying our solutions to tested libraries and operations. We implement FPRev (Algorithm 4) in Python (version 3.11). For comparison, we also implement the basic solution (Algorithm 2), denoted by BasicFPRev.

7.2 RQ1: How efficient is FPRev when applied to different libraries?

For RQ1, we test the single-precision summation function in three libraries: NumPy (version 1.26) [11], PyTorch (version 2.3) [27], and JAX (version 0.4) [3]. We run the experiments on Intel Xeon E5-2690 v4 with 24 v-cores. In these experiments, we also implement the naive brute-force solution (Section 3.3), denoted by "NaiveSol", to show its extremely low efficiency. For remaining RQs, we omit the naive solution because it is proven to be too inefficient.

We begin with the number of summands $n = 4$, and increment n until the execution time exceeds one second. Each experiment is carried out 10 times, and the arithmetic mean of the 10 results is reported in Figure 5. The red curves indicate that the execution time of NaiveSol grows exponentially as n grows. The results substantiate the $O(4^n/n^{3/2} \cdot t(n))$ time complexity of NaiveSol. The green and blue lines show that the execution time of BasicFPRev and FPRev grows polynomially. The different slopes also demonstrate that the execution time of BasicFPRev is longer than that of FPRev, and increases more rapidly as n increases. This is because the time complexity of BasicFPRev is $\Theta(n^2 t(n))$, while that of FPRev is $\Omega(nt(n))$ and $O(n^2 t(n))$.

These trends suggest that the scalability of BasicFPRev is much better than that of NaiveSol, and the scalability of FPRev is even better. For example, when $n = 16$, NaiveSol can take over 24 hours to produce an output, but BasicFPRev and FPRev only take less than 0.01 seconds. If $n = 8192$, BasicFPRev will take over 100 seconds to produce an output, but FPRev only takes about 1 second.

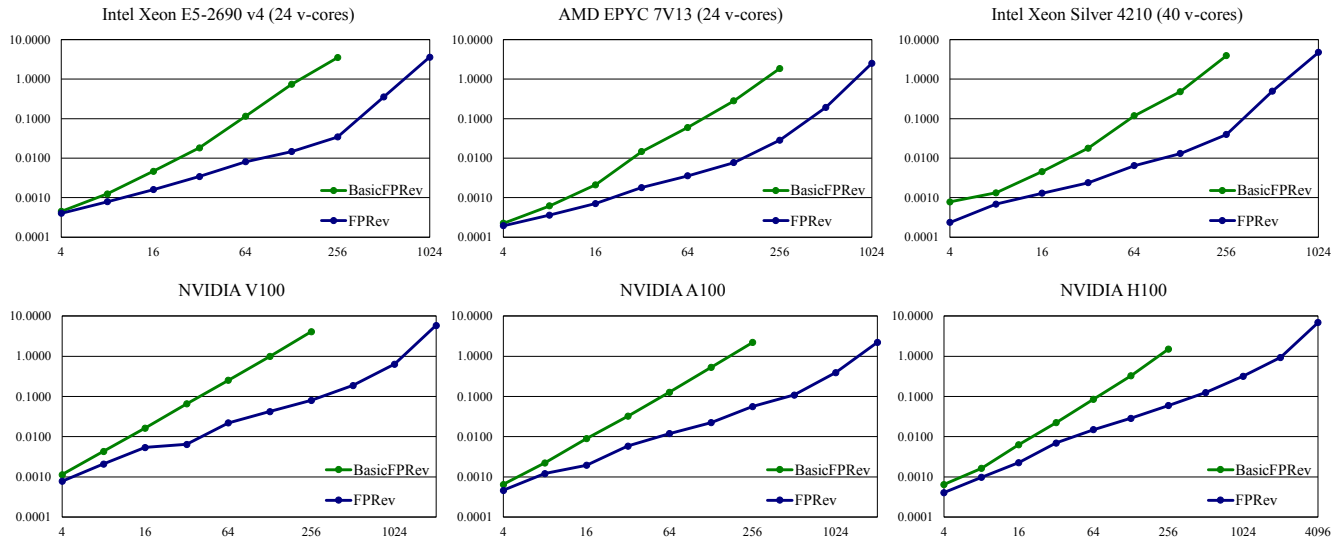


Figure 7: Execution time of applying BasicFPRev and FPRev to the matrix multiplication functions in PyTorch on different CPUs and GPUs. The vertical axis represents execution time in seconds. The horizontal axis represents the number of summands n.

7.3 RQ2: How efficient is FPRev when applied to different operations?

For RQ2, we test the single-precision dot product, matrix-vector multiplication, and matrix multiplication functions in NumPy on Intel Xeon E5-2690 v4. The time complexity of these functions is $O(n)$, $O(n^2)$, and $O(n^3)$, respectively.

Similarly, we begin with $n = 4$, and increment n until the execution time exceeds one second. Each experiment is carried out 10 times, and the arithmetic mean of the 10 results is reported in Table 6. The different slopes indicate that the time complexity of BasicFPRev is higher than that of FPRev. In addition, as the time complexity of the workload increases, the growth speed of the runtime with regard to n accelerates. Therefore, the speedup of FPRev over BasicFPRev is more pronounced as the workload is more complex. For example, for $n = 256$, FPRev is 13.0× as fast as BasicFPRev for dot product, 32.3× for matrix-vector multiplication, and 82.1× for matrix multiplication.

7.4 RQ3: How efficient is FPRev on different CPUs and GPUs?

For RQ3, we test the single-precision matrix multiplication in PyTorch on the CPUs and GPUs listed in Section 6. Similarly, we begin with $n = 4$, and increment n until the execution time exceeds one second. Each experiment is carried out 10 times, and the arithmetic mean of the 10 results is reported in Figure 7. The results demonstrate consistent improvements in the runtime of FPRev compared to BasicFPRev. Therefore, FPRev is consistently more efficient than BasicFPRev on different devices.

8 Discussion

8.1 Limitation and mitigation

8.1.1 Dynamic range of the input data type

When applying FPRev to data types with low dynamic range, the mask M may be too small to effectively mask the sum of ones. For example, the maximum value of the 8-bit floating-point number with 4 exponent bits (FP8-e4m3) defined in [21] is 1.75×2^8, so the condition $M \gg n$ may not hold, and $\pm M + \sigma \neq \pm M$ for $0 \leq \sigma \leq n-2$. To mitigate the issue, we can replace the ones in the masked all-one arrays with smaller numbers (e.g., $2^{-9} \times 2^{-9}$ for FP8-e4m3 matrix multiplication), and scale the sum back to an integer between 0 and $n-2$ when calculating $l^{i,j}$. This solution does not affect efficiency.

8.1.2 Precision of the accumulator

The precision of the floating-point accumulator can limit the input size that FPRev supports. For example, float32 has a precision of 24 bits, so the maximum number of summands (n) that FPRev supports is $2^{24} + 1 = 16777217$ for float32 accumulation operations. For larger numbers, the sum of $n-2$ ones cannot be represented precisely in float32, so the sum of masked all-one arrays may be incorrect. This issue can be mitigated by dynamically replacing the multiple ones corresponding to a constructed subtree with one and multiple zeros, as if compressing the constructed subtree into one node.

Specifically in the BUILDSUBTREE(I) function of FPRev (Algorithm 4), the computation of $\text{SUM}(A^{i,j}) = 0$ is accurate for js such that $l^{i,j} = n$, so we can build the subtree for them in the end. We extract those js to $J = \{j : l^{i,j} = n\}$ after computing $L_i = \{l^{i,j} : j \in I - \{i\}\}$. Then, we set the values at J to 0, and build the subtree for $I - J$ recursively (which results in a smaller subproblem). After the subtree for $I - J$ is constructed, we set the values at J back to 1.0, and set the values at $I - J - \{i\}$ to 0. Now, the constructed subtree (containing $I - J$) is treated as a node, represented by node #i. Next, we run the original tree construction algorithm (the last iteration of the for-loop, i.e., $l = |All|$) for J, and then the whole tree is constructed.

Combining the two mitigation techniques, the modified version of FPRev is shown in Algorithm 5. This version is applicable to data types with low dynamic range and low accumulation precision, such as 16-bit and 8-bit floating-point formats (including BF16, FP16, FP8-e5m2, and FP8-e4m3 on recent Tensor Cores).

Algorithm 5 The modified version of FPRev.

Require: Implementation SUMIMPL, the number of summands n, large value M, and tiny value e
Ensure: Summation tree of SUMIMPL
function MODIFIEDFPREV(SUMIMPL, n, M, e)
  function BUILDSUBTREE(I, All)
    $T \leftarrow \varnothing$
    if $|I| = 1$ **then** ▷ stop condition
      return T
    $i \leftarrow \min(I)$; $L_i \leftarrow \varnothing$
    for $j \in I - \{i\}$ **do** ▷ calculate $l^{i,j}$ on demand
      $A_k^{i,j} \leftarrow e$ for $k \in All$; $A_i^{i,j} \leftarrow M$; $A_j^{i,j} \leftarrow -M$
      $l^{i,j} \leftarrow |All| - \text{SUMIMPL}(A^{i,j})/e$
      $L_i \leftarrow L_i \cup \{l^{i,j}\}$
    $J \leftarrow \{j : l^{i,j} = \max(L_i)\}$
    $A_k^{i,j} \leftarrow 0$ for $k \in J$ ▷ Ignoring J
    $(T, n_{\text{leaves}}^{T_c}) \leftarrow$ BUILDSUBTREE($I - J, All - J$)
    $r \leftarrow \text{GetRoot}(T)$
    $A_k^{i,j} \leftarrow e$ for $k \in J$
    $K \leftarrow I - J - \{i\}$
    $A_k^{i,j} \leftarrow 0$ for $k \in K$ ▷ Treating $I - J$ as $\{i\}$
    $(T', n_{\text{leaves}}^{T_c}) \leftarrow$ BUILDSUBTREE($J, All - K$)
    $T \leftarrow T \cup T'$
    if $|J| = n_{\text{leaves}}^{T_c}$ **then** ▷ $T' = T_c$
      $T \leftarrow T \cup \{(r, r+n), (\text{GetRoot}(T'), r+n)\}$
    else ▷ $T' \subset T_c$
      $T \leftarrow T \cup \{(r, \text{GetRoot}(T'))\}$
    return $(T, \max(L_i))$
  $All \leftarrow \{0, 1, ..., n-1\}$
  $(T, n_{\text{leaves}}^{T_c}) \leftarrow$ BUILDSUBTREE(All, All)
  return T

8.2 Extensibility and future work

Section 6 has demonstrated that FPRev can be applied to popular numerical libraries like NumPy and PyTorch. FPRev can be applied to other accumulation implementations as long as they fall within the scope detailed in Section 3. In practice, we find most popular libraries have deterministic reduction orders and fall into the scope. FPRev also works for accumulation operations in collective communication primitives, such as the AllReduce operation, if their accumulation order is predetermined.

To further extend our tool to other functions based on special summation algorithms, the algorithms must satisfy the property $l^{i,j} = n - \text{SUMIMPL}(A^{i,j})$ or its variant in Algorithm 5. For example, the next generation of Tensor Core will support the microscaling data format [30], including the 4-bit and 6-bit formats MXFP4 and MXFP6. If their dynamic range and accumulator precision permit and the property holds, our methods can reveal the accumulation order within a block of microscaling numbers. Then, we can treat a block as one summand, and use FPRev to construct the summation tree for the summation of the blocks, and then expand each block to a subtree.

In addition to revealing accumulation orders, we plan to extend our methods to detect more floating-point behaviors in matrix accelerators. For example, we can determine the rounding mode and the precision of the accumulator of Tensor Cores by enumerating $n = 1, 2, ...$ and checking the result of $2^n + 1.75 - 2^n$. We are designing more numerical experiments to identify how the block fused multiply-add is conducted. Then, with the information detected, we can model the exact behavior of the hardware matrix accelerators.

Another direction is further optimizing the efficiency of FPRev. For example, we can randomize the selection of $i \in I$ in the FPRev algorithm, as if selecting the random pivot in quick sort. This might reduce the expected time complexity of FPRev.

9 Conclusion

In this paper, we introduce FPRev, a diagnostic tool for revealing the accumulation order in software and hardware implementations through numerical testing. It can help verify and facilitate the development of reproducible software. As a case study, FPRev reveal the undisclosed accumulation orders of prevalent numerical libraries such as NumPy and PyTorch on different CPUs and GPUs. We also demonstrated the efficiency FPRev through experiments covering various implementations and devices. Our source code is available at https://github.com/peichenxie/FPRev, encouraging further investigation and improvement by the research community.

References

[1] Andrea Arteaga, Oliver Fuhrer, and Torsten Hoefler. Designing Bit-Reproducible Portable High-Performance Applications. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1235–1244, 2014. doi:10.1109/IPDPS.2014.127.

[2] David H Bailey, Jonathan M Borwein, and Victoria Stodden. Facilitating reproducibility in scientific computing: Principles and practice. In *Reproducibility: Principles, Problems, Practices, and Prospects*, pages 205–231. Wiley Online Library, 2016. Publisher: Wiley Online Library.

[3] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. JAX: composable transformations of Python+NumPy programs, 2018. URL: http://github.com/google/jax.

[4] Boyuan Chen, Mingzhi Wen, Yong Shi, Dayi Lin, Gopi Krishnan Rajbahadur, and Zhen Ming Jiang. Towards Training Reproducible Deep Learning Models. In *International Conference on Software Engineering (ICSE)*, pages 2202–2214, 2022. doi:10.1145/3510003.3510163.

[5] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Q. Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 578–594, 2018. URL: https://www.usenix.org/conference/osdi18/presentation/chen.

[6] Sylvain Collange, David Defour, Stef Graillat, and Roman Iakymchuk. Numerical Reproducibility for the Parallel Reduction on Multi- and Many-Core Architectures. *Parallel Computing*, 49:83–97, 2015. doi:10.1016/j.parco.2015.09.001.

[7] James Demmel and Hong Diep Nguyen. Fast Reproducible Floating-Point Summation. In *IEEE Symposium on Computer Arithmetic (ARITH)*, pages 163–172, 2013. doi:10.1109/ARITH.2013.9.

[8] James Demmel and Hong Diep Nguyen. Parallel Reproducible Summation. *IEEE Transactions on Computers*, 64(7):2060–2070, 2015. doi:10.1109/TC.2014.2345391.

[9] Massimiliano Fasi, Nicholas J. Higham, Mantas Mikaitis, and Srikara Pranesh. Numerical behavior of NVIDIA tensor cores. *PeerJ Computer Science*, 7:e330, 2021. doi:10.7717/peerj-cs.330.

[10] Hui Guo, Ignacio Laguna, and Cindy Rubio-González. pLiner: isolating lines of floating-point code for compiler-induced variability. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, page 49, 2020. doi:10.1109/SC41405.2020.00053.

[11] Charles R. Harris, K. Jarrod Millman, Stéfan van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585:357–362, 2020. URL: https://doi.org/10.1038/s41586-020-2649-2, doi:10.1038/S41586-020-2649-2.

[12] Yun He and Chris H. Q. Ding. Using Accurate Arithmetics to Improve Numerical Reproducibility and Stability in Parallel Applications. *The Journal of Supercomputing*, 18(3):259–277, 2001. doi:10.1023/A:1008153532043.

[13] Nicholas J. Higham. The Accuracy of Floating Point Summation. *SIAM Journal on Scientific Computing*, 14(4):783–799, 1993. doi:10.1137/0914050.

[14] IEEE. *IEEE Standard for Floating-Point Arithmetic*, 2019. doi:10.1109/IEEESTD.2019.8766229.

[15] Intel Corporation. *Intel Math Kernel Library*. URL: https://www.intel.com/content/www/us/en/developer/tools/oneapi/onemkl.html.

[16] Ignacio Laguna. Varity: Quantifying Floating-Point Variations in HPC Systems Through Randomized Testing. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 622–633, 2020. doi:10.1109/IPDPS47924.2020.00070.

[17] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. Numba: a LLVM-based Python JIT compiler. In *Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC)*, pages 7:1–7:6. ACM, 2015. doi:10.1145/2833157.2833162.

[18] Xinyi Li, Ang Li, Bo Fang, Katarzyna Swirydowicz, Ignacio Laguna, and Ganesh Gopalakrishnan. FTTN: Feature-Targeted Testing for Numerical Properties of NVIDIA & AMD Matrix Accelerators. In *IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*, pages

39–46. IEEE, 2024. URL: https://doi.org/10.1109/CCGrid59990.2024.00014, doi:10.1109/CCGRID59990.2024.00014.

[19] Stefano Markidis, Steven Wei Der Chien, Erwin Laure, Ivy Bo Peng, and Jeffrey S. Vetter. NVIDIA Tensor Core Programmability, Performance & Precision. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS) Workshops*, pages 522–531. IEEE Computer Society, 2018. doi:10.1109/IPDPSW.2018.00091.

[20] Dolores Miao, Ignacio Laguna, and Cindy Rubio-González. Expression Isolation of Compiler-Induced Numerical Inconsistencies in Heterogeneous Code. In *ISC High Performance*, pages 381–401, 2023. URL: https://doi.org/10.1007/978-3-031-32041-5_20, doi:10.1007/978-3-031-32041-5_20.

[21] Paulius Micikevicius, Dusan Stosic, Neil Burgess, Marius Cornea, Pradeep Dubey, Richard Grisenthwaite, Sangwon Ha, Alexander Heinecke, Patrick Judd, John Kamalu, Naveen Mellempudi, Stuart F. Oberman, Mohammad Shoeybi, Michael Y. Siu, and Hao Wu. FP8 Formats for Deep Learning, 2022. arXiv: 2209.05433. URL: https://doi.org/10.48550/arXiv.2209.05433, doi:10.48550/ARXIV.2209.05433.

[22] Ingo Müller, Andrea Arteaga, Torsten Hoefler, and Gustavo Alonso. Reproducible Floating-Point Aggregation in RDBMSs. In *IEEE International Conference on Data Engineering (ICDE)*, pages 1049–1060, 2018. doi:10.1109/ICDE.2018.00098.

[23] NVIDIA. *Parallel Thread Execution ISA*. URL: https://docs.nvidia.com/cuda/parallel-thread-execution/.

[24] NVIDIA Corporation. *cuBLAS: Basic Linear Algebra on NVIDIA GPUs*. URL: https://developer.nvidia.com/cublas/.

[25] OpenAI. *OpenAI Documentation: Reproducible Outputs*. URL: https://platform.openai.com/docs/advanced-usage/reproducible-outputs.

[26] OpenBLAS Contributors. *OpenBLAS: An optimized BLAS library*. URL: https://www.openblas.net/.

[27] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Z. Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Conference on Neural Information Processing Systems (NeurIPS)*, pages 8024–8035, 2019. URL: https://proceedings.neurips.cc/paper/2019/hash/bdbca288fee7f92f2bfa9f7012727740-Abstract.html.

[28] Line Pouchard, Sterling Baldwin, Todd Elsethagen, Shantenu Jha, Bibi Raju, Eric G. Stephan, Li Tang, and Kerstin Kleese van Dam. Computational reproducibility of scientific workflows at extreme scales. *International Journal of High Performance Computing Applications*, 33(5), 2019. doi:10.1177/1094342019839124.

[29] Md Aamir Raihan, Negar Goli, and Tor M. Aamodt. Modeling Deep Learning Accelerator Enabled GPUs. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 79–92, 2019. doi:10.1109/ISPASS.2019.00016.

[30] Bita Darvish Rouhani, Ritchie Zhao, Ankit More, Mathew Hall, Alireza Khodamoradi, Summer Deng, Dhruv Choudhary, Marius Cornea, Eric Dellinger, Kristof Denolf, Dusan Stosic, Venmugil Elango, Maximilian Golub, Alexander Heinecke, Phil James-Roxby, Dharmesh Jani, Gaurav Kolhe, Martin Langhammer, Ada Li, Levi Melnick, Maral Mesmakhosroshahi, Andres Rodriguez, Michael Schulte, Rasoul Shafipour, Lei Shao, Michael Y. Siu, Pradeep Dubey, Paulius Micikevicius, Maxim Naumov, Colin Verilli, Ralph Wittig, Doug Burger, and Eric S. Chung. Microscaling Data Formats for Deep Learning, 2023. arXiv: 2310.10537. URL: https://doi.org/10.48550/arXiv.2310.10537, doi:10.48550/ARXIV.2310.10537.

[31] Sanjif Shanmugavelu, Mathieu Taillefumier, Christopher Culver, Oscar R. Hernandez, Mark Coletti, and Ada Sedova. Impacts of floating-point non-associativity on reproducibility for HPC and deep learning applications. In *Workshops of the International Conference for High Performance Computing*, pages 170–179, 2024. doi:10.1109/SCW63240.2024.00028.

[32] Robert Endre Tarjan and Jan van Leeuwen. Worst-case Analysis of Set Union Algorithms. *Journal of the ACM*, 31(2):245–281, 1984. doi:10.1145/62.2160.

[33] Michela Taufer, Omar Padron, Philip Saponaro, and Sandeep Patel. Improving numerical reproducibility and stability in large-scale numerical simulations on GPUs. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1–9, 2010. doi:10.1109/IPDPS.2010.5470481.

[34] Philippe Tillet, Hsiang-Tsung Kung, and David D. Cox. Triton: an intermediate language and compiler for tiled neural network computations. In *ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, pages 10–19. ACM, 2019. doi:10.1145/3315508.3329973.

[35] Oreste Villa, Daniel Chavarria-Miranda, Vidhya Gurumoorthi, Andrés Márquez, and Sriram Krishnamoorthy. Effects of floating-point non-associativity on numerical computations on massively multithreaded systems. In *Cray User Group Meeting (CUG)*, volume 3, 2009.

A Artifact Appendix

Abstract

The repository includes the source code of FPRev and the source code for reproducing the experiments of the paper. The following content includes main claims that can be verified via experiments, and instructions to reproduce the experiments.

Scope

The main claims include:

1. FPRev is functional to reveal the floating-point accumulation orders in common implementations. This claim is detailed by Section 6 "Case study" in the paper. To verify this claim, run `python experiments/casestudy.py` and check the output files in the `outputs` directory.
   (a) `outputs/Numpy*.pdf` represents the revealed accumulation orders for NumPy, as discussed in Section 6.1 "NumPy's implementation on CPUs". Among them, `outputs/NumpyGEMV8.pdf` corresponds to Figure 3 of the paper, if the CPU models are consistent to those in the paper.
   (b) `outputs/Torch*.pdf` represents the revealed accumulation orders for PyTorch, as discussed in Section 6.2 "PyTorch's implementation on GPUs". Among them, `outputs/TorchF16GEMM32.pdf` corresponds to Figure 4 of the paper, if the GPU models are consistent to those in the paper.
2. FPRev is efficient. This claim is detailed by Section 7 "Performance evaluation" in the paper. To verify this claim, run `python experiments/rq1.py`, `python experiments/rq2.py`, and `python experiments/rq3.py`, and check the output files in the `outputs` directory.
   (a) `outputs/rq1.csv` provides the results of Section 7.2 "RQ1: How efficient is FPRev when applied to different libraries?". It corresponds to Figure 5 if the CPU model is consistenst to that in the paper.
   (b) `outputs/rq2.csv` provides the results of Section 7.3 "RQ2: How efficient is FPRev when applied to different operations?". It corresponds to Figure 6 if the CPU model is consistenst to that in the paper.
   (c) `outputs/rq3.csv` provides the results of Section 7.4 "RQ3: How efficient is FPRev on different CPUs and GPUs?". It corresponds to Figure 7 if the CPU and GPU models are consistenst to those in the paper.

Contents

Installation

```
sudo apt install graphviz
git clone https://github.com/peichenxie/FPRev.git
cd FPRev
pip install .
pip install -r experiments/requirements.txt
```

Running experiments

- To reproduce the results in Section 6 (Case study), run `python experiments/casestudy.py` on different hardware models.
- To reproduce the results in Section 7.2 (RQ1: How efficient is FPRev when applied to different libraries?), run `python experiments/rq1.py`.
- To reproduce the results in Section 7.3 (RQ2: How efficient is FPRev when applied to different operations?), run `python experiments/rq2.py`.
- To reproduce the results in Section 7.4 (RQ3: How efficient is FPRev on different CPUs and GPUs?), run `python experiments/rq3.py` on different hardware models.

Then, check the output files in the outputs directory. See `outputs/README.md` for more information.

Hosting

https://github.com/peichenxie/FPRev

Requirements

The artifact requires the following platform:

- GPU: NVIDIA V100 or newer
- OS: Ubuntu 22.04
- Software: Python (version 3.11)

If you wish to reproduce the results in the paper, please use the identical CPU and GPU models:

1. CPU: Intel Xeon E5-2690 v4 (24 v-cores), GPU: NVIDIA V100 (5120 CUDA cores)
2. CPU: AMD EPYC 7V13 (24 v-cores), GPU: NVIDIA A100 (6912 CUDA cores)
3. CPU: Intel Xeon Silver 4210 (40 v-cores), GPU: NVIDIA H100 (16896 CUDA cores)

Inferring Likely Counting-related Atomicity Program Properties for Persistent Memory

Yunmo Zhang
City University of Hong Kong

Junqiao Qiu
City University of Hong Kong

Hong Xu
The Chinese University of Hong Kong

Chun Jason Xue
MBZUAI

Abstract

Persistent Memory (PM) technologies provide fast, byte-addressable access to durable storage but face crash consistency challenges, motivating extensive work of testing and verification of PM programs. Central to PM testing tools is the specification of program properties for object persistence order and atomicity. Although several methods have been proposed for inferring PM program properties, most focus on ordering properties, offering limited support for atomicity properties. This paper explores a class of important atomicity properties between the container-like arrays and their logical size variables, referred to as the counting correlation, which are common in PM programs but exceed the capability of existing approaches. We propose invariants to capture the necessary behaviors of counting-correlated variables, utilize symbolic range analysis to extract PM program behaviors, and encode them into SMT constraints. These constraints are checked against the invariants to infer likely PM program properties. We demonstrate the utility of the inferred properties by leveraging them for PM bug detection, which discovers 14 atomicity bugs (including 11 new bugs) in real-world PM programs.

1 Introduction

DRAM technology is facing significant challenges in density scaling and power leakage [37]. Alternatively, Persistent Memory (PM) technologies, such as Intel Optane [18] and CXL-SSD [2, 35], open a new paradigm that allows for byte-addressable interfaces like DRAM and avoids the overhead of the storage stack. However, PM-based systems are error-prone due to the crash consistency issue. A store to PM is first written to a volatile cache and then flushed to the durable media, during which data is reordered [32, 33]. Programmers need to explicitly use hardware instructions like `clflush` and `sfence`, or transaction interfaces provided by libraries to ensure crash consistency. Yet appropriate usage of these instructions or interfaces demands expertise, beyond the reach of most programmers. As a result, PM programs are found to have many crash consistency bugs by emerging PM testing tools [9, 11, 15, 24, 30].

Due to the explosive program state and the arbitrary crashes that need to be explored, most existing PM testing tools rely on manual annotation [5, 5, 16, 23, 24], which can be tedious, or PM *program properties* specified by developers [9, 30] to inject crashes in a way that efficiently triggers potential bugs. Consequently, there is a persistent need for specifying PM program properties to detect crash consistency violations. To relieve the burden of programmers, PM program property inference approaches, such as Witcher [9] and Huang et al. [15], have been proposed to infer likely program properties based on control and data dependencies. While these methods show promise in *ordering properties*, they face challenges when dealing with the diverse *atomicity properties* of common PM data structures.

This paper examines a crucial category of atomicity properties that pertain to the relationship between container-like PM data structures and the integer variables that represent their semantic sizes–a relationship we refer to as *counting*. For instance, in a tree node, the children array is counting-correlated with its length variable, which records the number of appended pointers to child nodes. Without ensuring atomic persistence of updates to the children array and its length variable, the crash could result in the program reading a dangling pointer or losing ingested data. This counting correlation is also present in other persistent data structures and PM-based systems, such as hash tables [22, 28, 41] and ring buffers [14, 36, 39]. Unfortunately, systematically inferring properties related to counting correlation is out of reach of dependency-based atomicity property inference approaches, which rely on assumptions about access patterns rather than directly analyzing the program semantics of counting.

This paper studies the inference approach of the counting-related atomicity properties, aiming to complement the existing atomicity inference methods and thereby expand the scope of violations that the PM testing tools can detect. However, identifying the counting correlation among variables is not as

straightforward as its simple definition suggests. The semantic sizes, as a logical concept derived from the programmer's intent, are not explicitly revealed through program behavior. To handle it, we first propose the invariants for describing the necessary behaviors of counting-correlated variables based on their access ranges. We leverage symbolic range lattice [29] to extract these behaviors from the program, then encode them into SMT (Satisfiability Modulo Theories) [4] constraints, which are checked against the invariant with an SMT solver, thereby validating the likely counting-related atomicity properties. To show the effectiveness of the proposed inference approach, we leverage the inferred properties to detect PM bugs and successfully detect 14 atomicity violations, including 11 previously unknown bugs.

In summary, this paper makes the following contributions.

- We identify an important class of atomicity properties, termed counting, which are crucial for ensuring the crash consistency of PM programs. We also propose invariants to capture the behaviors of counting-correlated variables.
- We propose a framework that infers the counting correlation in PM programs by combining the symbolic range analysis with SMT constraint solving to validate the invariant.
- We evaluated the effectiveness of our approaches on real-world PM programs and found 14 atomicity bugs.

2 Background and Motivation

2.1 Background

Applications accessing PM could directly perform byte-addressable load/store operations, instead of system calls, as with traditional storage devices. Store requests to PM are first buffered in a volatile cache and then wait an arbitrary amount of time before the cacheline containing the data is evicted to PM. As a result, the reordering of writes to different memory locations may occur [34], causing the data persistence order may differ from the original execution order in the program. When a crash occurs, durable data in PM is prone to unexpected inconsistency.

PM programming delegates the task of ensuring crash consistency to the programmers, by using low-level instructions like `clflush` (or `clwb`) to flush cache lines and `sfence` to enforce the order, as well as high-level transaction (TX) interfaces [17, 19, 38] to provide all-or-nothing semantics. Despite these explicit flush and fence instructions, PM programs suffer from crash consistency issues [5, 9, 24, 30], including correctness bugs (e.g., improper or missed flush/fence) and performance bugs (e.g., extra flush/fence). This work focuses on the correctness issues.

2.2 Related Work

While earlier PM testing tools discussed the detection of durability bugs (i.e., missing flush for a particular store)–the simplest form of correctness bug in PM programs [20, 24, 30], recent works focus more on discovering application-specific bugs that require an understanding of the program logic. In particular, application-specific testing frameworks are designed to detect (i) *ordering bugs* that violate the must persist-before requirements between writes, i.e., without properly using fences, and (ii) *atomicity bugs* that violate the must persist-together requirements of writes, i.e., without enclosing the writes in a transaction.

The main challenge faced by application-specific PM testing tools is the explosive PM state space [12, 20, 30] due to the arbitrary write reordering and crash point in PM programs. To overcome it, existing tools use manual annotation [5, 5, 16, 23, 24], known bug pattern [10, 30], or in a more general way, PM *program properties* specified by developers [9, 30] to inject crashes such that potential bugs are triggered efficiently.

However, relying on manual efforts to specify the PM properties is time-consuming and demands expert knowledge of program semantics and the PM hardware model, making it not within reach of most developers. Although a black-box approach [11] has been proposed to bypass the need for program properties by focusing on the recovery procedure–specifically, examining whether the crashed program states are recoverable–the recovery code itself is often buggy and unreliable [24]. As a result, existing PM testing tools still have a common requirement for PM program properties to identify violations effectively.

To help programmers specify PM program properties, Witcher [9] and Huang et al. [15] propose to infer likely program properties based on the control dependency and data dependency patterns. For example, from a single control dependency relationship `if(x) then y=0`, where the write to `y` is control dependent on the read from `x`, an ordering PM program property is inferred that a store of `x` should be persisted before an update of `y` to ensure crash consistency. Through static analysis or dynamic analysis of data and control dependencies in program traces, Witcher and Huang et al. have demonstrated their ability to infer several ordering and atomicity PM program properties and detected potential bugs by checking whether these properties are satisfied.

Limitations. While dependency analysis can easily infer a *persist-before* requirement from a single data or control dependency relationship, it is limited to capturing the *persist-together* requirements for common data structures in PM programs. Table 1 presents the control dependency patterns that two existing works rely on to infer the potential atomicity properties in PM programs. Witcher observes that the updates of two or more guardians should be persisted atomically; the guardian is the flag variable at the condition statement controlling the access to other variables, e.g., `x` in `if(x)`. In contrast, Huang et al. regard a set of guardians that guard each other as atomic variables. The logic is that since each guarding relationship produces an ordering requirement, the

	Witcher [9]	Huang et al. [15]
Dependency Pattern	`if(x) then m···` $(m \xrightarrow[dep]{ctrl} x)$ `if(y) then n···` $(n \xrightarrow[dep]{ctrl} y)$	`if(x) then y···` $(y \xrightarrow[dep]{ctrl} x)$ `if(y) then x···` $(x \xrightarrow[dep]{ctrl} y)$
Inferred Likely Atomicity Property	ATOMICITY(x, y)	ATOMICITY(x, y)

Table 1: Existing dependency-based atomicity PM program property inference approaches.

only way to reconcile the conflicting requirements between interdependent variables is to persist them together.

Consider an atomicity bug found by our work from P-ART [22], a persistent adaptive radix tree (ART), shown in Figure 1 (b). In this code, a `children` array that collects the valid elements from the node_4 of ART, along with a variable `childrenCount` that indicates the size of valid elements in `children`, are expected to update atomically. Failing to ensure atomicity upon crash could result in irrecoverable data loss if only the array update is persisted, or the return of stale values if only the size update is persisted.

However, neither of the aforementioned dependency patterns could be identified in the accesses to `children` and `childrenCount`. In Figure 1 (a), `childrenCount` guards the access to `children`, but the testing tools fail to detect any instances where `children` guards `childrenCount` (to form the pattern in Huang et al.) or other variables (to form the pattern in Witcher). Unlike a single variable acting as a guardian, the `children` array functions as a *container* with different semantics. It stores multiple user data and ensures their durability in PM to serve future queries. These container-like structures are common in PM use cases, such as database indexes [22, 27, 28, 41] and file systems [7, 40]. Without a guardian role, these structures are hardly captured by the dependency analysis to infer their related atomicity properties.

Besides, work on detecting atomicity violations in concurrent programs [25, 31], such as MUVI [25], might potentially uncover certain counting-related bugs. However, these methods lack the capability for systematic detection and employ runtime techniques that are not straightforward for adoption in PM. For instance, controlling thread interleaving [31] is challenging in the context of PM due to the costly PM writes, as well as arbitrary crashes, which alter the interleaving space.

2.3 Counting-related Atomicity Property

We now introduce the counting relationships between container-like structures and their size attributes commonly found in PM programs. These relationships form a new atomicity property pattern, complementing existing dependency-based ones. Formally, an integer variable *SZ* tracks a numerical value about the *logical size*[1] of one or multiple arrays *ARR*s in the following three scenarios:

[1]The physical size could be regarded as a special case of the logical size.

```
73  Tree::lookupRange (…) {
74      ...
99      for (uint32_t i = 0; i < childrenCount; ++i) {
100         const N *n = std::get<1>(children[i]);
101     ...}
    }                          (a) Tree.cpp

109 N4::getChildren (…) {          They should be
116     ...                        in the same TX
117     children[childrenCount] = std::make_tuple(key, child);
118     childrenCount++;
119     ...
    }                          (b) N_4.cpp
```

Figure 1: An example atomicity bug outside the reach of dependency-based inference approaches.

- the logical size of an array, and
- the sum of the logical sizes of multiple arrays, and
- the complementary size of an array to a constant.

If a *SZ* and one or more *ARR*s exhibit any of the above relationships, an atomicity property ATOMICITY(*SZ*,*ARR*) should be satisfied in PM programs to ensure the crash consistency.

Despite their intuitive semantics, counting-related atomicity properties are often violated in many software storage stacks, which are among the primary targets PM is expected to reshape. For example, in btrfs, `i_size` that indicates the file size was found to be inconsistent with the actual file size when appending data. This inconsistency arises when the log of `i_size` is delayed as improper flags are set by other operations, and then a fsync flushes the data with the value of `i_size` smaller than the actual data size [26]. And in ext4, `i_disksize` indicating the size of data stored on disk is found inconsistent with the real data size when a delayed end-of-file write back meets a crash while a direct non-end-of-file write succeeds [13]. Although only a limited number of atomicity PM bugs are found due to the lack of testing tools aside from Witcher [9] and Huang et al. [15], there is a counting-related PM atomicity bug [6]. In this bug, `size` that indicates the allocated size of an array becomes inconsistent with the array when a crash happens between their updates.

3 Property Inference

3.1 Overview

This work aims to infer counting-related atomicity properties from PM programs. Rather than relying on *causation relationships*, as dependency-based inference approaches do, we resort to a *correlation relationship*, in which variables maintain a semantic relation throughout the program. We expect our approach to complement the work of Witcher [9] and Huang et al. [15], thereby broadening the range of inferred atomicity properties and related bugs covered by testing tools.

Problem. Although the concept of counting is intuitive, discovering the counting correlation is not straightforward. As a

programmer's intent, an array's logical size, which indicates the number of valid elements in its allocated space, is unknown to the analysis tools. Furthermore, the variable that represents the array's logical size may not consistently align with the array's true size throughout the program's execution. Consider an array insertion example in List 1. After executing lines 2 to 5, which insert an element into the middle of the `array`, the real size of `array` after line 2 is `size+1`, different from the value of `size` until its update at line 6. Here, `size` is intended to be counting-correlated with `array`. But the definitions of counting cannot be used to identify counting-correlated variables.

```
1  // insert an element at position p
2  for(i = size - 1; i >= p; i--){
3      array[i + 1] = array[i];
4  }
5  array[p] = 20;
6  size += 1;
```

Listing 1: Array insertion example (`0 <= p < size`).

Main Idea. Our solution to the above problem is based on our observation that *the read access to the array must remain within the range of valid elements constrained by the real size*, where the range of valid elements is accessible by analyzing program behaviors. As a result, we propose that the likely PM atomicity property ATOMICITY(`array, size`) is identified if any read access to `array` could be proved to be restricted by `size`. We refer to this as the *read range invariant*. Later, we will show why write access to the array may be beyond the valid element range and how to leverage this fact to form an (optional) *write range invariant*.

Inference Approach. To infer the counting-related atomicity property, we need to discover the counting-correlated variable pairs that satisfy the above access range invariants. A possible solution is to employ dynamical analysis such as Daikon [8], which monitors the indices of all pointer accesses and the values of all integers, subsequently verifying each pair in traces against the invariant individually. However, due to the inequalities in access range invariants, dynamic analysis often yields highly inaccurate results and is hard to scale in the presence of loops. Instead, we resort to Symbolic Range Analysis (SRA) [29], which over-approximates the access ranges for array indices and offers a more reliable analysis of program behaviors.

3.2 Invariants for Counting Correlation

Since the definition of counting about the logical/real size of an array does not characterize the real behaviors of correlated variables, we propose two access range invariants as the necessary conditions to infer likely counting-related atomicity properties. The invariant refers to the assertion about the correlated variables that are always held to be true in the program. We denote the program with P in this section.

Invariant 1: Read Range. We observe that a read access, which incorporates the programmer's intent for acquiring the valid elements of a container, has its index always lie in the region restricted by the container's logical size. Thus, we propose to use read range invariants to capture the relationship between the access behavior to an array and its counting-correlated size variable. We first introduce the invariant for the first correlation pattern, which is the base of the two other correlation patterns, as shown below:

$$\forall \rho \in P,\ Read_\rho(ARR, idx) \implies idx < Val_\rho(SZ). \quad (1)$$

$Read_\rho(ARR, idx)$ indicates there is a read to ARR at line ρ with access index being idx, and $Val_\rho(SZ)$ is the value of the logical size variable SZ at line ρ. We assume $idx \geq 0$. Consider the example in Listing 1, where the program shifts all elements starting from the insertion position towards the right, the index of read to `array` is $[p, size-1]$, which could be proved to be contained by the range $[0, size)$.

We also motivate the access range invariants by demonstrating an alternative class of correlation-based atomicity invariants based on access-together-times (MUVI [25]). This invariant is originally designed for testing concurrent programming atomicity bugs, in which accessed-together times exceeding a threshold should be updated atomically. From Listing 1, we notice the access-together-times invariant does not hold unless a tiny threshold is set, since this piece of code accesses `array` $size - p + 1$ times, while accessing `size` once.

By leveraging the basic read range invariant defined in Eq. (1), our approach could be extended to handle the two additional correlation patterns related to the logical size, given certain specifications from users. In the second pattern, the variable SZ_s tracks the cumulative logical sizes of multiple arrays. Provided N, the number of arrays considered by the user, the read range invariant is:

$$\forall \rho \in P, \sum_{i \in [1,N] \wedge Read_\rho(ARR_i, idx_i)} idx_i < Val_\rho(SZ_s). \quad (2)$$

Here, the sum of the indices of accesses to a number of ARRs at line ρ is compared against the integer value of $Val_\rho(SZ_s)$, enabling the inference of the cumulative size variable by programmer intent. We set $N = 2$ as the default configuration. In the third correlation pattern, the variable SZ_c is used to represent the complementary size of the array relative to a constant. Provided the constant C from the user, the read range invariant under this scenario is:

$$\forall \rho \in P,\ Read_\rho(ARR, idx) \implies idx < C - Val_\rho(SZ_c). \quad (3)$$

Here, the read access index is checked against the complementary value $C - Val_\rho(SZ_c)$.

We use the read index to establish the range invariant because writes may exceed the region restricted by the logical size variable, e.g., during insertion. In Listing 1, the write

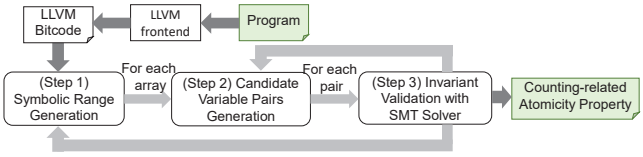


Figure 2: The workflow of our work.

indices to `array` are within the range $[p, size]$. As the insertion extends the logical size at the final step, writes access the index outside of the existing valid element region intended to be represented by `size` to include new elements.

Invariant 2: Write Range. According to the above observation regarding the write behaviors of arrays in programs, the write access range cannot be directly used for inferring the programmer's intent concerning the logical size of arrays. However, we can refine this approach by relaxing the condition on the maximum allowable write index. This adjustment allows us to incorporate an optional invariant, thereby enhancing the accuracy of our inferences. For example, under the assumption that each insertion function only adds one element, the write range invariant for the first pattern is:

$$\forall \rho \in P,\ Write_{\rho}(ARR, idx) \implies idx \leq Val_{\rho}(SZ), \quad (4)$$

where $Write_{\rho}(ARR, idx)$ indicates there is a write to ARR at line ρ with index idx. A constant term c could be added after the integer variable value to allow for different index relaxation limit assumptions. We set c as 0 by default.

3.3 Atomicity Property Inference

The workflow of inferring counting-related atomicity properties is shown in Figure 2. Our tool accepts the programs in C and translates them into LLVM bitcode IR [21]. We employ a symbolic range lattice [29] to abstract the symbolic access indices (**Step 1**) and identify candidate variable pairs for invariant validation (**Step 2**). Finally, we encode the symbolic information of array indices and the integer variables into SMT constraints and use an SMT solver to verify whether the invariants are satisfied (**Step 3**). For each variable pair that satisfies the invariant, an atomicity property is generated.

Step 1. Symbolic Range Generation. Before generating symbolic accesses to arrays, we first prepare all the candidate arrays. We use an LLVM pass to identify all pointer variables, including those that are members of structures. Since C conflates arrays with pointers, we differentiate the array variables from regular pointers by checking the pointer's operand. We collect the pointers that represent arrays into the set $\{ptr\}$. For each ptr, the symbolic range analysis is then employed to compute symbolic bounds for all accesses to ptr. Formally, the symbolic range of an access index int_v, denoted as $[\![int_v]\!]$, consists of a pair of symbolic bounds denoted by $[l, r]$, where

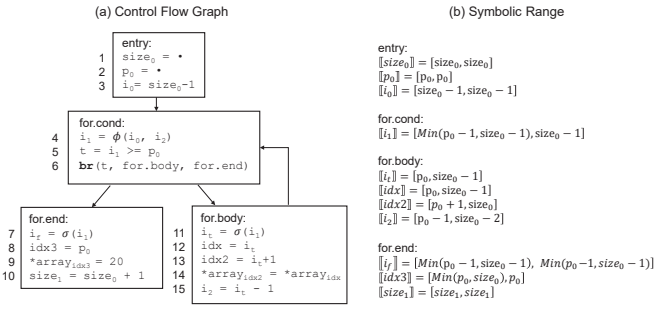


Figure 3: The control flow graph and the symbolic ranges of integer variables in Listing 1.

$l \leq r$. Here, l represents the lower bound, denoted as $[\![x_v]\!]_{\downarrow}$, and r represents the upper bound, denoted as $[\![x_v]\!]_{\uparrow}$. If the symbolic value is undecidable, the range is $[None, None]$.

Example. We use the program in Listing 1 to illustrate the computed symbolic ranges of integer variables, including the access indices. We show the control flow graph in Figure 3 (a). The program is transformed to Extended SSA form [1] before the analysis, where a ϕ-function is created when the value of a variable depends on multiple predecessor blocks, and a σ-function is a copy created for the splitting at a conditional. Before the analysis, all integer variables are initialized with a degenerate-interval symbolic range, i.e., $[\![\cdot]\!] = [\cdot, \cdot]$. The program inputs, which include the potential integer variables, are treated as symbols during SRA, also known as the symbolic kernel [29], to compute other symbolic ranges.

SRA computes the symbolic ranges of integer variables as follows. The assignment instructions after reads rewrite both bounds of the symbolic range, e.g., i_0 is assigned $[\![size_0]\!] - 1 = [size_0 - 1, size_0 - 1]$. Besides, the ϕ-function computes a union of two symbolic ranges from two blocks. In Line 4, $[\![i_1]\!] = [Min([\![i_0]\!]_{\downarrow}, [\![i_2]\!]_{\downarrow}), Max([\![i_0]\!]_{\uparrow}, [\![i_2]\!]_{\uparrow})]$ And σ-function computes the interval intersection of the operand's symbolic range and the condition. For example, i_t is in the condition $i_1 \geq p_0$ satisfied branch; its symbolic value becomes $[Max(p_0, [\![i_1]\!]_{\downarrow}), [\![i_1]\!]_{\uparrow}]$. Following these rules, SRA converges after iterations and finally yields the results in Figure 3 (b). Note that the symbolic range results provide over-approximation of the variable ranges [29], which means the actual values during program execution must fall within these ranges.

Step 2. Candidate Variable Pairs Generation. The next step is generating the candidate variable pairs to be checked against the counting invariants. The naive method enumerates all integers to form a pair with a candidate array. Instead, we only select the integer variables that appear in the symbolic range of any accesses to ptr. Specifically, we collect all integer-type IR instructions but exclude the integers unrelated to the computed symbolic ranges into the set $\{int\}$. The candidate variable pairs are $\{(ptr, int)\} = \{ptr\} \times \{int\}$. We

PM Programs	Description	Versions
P-ART [22]	Persistent Adaptive Radix Tree	f0b891a
P-BwTree [22]	Persistent BwTree	f0b891a
CCEH [28]	Dynamic Hashing for PM	b62a9c8
Level-Hashing [41]	Hash Indexing for PM	28eca31

Table 2: Tested PM Programs.

then validate the invariant for each candidate pair.

Step 3. Constraint-based Invariant Validation. To determine whether an integer variable *int* is counting-correlated with *ptr*, we encode *int* and the accesses to *ptr* into the SMT constraints of the invariant, e.g., the invariants shown in Eq. (1) and Eq. (4) for the first correlation pattern. Then, the SMT solver is used to check if the constraints are *always* satisfied, representing that the accesses to *ptr* are restricted by *int* across user (function) inputs. Specifically, for each access index *idx* to *ptr*, an SMT constraint $[\![idx]\!]_{\uparrow} < [\![int_v]\!]_{\downarrow}$ is built for read invariant, and $[\![idx]\!]_{\uparrow} \leq [\![int_v]\!]_{\downarrow}$ is built for write invariant. For the read index range that is undecidable or has an infinite value in any of the bounds, we directly add `False` to the constraints set. Then we make the conjunction of constraints for all accesses to form the invariant constraint.

Taking the first counting correlation pattern as an illustrative example, if we let all SSA forms of *int* constitute V_{int} and the read indices to *ptr* forms U_{ptr}, a read invariant INV is:

$$\bigwedge_{idx \in U_{ptr}} \bigvee_{v \in V_{int}} [\![idx]\!]_{\uparrow} < int_v. \tag{5}$$

Here, the disjunction among multiple SSAs of *int* ensures that the indexes are checked against a specific assignment of *int*. Thus, we don't need to distinguish which assignment manually. An SMT solver Z3 [3] is utilized to check whether the constraints are satisfied *across all possible values of int*. Since SMT solvers only find a specific satisfaction for given constraints, we make the negation of Eq. (5) (¬INV) and ask the solver to prove its UNSAT, i.e., no values could be found to make the invariant unsatisfied. If the proving is successful, a likely counting-related atomicity property ATOMICITY(*ptr*, *int*) is generated.

To encode the invariant in the second correlation pattern, the sum of indices to each accessed ptr_i is checked. Similarly, encoding the invariant in the third scenario requires only replacing an SSA of the integer variable int_v with $C - int_v$.

4 Evaluation

4.1 Methodology

The program analysis stage of our work is implemented as LLVM [21] compiler passes with a symbolic range analysis pass modified from the open-source Nazaré et al. [29] [2]. The invariant validation is implemented in Python with Z3 [3] as the SMT solver.

Experimental Setup. We conduct all experiments on a machine equipped with 2× Intel Xeon Gold 5317 CPUs, 128GB DRAM, and 512GB Intel Optane DC Persistent Memory. The PM consists of four 128GB Optane memory through interleaved mode. To demonstrate the utility of our counting-related atomicity property inference approach, we leverage the inferred properties to detect PM bugs. For each identified property, ATOMICITY(`x`,`y`), we confirm the atomicity violations by manually checking whether the updates of `x` and `y` are protected by a transaction, e.g., between TX_BEGIN and TX_END of `libpmemobj` [17]. Integrating our property inference approach into an established property-checking procedure to confirm bugs through trace analysis (e.g., Huang et al. [15]) is left in our future work. Table 2 shows the tested real-world PM programs in this evaluation, including popular PM data structures such as hash tables (CCEH and Level-hashing) and trees (P-ART and P-BwTree).

4.2 Results

Table 3 summarizes 14 atomicity violations in the tested problems, where 11 of them are new bugs, and the rest have been reported [9]. Among the detected bugs, 4 bugs (#10, #12, #13, and #14) violate the atomicity properties between an array and its correlated allocation size (a special case of logical size), producing a fault when the post-crash program reads the unallocated memory space. All the rest of the bugs violate the atomicity property between an array with its logical size. The impact of these bugs varies due to the diverse element types within the array. Bug #8 involves a value array failing to update atomically with its logical size, causing invalid reads or user data loss after a crash. In other bugs, the pointer arrays are not atomically updated with the integer variable indicating the number of valid pointers, leading to stale pointer reads in post-recovery programs. These different impacts of atomicity violations highlight the varied scenarios where an atomicity bug can occur.

Next, we compare our tool with a correlation inference tool MUVI [25]. Since Huang et al. [15] relies on the dual-dependent relationship that does not exist in any pair of array and integer, we compare with another dependency-based atomicity property inference approach Witcher [9]. Among the 14 bugs detected by our tool, MUVI successfully identified 4 of them, while Witcher identified 3. MUVI works effectively when the array and the size variable are closely tied to each other. We observe that this detection strategy struggles to identify counting-correlated variables in two main scenarios. First, when an array has more than one size variable, e.g., one for allocated size and one for logical size, the frequency of access to the array and its size variable tends to be random. Second, when the elements of the array are structures with multiple variables that might be accessed individually, the program

[2] https://github.com/henry-nazare/llvm-sra

PM Program	ID	New	Code	Description	Impact	MUVI	Witcher
P-ART	1	✓	N4.cpp:117	Creating an array of valid nodes	Fault or data loss		
	2		N4.cpp:22	Inserting a node to an array of children nodes	Fault or data loss		✓
	3	✓	N16.cpp:124	Creating an array of valid nodes	Fault or data loss		
	4	✓	N48.cpp:120	Creating an array of valid nodes	Fault or data loss		
	5	✓	N256.cpp:81	Creating an array of valid nodes	Fault or data loss		
	6		N16.cpp:13	Inserting a node to an array of children nodes	Fault or data loss		✓
	7	✓	Epoch.cpp:57	Adding to an array of fixed size arrays	Fault or data loss	✓	
P-BwTree	8	✓	bloom_filter.h:143	Inserting an element to a "ValueType" array	Stale read or data loss	✓	
CCEH	9	✓	CCEH_LSB.cpp:220	Resizing an array before insertions.	Fault or data loss	✓	
	10	✓	linear_probing.cpp:151	Resize a hash table	Memory corruption		
	11	✓	extendible_hash.cpp:329	Resizing an array before insertions.	Fault or data loss	✓	
	12	✓	cuckoo_hash.cpp:295	Resizing a "table" array	Memory corruption		
Level-Hashing	13		level_hashing.c:112	Expanding a level hash table	Memory corruption		✓
	14	✓	level_hashing.c:226	Shrinking a level hash table	Corruption or data		

Table 3: Atomicity bug detection results of our tool and related work.

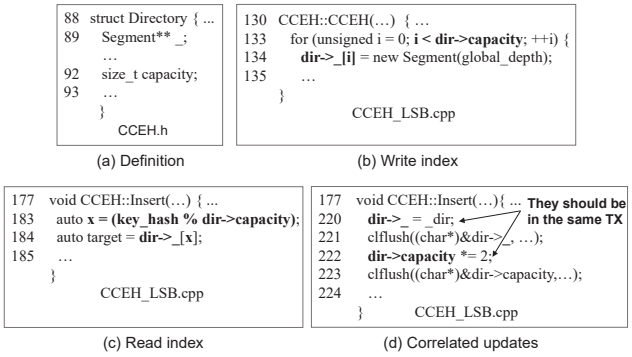


Figure 4: An example of a pair of counting-correlated variables (a), the example write to the array (b), the example read to the array (c), and an atomicity bug detected by our tool (d).

has more frequent access to the array than to its size variable. Witcher is unable to detect most counting-related atomicity violations, since few arrays serve as guardians within condition statements. However, there are three exceptions: bugs #2, #6, and #13, in which the pointer element in the array is used to control the read of the pointed-to structure.

False Positives. Our approach may produce false positives by incorrectly identifying variable pairs as counting-correlated. We have manually inspected the inferred properties and found the existence of such incorrect properties around the misidentified temporary loop variables. The temporary loop variables that control the instructions for accessing the array have the same analyzed behaviors as the logical size variable, leading to the wrong inference. Fortunately, such cases can be easily filtered by manual post-processing (e.g., checking whether the reported integer variable is in PM or DRAM).

A detected bug example. Figure 4 shows the bug #9, where a pointer array `dir->_` and its logical size variable `dir->capacity` is counting-correlated. Our tool could detect this atomicity property, as the read and write access to `dir->_` are restricted by `dir->capacity`. MUVI could detect it due to the symmetric access to the array and the size variable, while Witcher fails since `dir->_` is not a guardian.

Running Time of Property Inference. We recorded the runtime overhead of our tool's workflow on the benchmark programs for inferring PM properties, finding it is between 0.2 and 0.6 seconds. Our tool's static analysis method avoids exploring program states, resulting in a short analysis time. In contrast, Witcher spends 11 minutes to more than 1 hour to detect ordering and atomicity bugs [9]. Yet SMT solvers may become the bottleneck when applying our approach to large-scale PM systems with complex constraints to be checked. This issue could be mitigated by restricting the checked SSAs of an integer variable to the ones in the same block as the array access, reducing the number of constraints. Additionally, optimizations like constraint caching or lightweight approximations could prevent worst-case SMT solver behavior in larger PM systems, which we plan to explore in future work.

5 Conclusion

This paper presents an observation and inference approach for a class of important atomicity properties about the counting relationship between the array and its logical size variable in PM programs. We first propose the necessary conditions for the behaviors of counting-correlated variables among diverse array operations as invariants. We then use symbolic range analysis combined with an SMT solver to discover the variables satisfying the invariants, generating likely atomicity properties. The evaluation shows that using our inference method, 14 bugs are detected in real-world PM programs.

Acknowledgment

We are grateful to the anonymous reviewers and our shepherd Haris Volos for their constructive comments and suggestions. The work was supported in part by City University of Hong Kong internal and donation fundings (No. 9610598 and No. 9220148), and the National Science Foundation (NSF) Grant (No. 2105006).

References

[1] Rastislav Bodik, Rajiv Gupta, and Vivek Sarkar. Abcd: eliminating array bounds checks on demand. In *Proceedings of the ACM SIGPLAN conference on Programming language design and implementation*, pages 321–333, 2000.

[2] CXL Consortium. Compute Express Link™: The Breakthrough CPU-to-Device Interconnect. https://www.computeexpresslink.org/, 2020.

[3] Leonardo De Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[4] Leonardo De Moura and Nikolaj Bjørner. Satisfiability Modulo Theories: Introduction and Applications. *Communications of the ACM*, 54(9):69–77, 2011.

[5] Bang Di, Jiawen Liu, Hao Chen, and Dong Li. Fast, flexible, and comprehensive bug detection for persistent memory programs. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 503–516, 2021.

[6] dibang2008. ISSUE: Inconsistency bugs in array example for libpmemobj. https://github.com/pmem/pmdk/issues/4927, 2020.

[7] Subramanya R Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. System software for persistent memory. In *Proceedings of the Ninth European Conference on Computer Systems*, pages 1–15, 2014.

[8] Michael D Ernst, Jeff H Perkins, Philip J Guo, Stephen McCamant, Carlos Pacheco, Matthew S Tschantz, and Chen Xiao. The daikon system for dynamic detection of likely invariants. *Science of computer programming*, 69(1-3):35–45, 2007.

[9] Xinwei Fu, Wook-Hee Kim, Ajay Paddayuru Shreepathi, Mohannad Ismail, Sunny Wadkar, Dongyoon Lee, and Changwoo Min. Witcher: Systematic crash consistency testing for non-volatile memory key-value stores. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 100–115, 2021.

[10] Xinwei Fu, Dongyoon Lee, and Changwoo Min. Durinn: Adversarial memory and thread interleaving for detecting durable linearizability bugs. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 195–211, 2022.

[11] João Gonçalves, Miguel Matos, and Rodrigo Rodrigues. Mumak: Efficient and black-box bug detection for persistent memory. In *Proceedings of the Eighteenth European Conference on Computer Systems*, pages 734–750, 2023.

[12] Hamed Gorjiara, Guoqing Harry Xu, and Brian Demsky. Jaaru: Efficiently model checking persistent memory programs. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 415–428, 2021.

[13] Eryu Guan. ext4: update i_disksize if direct write past ondisk size. https://marc.info/?l=linux-ext4&m=151669669030547&w=2, 2018.

[14] Shashank Gugnani, Arjun Kashyap, and Xiaoyi Lu. Understanding the idiosyncrasies of real persistent memory. *Proceedings of the VLDB Endowment*, 14(4):626–639, 2020.

[15] Zunchen Huang, Srivatsan Ravi, and Chao Wang. Discovering likely program invariants for persistent memory. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, pages 1795–1807, 2024.

[16] Intel. How to Detect Persistent Memory Programming Errors Using Intel® Inspector - Persistence Inspector. https://www.intel.com/content/www/us/en/developer/articles/technical/detect-persistent-memory-programming-errors-with-intel-inspector-persistence-inspector.html, 2018.

[17] Intel. PMDK Introduction. https://docs.pmem.io/persistent-memory/getting-started-guide/what-is-pmdk, 2019.

[18] Intel. Next generation Intel® Optane™ Persistent Memory. https://community.intel.com/t5/Blogs/Products-and-Solutions/Memory-Storage/Next-generation-Intel-Optane-Persistent-Memory-bringing-more/post/1334979, 2020.

[19] Aasheesh Kolli, Steven Pelley, Ali Saidi, Peter M Chen, and Thomas F Wenisch. High-performance Transactions for Persistent Memories. In *Proc. ACM ASPLOS*, 2016.

[20] Philip Lantz, Subramanya Dulloor, Sanjay Kumar, Rajesh Sankaran, and Jeff Jackson. Yat: A validation framework for persistent memory software. In *2014 USENIX Annual Technical Conference (ATC)*, pages 433–438, 2014.

[21] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *International Symposium on Code Generation and Optimization.*, pages 75–86. IEEE, 2004.

[22] Se Kwon Lee, Jayashree Mohan, Sanidhya Kashyap, Taesoo Kim, and Vijay Chidambaram. Recipe: Converting concurrent dram indexes to persistent-memory indexes. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 462–477, 2019.

[23] Sihang Liu, Korakit Seemakhupt, Yizhou Wei, Thomas Wenisch, Aasheesh Kolli, and Samira Khan. Cross-failure bug detection in persistent memory programs. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1187–1202, 2020.

[24] Sihang Liu, Yizhou Wei, Jishen Zhao, Aasheesh Kolli, and Samira Khan. Pmtest: A fast and flexible testing framework for persistent memory programs. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 411–425, 2019.

[25] Shan Lu, Soyeon Park, Chongfeng Hu, Xiao Ma, Weihang Jiang, Zhenmin Li, Raluca A Popa, and Yuanyuan Zhou. Muvi: Automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 103–116, 2007.

[26] Filipe Manana. Btrfs: fix fsync data loss after append write. https://patchwork.kernel.org/project/linux-btrfs/patch/1434541763-23753-1-git-send-email-fdmanana@kernel.org/, 2015.

[27] Virendra J Marathe, Margo Seltzer, Steve Byan, and Tim Harris. Persistent memcached: Bringing legacy code to {Byte-Addressable} persistent memory. In *9th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 17)*, 2017.

[28] Moohyeon Nam, Hokeun Cha, Young-ri Choi, Sam H Noh, and Beomseok Nam. Write-optimized dynamic hashing for persistent memory. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 31–44, 2019.

[29] Henrique Nazaré, Izabela Maffra, Willer Santos, Leonardo Barbosa, Laure Gonnord, and Fernando Magno Quintão Pereira. Validation of memory accesses through symbolic analyses. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, pages 791–809, 2014.

[30] Ian Neal, Ben Reeves, Ben Stoler, Andrew Quinn, Youngjin Kwon, Simon Peter, and Baris Kasikci. {AGAMOTTO}: How persistent is your persistent memory application? In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 1047–1064, 2020.

[31] Soyeon Park, Shan Lu, and Yuanyuan Zhou. Ctrigger: exposing atomicity violation bugs from their hiding places. In *Proceedings of the 14th international conference on Architectural support for programming languages and operating systems*, pages 25–36, 2009.

[32] Azalea Raad, John Wickerson, Gil Neiger, and Viktor Vafeiadis. Persistency semantics of the intel-x86 architecture. *Proceedings of the ACM on Programming Languages*, 4(POPL):1–31, 2019.

[33] Andy Rudoff. Persistent memory programming. *Login: The Usenix Magazine*, 42(2):34–40, 2017.

[34] Steve Scargall. *Programming Persistent Memory: A Comprehensive Guide for Developers*. 2020.

[35] Anton Shilov. Samsung's Memory-Semantic CXL SSD Brings a 20X Performance Uplift. https://www.tomshardware.com/news/samsung-memory-semantic-cxl-ssd-brings-20x-performance-uplift, 2022.

[36] Yongseok Son, Hara Kang, Heon Young Yeom, and Hyuck Han. A log-structured buffer for database systems using non-volatile memory. In *Proceedings of the Symposium on Applied Computing*, pages 880–886, 2017.

[37] Shyamkumar Thoziyoor, Jung Ho Ahn, Matteo Monchiero, Jay B Brockman, and Norman P Jouppi. A Comprehensive Memory Modeling Tool and Its Application to the Design and Analysis of Future Memory Hierarchies. *ACM SIGARCH Computer Architecture News*, 36(3):51–62, 2008.

[38] Haris Volos, Andres Jaan Tack, and Michael M Swift. Mnemosyne: Lightweight Persistent Memory. *ACM SIGARCH Computer Architecture News*, 2011.

[39] Rui Wang, Shuibing He, Weixu Zong, Yongkun Li, and Yinlong Xu. Xpgraph: Xpline-friendly persistent memory graph stores for large-scale evolving graphs. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1308–1325. IEEE, 2022.

[40] Jian Xu and Steven Swanson. Nova: A log-structured file system for hybrid volatile/non-volatile main memories. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 323–338, 2016.

[41] Pengfei Zuo, Yu Hua, and Jie Wu. Write-optimized and high-performance hashing index scheme for persistent memory. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 461–476, 2018.

Optimizing Input Minimization in Kernel Fuzzing

Hui Guo[†]
East China Normal University

Hao Sun
ETH Zurich

Shan Huang[†]
East China Normal University

Ting Su[*†]
East China Normal University

Geguang Pu[†]
East China Normal University

Shaohua Li[*]
The Chinese University of Hong Kong

Abstract

Ensuring the reliability and security of an operating system (OS) kernel is a critical and challenging task. To this end, *coverage-guided kernel fuzzing* has been employed as an effective technique for finding kernel bugs. Specifically, in kernel fuzzing, *input minimization* is one critical stage to provide short, coverage-preserving seeds for improving the efficacy of fuzzing. However, we observe that the cost of the minimization — taking over half of the fuzzing resources — significantly limits the potential of kernel fuzzing.

To the best of our knowledge, no prior work explores and mitigates the preceding problem in kernel fuzzing. To this end, we introduce and design two general and novel optimization strategies — *influence-guided call removal* and *type-informed argument simplification* — for reducing the minimization cost. The *key idea* of these two strategies is to reduce the number of dynamic program executions needed for verifying whether the new coverage achieved by the inputs is always preserved.

We optimized the input minimization stage by our strategies in Syzkaller, the most popular and representative kernel fuzzer, resulting in a prototype named SyzMini. Our evaluation shows that SyzMini can significantly reduce the minimization cost by 60.7%. Moreover, SyzMini improves branch coverage by 12.5%, and finds 1.7~2X more unique bugs. On the latest upstream kernel version, Syzmini has found 13 previously unknown bugs, all of which have been confirmed and four have already been fixed. Our optimization strategies also show the general applicability for improving the effectiveness of other kernel fuzzers. We have made our implementation of SyzMini publicly available at [1].

1 Introduction

The reliability and security of an operating system (OS) kernel are of paramount importance. The bugs in the OS kernels could potentially lead to serious system-wide consequences, such as crashes [6, 16, 29], privilege escalation [2, 31] and arbitrary code execution [25, 48]. To this end, coverage-guided kernel fuzzing [8, 9, 19, 22, 38, 40, 44] has emerged as one of the most effective approaches for discovering kernel bugs. Indeed, Syzkaller [15], the most popular and representative coverage-guided kernel fuzzer, has been integrated into the testing pipeline of the Linux kernel. It has successfully uncovered over five thousand bugs in Linux [43].

Generally, coverage-guided kernel fuzzing usually includes two major stages in its core fuzzing loop: (1) *mutation* and (2) *minimization*. The mutation stage mutates the seeds to find interesting input programs (in the form of system call sequences) that can cover new code of the tested kernel. Since the new inputs are randomly generated, they are likely cluttered with redundant system calls or arguments. Therefore, an important stage, minimization, is adopted. The minimization stage compresses these interesting programs into a more streamlined and higher-quality set containing only essential calls and arguments while preserving the new coverage. Minimization, in turn, improves the efficacy of subsequent mutations, thus being critical in the fuzzing campaign [35].

Take Syzkaller as an example, it adopts a *one-by-one minimization strategy* [15] to minimize interesting programs. Given an interesting program P in the form of a system call sequence, the minimization attempts to reduce P by (1) removing each call in P one by one, and *later* (2) simplifying each argument in the remaining calls one by one. Crucially, each minimization attempt must be verified that the original coverage achieved by P is always preserved by *dynamically* executing the reduced program; otherwise, the attempt is rolled back. Take Figure 1 for instance. Given a program P that contains five system calls, with the last call c_5 achieving the new coverage, this stage first removes c_1 and executes the reduced sequence. The attempt is rolled back as the coverage of c_5 is altered due to the removal. This procedure is applied to each of the other calls, thus four executions are required (Figure 1(a)). Similarly, the minimization then considers each call argument, where three executions are conducted for a_1,

[*] Ting Su and Shaohua Li are the corresponding authors.
[†] Hui Guo, Shan Huang, Ting Su and Geguang Pu are also affiliated with Shanghai Key Laboratory of Trustworthy Computing.

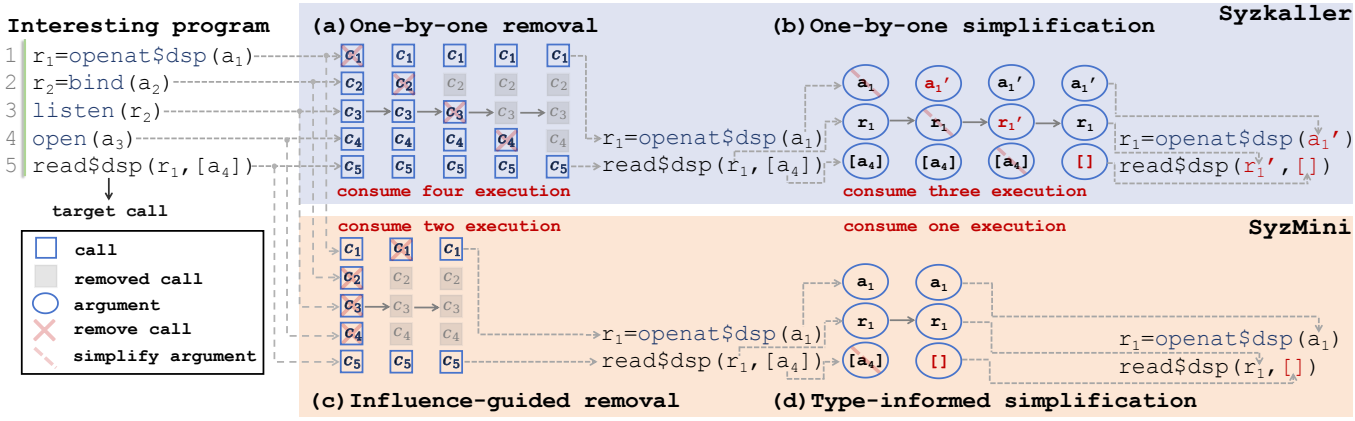


Figure 1: Illustrative example of input minimization between Syzkallar and our approach Syzmini.

r_1 and $[a_4]$ from the remaining calls $[c_1, c_5]$ (Figure 1(b)). In total, seven program executions are needed to minimize P.

However, we observe a significant *tension* exists between the benefits and the cost of the preceeding minimization in kernel fuzzing. On the one hand, the minimization stage improves the efficacy of subsequent mutations by providing short, coverage-preserving inputs. Our experiment on Syzkaller (detailed in Section 3) shows that omitting the minimization, the coverage and the number of found bugs would decrease by 27.5% and 40.4% respectively within a 48-hour fuzzing campaign; moreover, the gap continues to widen with prolonged fuzzing. Indeed, this minimization stage is critical for the effectiveness of kernel fuzzing.

On the other hand, the minimization stage incurs significant costs. For example, the one-by-one minimization in Syzkaller requires *dynamic* program execution to verify the preservation of coverage at each attempt to remove calls or simplify arguments. However, each program execution is expensive because it involves context switching between the user and kernel spaces. Our experiment (detailed in Section 3) reveals that, during a 48-hour fuzzing campaign, 57.5% of program executions are expended in the minimization stage, while the remaining 42.5% of program executions are used by the mutation stage. It is overwhelming that the minimization takes over half of the fuzzing resources.

However, to the best of our knowledge, no prior work explores and mitigates the preceding tension in kernel fuzzing. This paper aims to fill this gap. The key challenge is how to significantly reduce the minimization cost of reducing interesting programs. At the high level, our *key idea* is to reduce the number of program executions needed for verifying whether the new coverage is preserved—the fewer the program executions are, the lower the cost is.

To this end, this paper introduces two *general* and *novel* strategies to optimize the minimization by respectively reducing the number of program executions during removing calls and simplifying arguments. In the scenario of removing calls, let P be an interesting program and c be the target call in P which achieves new coverage. Our *key insight* is that removing the irrelevant calls in P (which have no execution influence on c) will not affect the new coverage achieved by c. Therefore, we could substantially reduce the number of program executions needed for verifying the coverage by removing these irrelevant calls *at one time* rather than *one by one*. Indeed, whether a call can affect c's coverage depends on their inherent dependencies. For example, if a call does not share any global kernel state with c, then this call has no execution influence on c_j and can be safely removed. In practice, this form of execution influence–often referred to as *influence relation* [8, 13, 17, 35, 40]–can be inferred through a combination of static and dynamic analysis on the system calls. In this paper, we term this optimization strategy as *influence-guided call removal strategy*.

Figure 1(c) illustrates this strategy. Based on the inferred influence relation, we could identify that the calls c_2, c_3, c_4 do not impact the execution of c_5, and then attempt to remove these calls at one time. If this attempt is successful (*i.e.*, the new coverage achieved by c_5 is preserved), P will be reduced to $[c_1, c_5]$ by only one program execution. Later, we attempt to remove c_1, which however fails to preserve the new coverage. As a result, our strategy only consumes two executions to minimize P, which reduces the minimization cost to half.

In the scenario of simplifying arguments, one-by-one minimization is conservative and minimizes every argument. As a result, it leads to a large number of program executions due to the numerous arguments and their sub-fields involved. Our *key insight* is that we only need to simplify those redundant arguments or their sub-fields that truly matter for the subsequent mutations. For example, removing superfluous elements from an array argument could significantly reduce the space for subsequent mutations, while investing similar effort in simplifying a simple scalar argument leads to marginal benefits. In this way, we could substantially reduce the number of program executions needed for verifying the coverage without sacrificing the benefits of minimization.

Based on the preceding insight, we design a *type-informed argument simplification strategy* to optimize the minimization. Given an interesting program P, this strategy analyzes the

type information of each argument and its sub-fields in P. If an argument or a field is fixed-size, the strategy skips the simplification; the strategy only attempts to simplify those variable-size arguments which may have redundant mutation space. For example, in Figure 1(d), this strategy does not simplify the fixed-size arguments a_1 (*Integer* type) and r_1 (*Resource* type, a 4-byte file descriptor). It would simplify the variable-size argument $[a_4]$ (*Array* type) to an empty array while ensuring the coverage is preserved. As a result, the type-informed strategy only consumes one program execution, while the original minimization requires three executions.

To evaluate the effectiveness of our approach, we integrated our proposed optimization strategies into Syzkaller, resulting in a prototype named SyzMini [1]. Our evaluation demonstrates that SyzMini significantly reduces the cost of minimization: the number of program executions in minimization is reduced by 60.7%. Compared to vanilla Syzkaller, SyzMini achieves a 12.5% improvement in branch coverage and a 1.7× speedup, while discovering 1.7–2× more unique bugs. Furthermore, SyzMini uncovered 13 previously unknown bugs in the latest upstream Linux, all of which have been confirmed, and four have already been fixed. To further illustrate the generality and benefits of our approach, we enhanced three distinct kernel fuzzing tools by optimizing the minimization: SyzVegas (employing reinforcement learning for scheduling fuzzing tasks), CountDown (accelerating finding memory-related kernel bugs), and SyzDirect (directed fuzzing for reproducing kernel bugs). With our enhancements, SyzVegas increases branch coverage by 14.5% and discovers 50% more bugs, CountDown identifies 66.7% more memory bugs, and SyzDirect reproduces 1.5× more bugs within the same time budget.

This paper has made the following contributions:

- To our knowledge, we are the *first* to explore and mitigate the significant tension between the benefits and cost of input minimization in kernel fuzzing.
- We introduce two general and novel optimization strategies respectively for removing system calls and simplifying arguments in minimization.
- We implement our optimization strategies for minimization in Syzkaller and three distinct kernel fuzzers. The extensive evaluation results demonstrate the effectiveness, general applicability, and benefits of our approach.

2 Background

Fuzzing is an automated software testing technique first introduced by Miller [34]. It aims to uncover bugs by repeatedly sending random inputs to the target program [14, 33]. Kernel fuzzers, domain-specific fuzzing tools for OS kernels, continuously generate system call sequences to find kernel bugs.

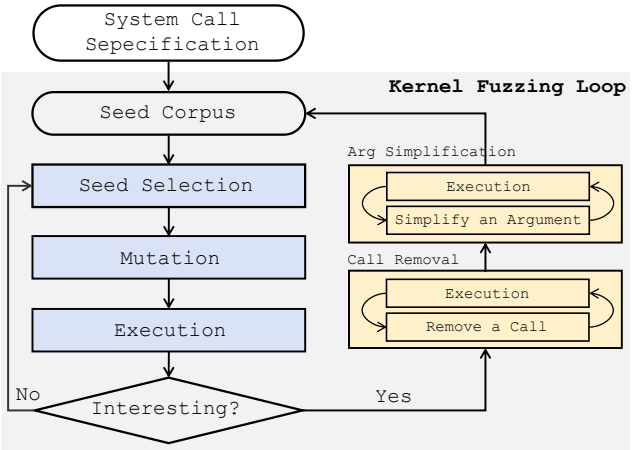


Figure 2: Overall workflow of Syzkaller

Figure 2 shows the workflow of Syzkaller [15], a state-of-the-art coverage-guided kernel fuzzing tool. Syzkaller first loads system call descriptions written in the declarative language *Syzlang* [42] as well as the initial seed corpus, and then conducts a fuzzing loop to explore interesting programs and find new bugs. The fuzzing loop includes two major stages, *i.e.*, *mutation* and *minimization*.

Mutation Stage. The mutation stage randomly selects a seed from the corpus (the programs that previously found new coverage), and then performs a series of random mutations on the seed (*e.g.*, inserting/removing a call, or changing the value of an argument or its sub-fields) to generate more diverse programs. Each program will be executed on the tested kernel and the achieved coverage for each call in the program is collected independently. For instance, given a call sequence $[c_1, c_2, ..., c_n]$, the coverage of each call is collected separately in n branch arrays, *e.g.*, cov_c_1, cov_c_2, ..., cov_c_n. The total coverage achieved so far is maintained in another global set. Hence, the fuzzer can identify the call that achieves new coverage by simply comparing each branch array with the global set, *e.g.*, comparing cov_c_1 with the global set determines if the call c_1 achieves new coverage. When the program achieves new coverage, each call contributing to this new coverage is treated as a target call for the next minimization.

Minimization Stage. The minimization stage minimizes an interesting program as a new seed for subsequent mutation. It first attempts to remove irrelevant calls and later simplifies arguments in the program. It employs a one-by-one minimization strategy. We explain the process as follows.

① *Removing Calls*. Given an interesting program $P = [c_1, c_2, ..., c_n]$, where c_i is the ith call in P. Let the last call c_n be the target call in P which achieves the new coverage. In the minimization stage, Syzkaller first removes the first call c_1 and executes the reduced program $[c_2, c_3, ..., c_n]$ to verify whether the new coverage is preserved. If the coverage is preserved, c_1 will be removed from P. Because c_1 is an irrelevant call *w.r.t.* c_n which does not affect the execution of

Table 1: Argument simplification methods in Syzkaller

Data Types		Simplification Methods
Primitive Type	Integer Flag Protocol Resource	Replace the current value with the default value of this type
Derived Type	Array Buffer	Minimize the number of elements
	Pointer	Simplify the pointed-to object
User-defined Type	Struct Union Enum	Simplify each sub-fields

c_n. In this case, P will be reduced to $[c_2, c_3, ..., c_n]$. Otherwise, P remains unchanged. Later, Syzkaller attempts to remove the next call c_2. This process continues until each call c_i in P (except c_n) has been attempted.

② *Simplifying Arguments*. Assuming after the call removal there are m arguments $[a_1, a_2, ..., a_m]$ in P. Syzkaller would first attempt to simplify the first argument a_1 to a_1' and execute the simplified program to verify whether the new coverage is preserved. If the coverage is preserved, a_1' replaces a_1; otherwise, a_1 remains unchanged. Each argument will be attempted one by one to minimize P.

It is worth noting that simplifying arguments depends on the argument types. Table 1 lists the simplification methods for different argument types in Syzkaller. For the argument of primitive type (*e.g.*, *Integer*, *Flag*, *Protocol* or *Resource*), Syzkaller replaces the argument by the default value of this type. For the argument of derived type (*e.g.*, *Array*, *Buffer*), Syzkaller attempts to minimize the number of elements by binary search. For the argument of user-defined data type (*e.g.*, *Struct*, *Union*), Syzkaller recursively simplifies each sub-fields of this argument one by one.

3 Motivation and Observation

In this section, we demonstrate and characterize the tension between the benefits and cost of minimization in kernel fuzzing. To this end, we selected Linux 5.15, the widely-used and stable long-term Linux version, as the fuzzing target; Syzkaller [15], the most popular kernel fuzzing tool, as the kernel fuzzer. Specifically, we implemented a version of Syzkaller without the minimization stage (named as Syzkaller⁻). For each tool, we allocated 48 hours for one fuzzing campaign to fuzz the kernel, and repeated each fuzzing campaign for ten times to mitigate the randomness.

Observation 1: The minimization stage is crucial in improving the effectiveness of kernel fuzzing in terms of code coverage and number of found bugs.

We compared the branch coverage and the number of found unique bugs of Syzkaller to those of Syzkaller⁻. As shown in Figure 3, compared to Syzkaller, the branch coverage and

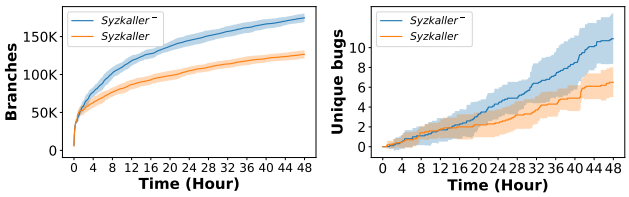


Figure 3: Syzkaller⁻ *vs.* Syzkaller on average coverage and bug finding (shaded area: mean ±1 standard deviation)

the number of found unique bugs of Syzkaller⁻ decrease by 27.5% and 40.4% respectively. Moreover, the gap gradually widens over time. The results show that minimization plays a crucial role in improving the effectiveness of kernel fuzzing.

Observation 2: The proportion of program executions during the minimization stage accounts for 57.5%, leading to significant costs.

We recorded the number of programs executed during the mutation and minimization stages respectively to measure the cost. Figure 4 shows the proportions of the numbers of program executions between mutation and minimization. We can see that the minimization stage consumes more program executions than the mutation stage. The minimization stage reaches a peak of 68.1% at the time of 4.5 hours. The portion gap gradually decreases over time but still accounts for 57.5% at the time of 48 hours. In particular, removing calls accounts for 34.0% and simplifying arguments accounts for 66.0%.

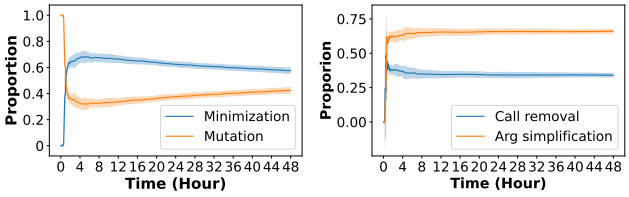


Figure 4: The proportion of program executions during minimization and mutation over time (shaded area: mean ±1 standard deviation).

The reason for this trend is that the fuzzer initially generates a large number of interesting programs with new coverage. Minimizing these programs requires a considerable number of program executions. As a result, the proportion of program executions in minimization rapidly increases in the first four hours. Over time, although the number of interesting programs gradually decreases, these programs become more complex (*e.g.*, involving more calls and arguments) due to the subsequent mutation. These complex programs requires more program executions to conduct the one-by-one minimization. Overall, the proportion of program executions in the minimization stage decreases slowly, but still surpasses that of the mutation stage (after the 48 hours), leading to significant cost.

Summary: The preceding observations indicate that the minimization stage is crucial for the effectiveness of kernel

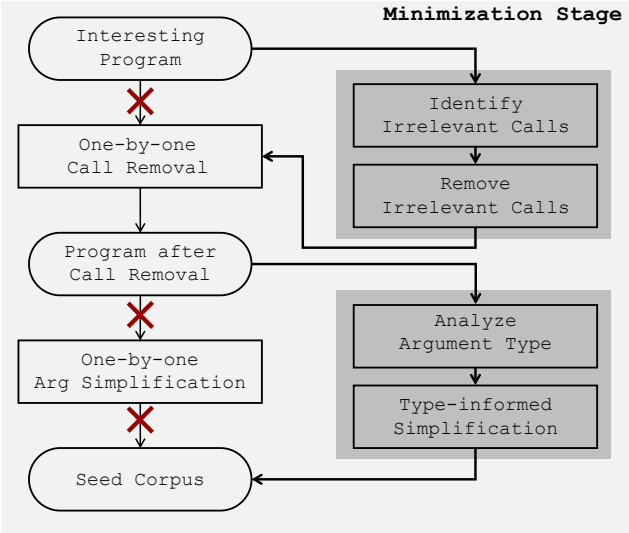


Figure 5: Overview of the influence-guided call removal and the type-informed argument simplification strategies.

fuzzing, but introduces significant cost in terms of program executions. Our goal is to reduce this minimization cost, thereby unleashing the potential of kernel fuzzing.

4 Approach and Implementation

Figure 5 shows the overview of our optimization strategies (*i.e.*, influence-guided call removal and type-informed argument simplification). These two strategies extend the default one-by-one minimization in Syzkaller. Specifically, the influence-guided call removal leverages the influence relations between system calls to identify irrelevant calls and remove these calls at one time. Later, the reduced program undergoes the one-by-one minimization to guarantee the number of calls in P is minimal (detailed in Section 4.1). The type-informed argument simplification analyzes the types of each argument and its sub-fields to identify which argument is fixed-size or variable-size. It would skip the simplification of the fixed-size arguments and only attempts to simplify the variable-size arguments (detailed in Section 4.2).

4.1 Influence-guided Call Removal Strategy

We define the influence relation between two system calls:

***Influence Relation**: For two system calls c_i and c_j, c_i has an influence relation with c_j if executing c_i can change the execution path of c_j by modifying the kernel's internal state that c_j depends on.*

The definition describes the relation between system calls based on the execution path. The relation exists when one call c_i can modify the global kernel state that the behavior of another call c_j depends on, thereby changing the execution path of c_j. In practice, the influence relation can be inferred by the static and dynamic analysis on system calls [40]. We will

Algorithm 1: Influence-guided Removal Strategy

Input : Interesting Program: $P = [c_1, c_2, ..., c_n]$, Influence relation matrix: M
Output : Program after call removal P

```
// Step 1. Identify irrelevant calls
1  relevant_calls ← {}
   // analyze directly relevant calls
2  for i from 1 to n − 1 do
3      if M[c_i][c_n] = 1 then
4          relevant_calls ← relevant_calls ∪ {c_i}
   // analyze indirectly relevant calls
5  work_list ← all relevant calls
6  while work_list is not empty do
7      Pick the last call c_j from work_list
8      for k from j − 1 to 1 do
9          if M[c_k][c_j] = 1 ∧ c_k ∉ relevant_calls then
10             relevant_calls ← relevant_calls ∪ {c_k}
11             Add c_k to work_list
12 irrelevant_calls←{c_1, c_2, ..., c_n}\relevant_calls
   // Step 2. Remove irrelevant calls
13 P' ← P.remove_calls(irrelevant_calls)
14 target_call_cov ← execute(P)
15 target_call_cov' ← execute(P')
16 if target_call_cov ∈ target_call_cov' then
17     P ← P'
   // Step 3. One-by-one call removal
18 one_by_one_call_removal(P)
```

describe how to collect the influence relations in Section 4.3. In this section, we focus on explaining how to leverage these collected influence relations to reduce the minimization cost. Thus, we assume the influence relations have been collected for n calls and stored in M—a two-dimensional matrix $R^{n \times n}$.

Identify Irrelevant Calls. The strategy aims to identify all the calls that are irrelevant to the target call based on the collected influence relations. In algorithm 1, let $P = [c_1, c_2, ..., c_n]$ be an interesting program (assuming the last call c_n in P is the target call which achieves new coverage) and M be the influence relation matrix. The strategy first obtains all the calls that have *direct* influence relations with c_n (lines 2-4), and then identifies the calls that have indirect influence relations with c_n based on the transitivity of influence relation (lines 5-11). Specifically, on line 3, when $M[c_i][c_n]$=1 (denoting that c_i has direct influence on c_n), c_i will be saved into the list *relevant_calls* (which stores all the relevant calls). On line 9, when $M[c_k][c_j]$=1 (denoting that c_k has indirect influence on c_n through c_j), c_k will be also saved into the list *relevant_calls* (line 10). Specifically, the strategy uses a worklist algorithm to iteratively identify all the calls that have indirect influence on c_n. Finally, we obtain the set of irrelevant calls (stored in *irrelevant_calls*) *w.r.t.* P by filtering these collected relevant calls from the set of the calls from P (line 12).

Example. In Figure 6, the program P is an interesting program and c_6 is the target call. According to the influence

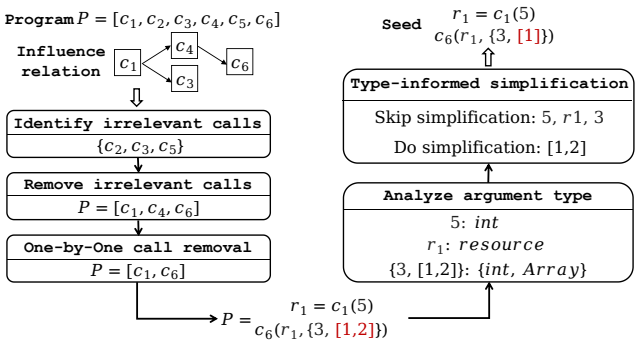


Figure 6: An Example for Influence-guided Removal and Type-informed Simplification

relation, c_4 has direct influence on c_6, and c_1 has indirect influence on c_6 through c_4. Thus, $\{c_1, c_4\}$ are the relevant calls and the other calls $\{c_2, c_3, c_5\}$ in P are identified as irrelevant.

Remove irrelevant calls at one time. When the irrelevant calls are identified, the strategy attempts to remove these calls at one time from P (line 13). If the new coverage of the target call is preserved by P', this attempt is successful and P is reduced to P' (line 17). Otherwise, the attempt will be rolled back — some calls in *irrelevant_calls* actually influence the execution of c_n. No matter whether this attempt is successful or not, the remaining relevant calls in P may not contribute to the new coverage. Thus, P needs to undergo the one-by-one call removal (*i.e.*, the default one-by-one minimization for removing calls) to ensure that each remaining call is truly associated with the new coverage of the target call and the number of calls in P is minimal (line 18). Compared to the default one-by-one minimization in Syzkaller, this influence-guided strategy is likely to remove many calls at one time and thus reduce the minimization cost.

Example. After identifying $\{c_2, c_3, c_5\}$ are the irrelevant calls, the strategy attempts to remove all these calls at one time. Assuming this attempt is successful, P is reduced to $[c_1, c_4, c_6]$ by only requiring one program execution. Later, the one-by-one minimization consumes two program executions to check whether c_1 and c_4 would influence the execution of c_6 and finally P is reduced to $[c_1, c_6]$ (assuming c_4 actually does not influence the execution of c_6). As a result, the optimization strategy only takes three executions to remove calls, while the default minimization requires five executions.

4.2 Type-informed Arg Simplification Strategy

We define fixed-size or variable-size arguments as follows.

*For an argument a, a is a fixed-size argument if a only involves fixed-length data types (*e.g.*, Integer, Flag, Protocol and Resource); a is a variable-size argument if a involves any variable-length data type (*e.g. *Array and Buffer).*

Analyze Argument Type. We detail how we analyze an argument type as follows. Given a program P after call removal, the strategy analyzes the type of each argument a from the

Algorithm 2: Type-informed Arg Simplification Strategy

Input : Program after call removal: $P = [c_1, ..., c_n]$
Output : Final minimized program P

```
1  for i from 1 to n do
2      for arg in c_i.args do
3          Analyze arg's type
4          P ← type_informed_simplification(P, arg)
5  Function type_informed_simplification(P, arg)
6      if arg is a fixed-size argument then
7          continue
8      else
           // arg is a variable-size argument
9          if arg.type is Array or Buffer then
10             P ← simplify_variable-size_arg(P, arg)
11         if arg.type is Pointer then
12             P ← type_informed_simplification(P, arg.point_to_obj)
13         if arg.type is User-defined type then
14             for sub_field in arg.sub_fields do
15                 P ← type_informed_simplification(P, sub_field)
16     return P
17 Function simplify_variable-size_arg(P, arg)
18     P' ← P.remove_element_by_binary_search(arg)
19     target_call_cov ← execute(P)
20     target_call_cov' ← execute(P')
21     if target_call_cov ∈ target_call_cov' then
22         P ← P'
23         P ← simplify_variable-size_arg(P, arg)
24     return P
```

calls in P based on Table 2 (lines 2-3 in Algorithm 2). Specifically, if a belongs to primitive type (such fixed-length data types as *Integer*, *Flag*, *Protocol* and *Resource*), a is classified as a fixed-size argument. If a belongs to derived type (such variable-length data types as *Array* and *Buffer*), a is classified as a variable-size argument. If a belongs to *Pointer*, the strategy will recursively analyze the type of the pointed-to object of a to decide a's argument type according to the preceding definition. If a belongs to *user-defined* type, the strategy will recursively analyze the type of each sub-fields of a to decide a's argument type according to the preceding definition. If any point-to object or sub-field is a variable-length data type, a is a variable-size argument. Otherwise, a is classified as a fixed-size argument. In this way, any argument from the calls in P can be classified into either fixed-size or variable-size.

Example. Taking the program $P = [c_1, c_6]$ after call removal in Figure 6 as an example, assuming c_1 takes an argument 5 and returns r_1; c_2 takes r_1 and $\{3, [1,2]\}$. The strategy can identify 5 as *Integer* type and r_1 as *Resource* type. Since *Integer* and *Resource* are fixed-length data types, both of these two arguments 5 and r_1 are classified as the fixed-size arguments. For the argument $\{3, [1,2]\}$, the strategy will recursively ana-

Table 2: Argument type classification in Linux kernels

Primitive type	Category	Derived type	Category	User-defined type	Category
Integer Flag Protocol Resource	Fixed-size argument	Array Buffer	Variable-size argument	Struct Union Enum	Classify each sub-field
		Pointer	Classify the pointed-to object		

lyze the type of each sub-field. As a result, 3 is identified as *Integer* type and $[1,2]$ is identified as *Array* type. Since the field $[1,2]$ involves a variable-length data type, the argument $\{3,[1,2]\}$ is classifed as a variable-size argument.

Type-informed Simplification. After obtaining the type information of the arguments, in Algorithm 2, this strategy skips the simplification of fixed-size arguments (lines 6-7) and only simplifies the variable-size arguments (lines 9-10). For a variable-size argument *arg*, if the type of *arg* is *Array* or *Buffer*, the strategy will iteratively remove elements from *arg* to minimize the number of elements in the array by binary search while ensuring the coverage is always preserved (lines 18-24). When the type of *arg* is *Pointer* or *User-defined*, the strategy recursively checks the pointed-to object ($point_to_obj$ for *Pointer* type) or each sub-field (sub_field for *user-defined* type) in *arg* and adopts the same method to do simplification (lines 11-15). After traversing all the arguments, the simplified program will be returned and added to the seed corpus.

Example. According to the type analysis result of each argument in Figure 6, since 5 and r_1 are fixed-size arguments, the strategy will skip their simplifications. For the variable-size argument $\{3,[1,2]\}$, the strategy will skip simplifying the sub-field 3 but simplify the sub-field $[1,2]$. It would consume two executions to simplify $[1,2]$ to $[1]$ (assuming the argument $[1]$ can affect the execution of c_6). Finally, the simplified P becomes a seed. Overall, the strategy only takes two executions to simplify the arguments, while the default minimization requires five executions. The reason of skipping the simplification of those fixed-size arguments is that such simplification does not help reduce redundant mutation space.

4.3 Implementation

We implemented the two optimization strategies (*i.e.*, the influence-guided removal and the type-informed simplification) in Syzkaller (commit 1759857fa9bd [15]). We chose Syzkaller because it is the most popular and representative kernel fuzzing tool, and is a standard baseline used by many recent work [3,8,9,20,41,44]. We name the version of Syzkaller enhanced by the two optimization strategies as SyzMini.

Specifically, the influence-guided strategy needs the influence relations between the system calls as input. To this end, we collected the influence relations by following Healer [40]. In practice, we use both static and dynamic analysis to infer and collect the influence relations. In static analysis, given two system calls c_i and c_j, c_i has the influence relation with c_j if (1) c_i returns a resource type r, and at least one of the parameters in c_j is the resource type r, or (2) c_i has a pointer parameter a that points to one resource type r with an outward data flow direction, and c_j has the resource type r with an inward data flow direction. To collect static influence relations, we analyze the type information of each parameter specified by *Syzlang* [42]. In dynamic analysis, we leverage the default minimization process in Syzkaller to infer dynamic influence relations. Specifically, during minimization, if removing a call c_i affects the execution path (*i.e.*, the coverage) of the next call c_{i+1}, then c_i influences c_j. We ran Syzkaller until the branch coverage saturated and collected the dynamic influence relations, which took about four days. We collected a total of 74,865 influence relations including 44,966 static influence relations and 29,899 dynamic influence relations.

SyzMini uses the default branch coverage supported by Syzkaller to record the execution information of each call. To check whether one minimization attempt is successful, SyzMini verifies whether the branch coverage of a target call is preserved. If the coverage is preserved, this attempt is considered as successful.

5 Evaluation

This section evaluates the effectiveness of the two optimization strategies for minimization (*i.e.*, the influence-guided and type-informed strategies) implemented in SyzMini. we aim to answer the following research questions:

RQ1. How effective is SyzMini (which incorporates the two optimization strategies for minimization) in terms of code coverage and bug finding?

RQ2. What are the respective contributions of the influence-guided and type-informed strategies in improving kernel fuzzing?

RQ3. Why are the influence-guided and type-informed strategies effective in improving kernel fuzzing?

RQ4. How effective are the two optimization strategies when they are integrated into other distinct kernel fuzzers?

RQ5. What factors affect the effectiveness of the influence-guided and type-informed strategies?

5.1 Evaluation Setup

All the experiments were conducted on a machine running 64-bit Ubuntu 20.04 LTS and equipped with an AMD 3995WX 64-core CPU and 128G RAM. We selected Linux 5.15, 6.1, and 6.11 as the target kernels to be tested. Specifically, Linux 5.15 and 6.1 are the stable long-term versions and are the most widely used kernel versions by many distributions. Linux 6.11 is the latest upstream version at the time of our work. All these three kernel versions were compiled with the same configuration, *i.e.*, enabling the KCOV and KASAN features to collect code coverage and detect memory-related bugs. During the fuzzing campaign, each kernel runs on a virtual machine instance allocated with a 4-core CPU and 4GB RAM.

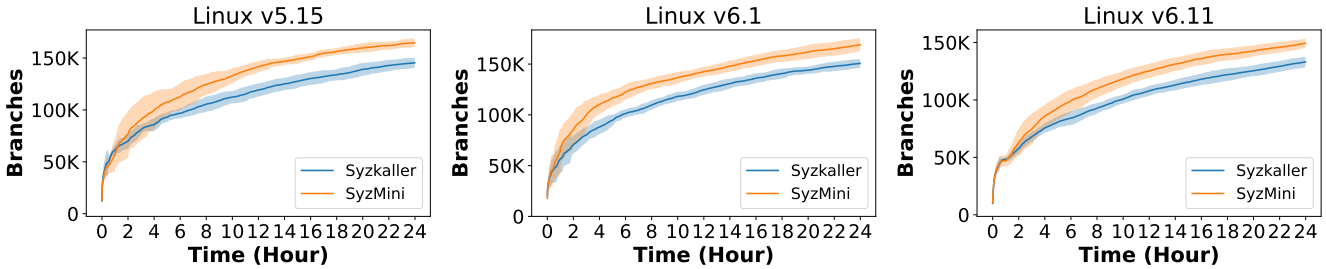


Figure 7: Average branch coverage growth of SYZMINI on v5.15, v6.1 and v6.11 over 24 hours compared to those of SYZKALLER (shaded area: mean ±1 standard deviation)

Table 3: Coverage statistics of Syzkaller *vs.* SyzMini

	v5.15	v6.1	v6.11	Overall
Syzkaller	145.371K	150.677K	133.172K	143.073K
SyzMini	164.724K	169.015K	149.311K	161.017K
Improvement	+13.3%	+12.2%	+12.1%	+12.5%
Speed-up	+1.83×	+1.61×	+1.62×	+1.69×

Table 4: Bug statistics of SyzMini and Syzkaller

Fuzzer	#Unique Bugs v5.15	v6.1	v6.11	Total	Bug Distribution v5.15	v6.1	v6.11	Total
SYZKALLER	14	12	6	27	18 10 4	12 8 4	8 4 2	32 18 9
SYZMINI	28	20	12	50				

[1] The blue and yellow regions in the venn diagrams denote #bugs found by SyzMini and Syzkaller, respectively; the gray regions denote #bugs found by both tools.

The influence-guided strategy is equipped with 74,865 entries of influence relations (detailed in Section 4.3). Each tool uses the same initial seeds as Syzkaller by default across the trials.

5.2 RQ1: Effectiveness of SyzMini

Setup of RQ1. We aim to evaluate the end-to-end effectiveness of SyzMini which incorporates the two optimization strategies. Thus, we compared SyzMini with Syzkaller in terms of code coverage and the number of bugs found. Following the general guidelines for fuzzing evaluation [24, 37], we allocated 24 hours for each kernel fuzzing tool (SyzMini and Syzkaller) for one round of fuzzing campaign and repeated such a fuzzing campaign for 10 rounds to reduce potential randomness. We also measured the *hitting-round* and *time-to-exposure* (*TTE*) for each found bug. These two metrics are widely used in evaluating fuzzing techniques [4, 7, 41]. Specifically, *hitting-round* represents the number of times a fuzzer triggers a target bug in repeated experiments. *TTE* is the time that a fuzzer triggers the target bug at the first time. We computed the arithmetic average of *TTE* (denoted by μTTE) for repeated experiments. If a fuzzer fails to trigger a target bug in a given time duration (24 hours in our setting), the *TTE* of this bug is treated as 24 hours. Additionally, to study the long-term effectiveness, we ran SyzMini and Syzkaller respectively for three days on the latest upstream kernel version Linux 6.11. We collected branch coverage and the number of found bugs.

Code Coverage. Figure 7 plots the branch coverage achieved by Syzkaller and SyzMini during the 24-hour fuzzing. Overall, SyzMini outperforms Syzkaller on the three kernel versions. Specifically, in the first 5 minutes, both tools generate and execute the initial seeds, so they achieve similar coverage. Later, a large number of interesting programs that require minimization are generated. However, performing minimization on these programs does not lead to coverage increases. Thanks to our optimization strategies, SyzMini avoids a large number of unnecessary program executions during minimization. It allows SyzMini to spend more time on mutation. As a result, SyzMini achieves the coverage improvement over Syzkaller after 1.5 hours, and this improvement has been maintained throughout the subsequent fuzzing. The shaded area denotes the average branch coverage ±1 standard deviation. The overlap between the shaded areas of SyzMini and Syzkaller is small, indicating a significant difference.

Table 3 gives the detailed coverage statistics. For example, on v5.15, SyzMini covered an average of 164.724K branches in the 10 rounds of a 24-hour fuzzing campaign, while Syzkaller covered an average of 145.371K branches. As a result, SyzMini achieved 13.3% improvement in branch coverage (see Row "Improvement"). Column "Overall" gives the average number of covered branches on the three kernel versions. Row "speed-up" presents the average speed-up of SyzMini when achieving the same coverage as Syzkaller. On average, SyzMini achieved 12.5% improvement in branch coverage compared to Syzkaller with a speed-up of 1.69×. The results indicate that the optimization strategies can significantly improve the effectiveness of kernel fuzzing in terms of code coverage.

Bug Detection. Table 4 lists the bug statistics. For example, on v5.15, SyzMini found a total of 28 unique bugs in the 10 rounds of 24-hour fuzzing, while Syzkaller found a total of 14 unique bugs. SyzMini found 2× more bugs than Syzkaller. Specifically, on v5.15, SyzMini found 18 bugs that Syzkaller missed, while Syzkaller only found 4 bugs that SyzMini missed. Column "Total" counts the total number of unique bugs found in the three kernel versions. SyzMini found a total of 50 unique bugs, while Syzkaller only 27 unique bugs. Specifically, SyzMini and Syzkaller found a total of 22 common bugs on the three kernel versions (accumulating the common bugs shown in the three venn diagrams of v5.15,

Table 5: *μTTE* and *hitting-round* of each bug

ID	*hitting-round* SK/SM	*μTTE* (h) SK/SM	ID	*hitting-round* SK/SM	*μTTE* (h) SK/SM
1	3 / 6	22.2 / 12.2	12	3 / 5	16.8 / 14.0
2	1 / 2	23.9 / 19.7	13	7 / 10	12.5 / 3.5
3	4 / 8	15.8 / 10.8	14	1 / 2	23.5 / 20.9
4	1 / 3	23.1 / 20.9	15	2 / 4	19.4 / 15.9
5	5 / 5	17.0 / 13.8	16	1 / 4	22.3 / 18.9
6	7 / 9	19.4 / 15.8	17	2 / 8	22.4 / 13.1
7	3 / 4	22.8 / 19.4	18	1 / 6	23.4 / 13.3
8	3 / 3	21.7 / 20.7	19	1 / 3	22.3 / 19.5
9	1 / 2	22.3 / 20.1	20	10 / 10	4.2 / 2.0
10	1 / 2	22.0 / 20.8	21	1 / 3	23.9 / 19.4
11	9 / 9	11.9 / 7.5	22	8 / 10	5.2 / 2.8

[1] SK and SM are Syzkaller and SyzMini, respectively. 1~10, 11~18 and 19~22 are the common bugs on v5.15, v6.1 and v6.11, respectively.
[2] The green one indicates which tool achieves the higher *hitting-round* or the lower *μTTE*.

v6.1, and v6.11 in Column "Bug Distribution").

Table 5 gives the *hitting-round* and *μTTE* for each common bug. On any of these bugs, SyzMini found them faster and more frequently than Syzkaller. Taking the bug with ID 1 as an example, the *μTTE* of Syzkaller is 22.2h, which is 1.8× more than that of SyzMini (12.2h); moreover, SyzMini successfully found this bug in 6 out of 10 rounds of fuzzing, while Syzkaller only found it in 3 rounds. These strong results show that SyzMini significantly outperforms Syzkaller in bug finding.

Long-term Effectiveness. In the 10 rounds of a three-day fuzzing campaign, SyzMini covered an average of 199.4K branches, which is 5.6% more than that of Syzkaller (188.9K). In terms of bug finding, although Syzkaller has been continuously deployed to find bugs in the upstream kernel version, SyzMini has still found 13 previously unknown bugs. We reported each bug (with the corresponding reproducer and kernel configuration) to the kernel maintainers. All these 13 bugs have been confirmed and 4 have already been fixed. Table 6 lists these bugs. Therefore, SyzMini is also effective in long-term fuzzing for improving coverage and finding bugs.

Table 6: Previously unknown bugs found by SyzMini.

ID	Subsystem	Function	Risk	Status
1	Block	bio_put_slab	use after free	confirmed
2	Gfs2	gfs2_quota_init	kernel bug	confirmed
3	NTFS3	end_buffer_read_sync	out of bounds	confirmed
4	Reiserfs	update_stat_data	kernel bug	confirmed
5	Net	netlink_unicast	general protection fault	confirmed
6	MM/pat	get_pat_info	inconsistent lock state	fixed
7	Net	cleanup_net	use after free	fixed
8	Bpf	btf_name_valid_section	out of bounds	confirmed
9	Jfs	jfs_syncpt	null-ptr-deref	fixed
10	Usb	usb_disable_device	refcount bug	confirmed
11	Bpf	build_id_parse_nofault	paging fault	fixed
12	Bluetooth	sco_sock_close	use after free	confirmed
13	Jfs	jfs_statfs.cold	out of bounds	confirmed

In summary, our optimization strategies not only improve branch coverage by 12.1~13.3% and found 1.7~2.0× more unique bugs, but find kernel bugs faster and more stably.

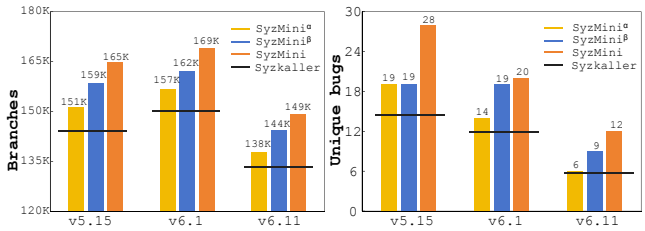


Figure 8: Numbers of covered branches and found unique bugs by Syzkaller, SyzMini$^{\alpha}$, SyzMini$^{\beta}$ and SyzMini.

5.3 RQ2: Contributions of Optimizations

Setup of RQ2. To investigate the respective contributions of the two optimization strategies in improving kernel fuzzing, we implemented two versions of SyzMini: SyzMini$^{\alpha}$ and SyzMini$^{\beta}$. SyzMini$^{\alpha}$ only includes the influence-guided strategy while SyzMini$^{\beta}$ only includes the type-informed strategy. We compare the code coverage and the number of found unique bugs between Syzkaller, SyzMini$^{\alpha}$, SyzMini$^{\beta}$ and SyzMini to analyze the contributions of the two optimization strategies. Following the setup of RQ1, we ran each tool for 24 hours and repeated 10 rounds.

Results. Figure 8 shows the numbers of covered branches and the numbers of found unique bugs by Syzkaller, SyzMini$^{\alpha}$, SyzMini$^{\beta}$ and SyzMini. For example, on v5.15, equipped with the influence-guided strategy, SyzMini$^{\alpha}$ achieved an average of 4.0% branch coverage improvement (from 145K to 151K) and found 5 more unique bugs (from 14 to 19), compared to Syzkaller. By comparing SyzMini$^{\beta}$ and SyzMini, without this influence-guided strategy, SyzMini$^{\beta}$'s branch coverage dropped by 3.7% (from 165K to 159K) and missed 9 unique bugs (from 28 to 19). Thus, the influence-guided strategy is useful in improving kernel fuzzing.

For the type-informed strategy, on v5.15, SyzMini$^{\beta}$ achieved an average of 9.1% more branch coverage (from 145K to 159K) and found 5 more unique bugs, compared to Syzkaller. By comparing SyzMini$^{\alpha}$ and SyzMini, without the type-informed strategy, the branch coverage achieved by SyzMini$^{\alpha}$ dropped by 8.2% (from 165K to 151K) and missed 9 unique bugs. The results indicate that the type-informed strategy is also important in improving kernel fuzzing. We also note that SyzMini$^{\beta}$ achieved more improvement than SyzMini$^{\alpha}$. We will explain this phenomenon in Section 5.4.

It is worth noting that the optimization effects of these two strategies are orthogonal. Because these two strategies act on two different steps (*i.e.*, removing calls and simplifying arguments) in minimization. Thus, SyzMini achieved the highest coverage and found the most number of bugs compared to Syzkaller, SyzMini$^{\alpha}$ and SyzMini$^{\beta}$.

In summary, the influence-guided strategy improves the coverage by 3.5~4.6% and increases the number of found unique bugs by 1~1.35×, while the type-informed strategy enhances coverage by 8.3~9.1% and increases the number of found unique bugs by 1.2~1.5×. The optimization effects of

Table 7: Numbers of program executions under different minimization strategies in removing calls and simplifying args.

Minimization	Number of Program Executions		
	Removing Calls	Simplifying Args	All
one-by-one	140,620	260,239	400,859
influence-guided	60,372 (57.1%↓)	–	320,611 (20.0%↓)
type-informed	–	96,896 (62.8%↓)	237,516 (40.7%↓)
both optimizations	60,372 (57.1%↓)	96,896 (62.8%↓)	157,268 (60.7%↓)

the two optimization strategies are orthogonal.

5.4 RQ3: Reasons of Effectiveness

Setup of RQ3. We designed a controlled experiment to explain why the two optimization strategies can improve the effectiveness of kernel fuzzing by reducing the cost of minimization. To this end, We ran Syzkaller for 24 hours and collected 16,266 interesting programs that need minimization. We applied the default one-by-one minimization strategy in Syzkaller, the influence-guided strategy and the type-informed strategy in SyzMini respectively to minimize these interesting programs. We counted the number of program executions needed for verifying the coverage to approximate the cost required by these different minimization strategies. Additionally, we also measured the portions of program executions between mutation and minimization before and after applying the two optimization strategies to explain the effectiveness.

Results. Table 7 gives the numbers of program executions under different minimization strategies. Compared to the one-by-one minimization, the influence-guided strategy reduced 57.1% program executions when removing calls (see Column "Removing Calls"), while the type-informed strategy reduced 62.8% program executions when simplifying arguments (see Column "Simplifying Args"). Overall, the influence-guided and type-informed strategies respectively avoided 20.0% and 40.7% of the total program executions required by the one-by-one strategy. Moreover, when both strategies are enabled, they can reduce a total of 60.7% program executions. These results show that the two optimization strategies can avoid a large number of program executions during minimization. As a result, SyzMini can allocate more time to the mutation stage and execute more diverse programs. It explains why SyzMini can achieve higher coverage and find more bugs than Syzkaller.

It is also worth noting that the type-informed strategy reduces 40.7% program executions, which is about 2X more than that of the influence-guided strategy (20.0%). The former strategy can reduce more cost of minimization than the latter. This result explains why SyzMini$^{\beta}$ can achieve more improvement than SyzMini$^{\alpha}$ for kernel fuzzing in RQ2.

Figure 9 shows the portion of program executions between the minimization and mutation stages in a 24-hour fuzzing campaign *before* and *after* applying the optimizations on the three kernel versions (*i.e.*, comparing between Syzkaller and SyzMini). Compared to Syzkaller, SyzMini

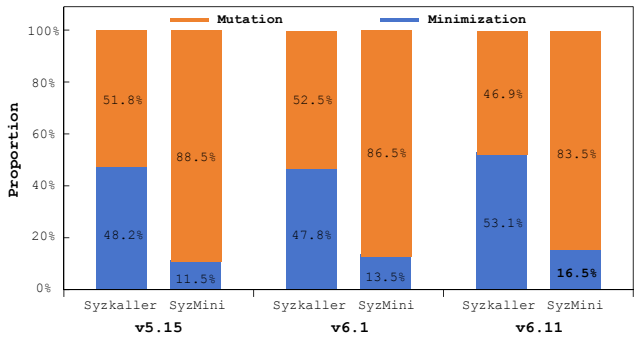


Figure 9: Proportion of program executions between mutation and minimization in SYZKALLER and SYZMINI

significantly reduce the proportion of program executions in minimization, while substantially increases the proportion of program executions in mutation. Taking v5.15 as an example, Syzkaller's minimization occupies 48.2% program executions, while SyzMini's minimization only occupies 11.5% (reduced by 75.7%); on the other hand, Syzkaller's mutation occupies 51.8% program executions, while SyzMini's mutation occupies 88.5% (increased by 70.5%). The results show that SyzMini does reduce the minimization cost and more time is allocated to mutation.

In summary, the influence-guided and type-informed optimization strategies reduce 60.7% of program executed during the minimization and more time is allocated to the mutation, thereby enhancing the effectiveness of kernel fuzzing.

5.5 RQ4: Applicability of Optimizations

Setup of RQ4. We aim to evaluate the generability and applicability of our optimization strategies on other distinct kernel fuzzers. To this end, we conducted a survey of research work on kernel fuzzing which was published in the recent five years (2019-2024) in relevant top venues (S&P, USENIX, CCS, NDSS, OSDI, SOSP, Eurosys, ATC). We collected 11 papers. We excluded 3 papers that did not open-source their tools, and 5 papers whose tools had not been actively maintained anymore. We finally obtained SyzVegas [44], CountDown [3] and SyzDirect [41] as the target fuzzers, all of which are built on Syzkaller but are designed for different purposes.

Specifically, SyzVegas targets adopting reinforcement learning for better scheduling seed selection, mutation and generation. CountDown targets detecting memory-related kernel bugs by a reference counter-guided mutation operator. SyzDirect is a direct kernel fuzzing tool which targets reproducing kernel bugs. These three tools are the state-of-art with different application scenarios. We implemented our optimization strategies in SyzVegas, CountDown and SyzDirect, and obtained their enhanced versions SyzVegas^{+}, CountDown^{+} and SyzDirect^{+}. We investigate whether their performance could be further improved.

To ensure fairness, we followed the default setups and evaluation metrics of these tools described in their papers. To

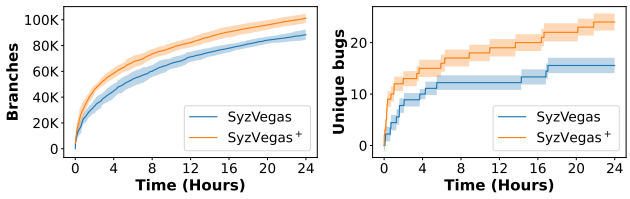


Figure 10: SyzVegas *vs.* SyzVegas$^+$ in term of branch coverage and number of unique bugs (shaded area: mean ±1 standard deviation).

compare SyzVegas and SyzVegas$^+$, we measured code coverage and number of found bugs. To compare CountDown and CountDown$^+$, we counted the number of found KSAN bugs (memory-related errors reported by Kernel Address Sanitizer). To compare SyzDirect and SyzDirect$^+$, we used *µTTE* and *hitting-round* to evaluate the directed fuzzing abilities. Specifically, SyzDirect provides a list of 100 kernel bugs as its evaluation dataset. Considering the experimental cost, we selected the first 10 bugs as the evaluation dataset in this paper. To prepare the dataset, we compiled the corresponding ten versions of Linux kernels with the configuration recommended by SyzDirect. We ran each tool 24 hours and repeated each experiment 10 rounds to migrate randomness. Thus, this evaluation for the three kernel fuzzing tools and their respective enhanced versions took 5,760 machine hours.

SyzVegas vs SyzVegas$^+$. Figure 10 shows the achieved code coverage and the number of found unique bugs on Linux v5.15. With the help of the two optimization strategies, SyzVegas$^+$ improves branch coverage by 14.5%. SyzVegas$^+$ found 24 unique bugs, while SyzVegas only found 16 unique bugs. It is clear that our optimization strategies enhances SyzVegas in code coverage and bug finding.

CountDown vs CountDown$^+$. Table 8 lists the number of KASAN bugs found by CountDown and CountDown$^+$ on Linux v5.11. During the 10 rounds of a 24-hour fuzzing campaign, CountDown$^+$ found a total of 43 KASAN bugs, while CountDown only found 27 bugs. CountDown$^+$ also achieved 4.5% more branch coverage. CountDown leverages the memory relation between system calls to guide the mutation of seeds for finding memory-related bugs. However, all the seeds in the corpus require minimization to improve mutation effectiveness. Our optimization strategies reduce the minimization cost, thereby allowing CountDown to have more time in performing memory-related mutations for finding KASAN bugs.

Table 8: CountDown *vs.* CountDown$^+$ on the KASAN bugs

Tool	All KASAN Bugs			Branch Coverage		
	Min	Max	Total	Min	Max	Avg
CountDown	0	5	27	116.9k	120.7k	119.0k
CountDown$^+$	1	7	43	119.7k	130.1k	124.3k
Improvement	1↑	2↑	66.7%↑	2.4%↑	7.8%↑	4.5%↑

SyzDirect vs SyzDirect$^+$. Table 9 gives the *µTTE* and *hitting-round* for reproducing each target bug. SyzDirect$^+$ successfully reproduced 9 bugs, while SyzDirect reproduced 6 bugs.

Table 9: SyzDirect vs SyzDirect$^+$ on bug reproduction

Bug ID	Commit ID	SyzDirect *µTTE*∼*hitting-round*	SyzDirect$^+$ *µTTE*∼*hitting-round*
1	3f2db250099f	×	16.0h∼4/10
2	c9a2f90f4d6b	21.6h∼2/10	20.0h∼4/10
3	b1a811633f73	×	21.2h∼1/10
4	2fd10bcf0310	21.2h∼2/10	10.1h∼8/10
5	b648eba4c69e	×	×
6	20aaef52eb08	13.6h∼10/10	12.8h∼10/10
7	3b0c40612471	18.8h∼4/10	17.6h∼6/10
8	ffb324e6f874	×	22.1h∼1/10
9	01faae5193d6	18.7h∼4/10	14.4h∼6/10
10	b2a616676839	20.4h∼6/10	14.10h∼8/10

The 6 bugs reproduced by SyzDirect were all reproduced by SyzDirect$^+$. Moreover, for these 6 bugs, SyzDirect$^+$ achieved higher *hitting-round*s and lower *µTTE* compared to SyzDirect. The results shows that our optimization strategies can enhance the directed fuzzing ability of SyzDirect. Given a bug location, SyzDirect prioritizes the mutations of seeds that are close to the bug location. But it still requires minimization to improve the quality of seeds. Our optimization strategies save the minimization cost, and give SyzDirect more time to mutation for increasing the probability of reproducing bugs.

In summary, the influence-guided and type-informed optimization strategies are orthogonal to the optimization strategies of existing kernel fuzzing tools. They improve SyzVegas by covering 14.5% more coverage and finding 1.5× more unique bugs, help CountDown identify 66.7% more KASAN bugs and enhance SyzDirect by reproducing 1.5× more kernel bugs.

5.6 RQ5: Factors affecting Optimizations

Setup of RQ5. We aim to analyze the factors that could affect the effectiveness of our optimization strategies. Specifically, the influence-guided optimization strategy leverages the influence relations between system calls to reduce the minimization cost. Thus, the number of influence relations is a critical factor affecting the strategy's effectiveness. To this end, we conducted a controlled experiment to study the impact of different numbers of influence relations on the effectiveness of this strategy. In our experiment, we have 74,865 influence relations in total. We selected the first 10%, 20%, …, 100% respectively from the list of these 74,865 influence relations as the input of the influence-guided strategy and observe the number of program executions during minimization.

The type-informed strategy reduces the minimization cost by skipping simplifying the fixed-size arguments. Thus, the number of fixed-size parameters (including the pointed-to objects if the parameters are *Pointer* type and the sub-fields if the parameters are *user-defined* type) in the system calls affects this strategy's effectiveness. However, obtaining all the system calls in the Linux kernel is difficult. We note that Syzkaller regularly updates the tested system calls when the Linux kernel has new updates. To this end, we analyzed the tested system calls provided by Syzkaller in the recent five

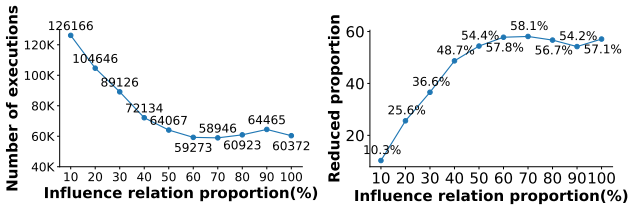


Figure 11: Number of program executions used for removing calls in minimization, and the reduced proportion of program executions compared to the one-by-one minimization.

years (2019-2024). We developed a script to scan all the tested system calls based on the *Syzlang* syntax [42], and counted the numbers of fixed-size and variable-size parameters according to the analysis method in Section 4.2.

Influence-guided Strategy. Figure 11 shows (1) the numbers of program executions used for removing calls in minimization, and (2) the reduced proportions of program executions compared to the one-by-one minimization (calculated by the similar method in Table 7) under different proportions of influence relations. We can see that when the proportion of influence relations increases, the number of program executions used for removing calls gradually decreases (see the left subfigure), and the reduced portion of program executions gradually increases (see the right subfigure). When the proportion of influence relations reaches 60%, the number of program executions and the reduced portion of program executions gradually stabilize. The reason is that, when the number of influence relations increases, the number of irrelevant calls identified by the influence-guided strategy in the program to be minimized gradually saturates. It also indicates that the number of collected influence relations in our experiment is enough for the strategy to take effect.

It is worth noting that this evaluation has also implied the relationship between such precision and optimization effectiveness. The reason is that proportion adopted essentially approximates the precision of influence relationship. When a smaller proportion of influence relation is given, the type-informed strategy will erroneously identify more irrelevant calls in the interesting program, reducing the precision of inferring irrelevant calls. The final result in Figure 11 shows that when the proportion of influence relation increases, the optimization effect increases; however, when the proportion reaches beyond 60%, its effect on the optimization strategy is negligible.

Type-informed Strategy. Table 10 shows the numbers of fixed-size and variable-size parameters in the system calls provided by Syzkaller. Indeed, we can see that the number of fixed-size parameters account for a large portion. Taking the version of Syzkaller in Dec. 2020 as an example, the proportion of fixed-size parameters is 80.5%, while the proportion of variable-size parameters is only 19.5%. The results shows that the tested system calls in Linux kernels are composed of a large number of fixed-size parameters. The type-informed strategy can avoid simplifying many unnecessary arguments which cannot benefit reducing the redundant mutation space. As a result, the number of program executions needed for verifying the coverage could be significantly reduced.

Table 10: Numbers of fixed-size and variable-size parameters in the system calls provided by Syzkaller.

Date	Commit ID	Fixed-size Parameter	Variable-size Parameter
2020-12	79264ae39c1e	101,897 (80.5%)	24,679 (19.5%)
2021-12	36bd2e486525	103,442 (80.1%)	25,643 (19.9%)
2022-12	ab32d50881df	107,575 (80.2%)	26,557 (19.8%)
2023-12	fb427a078200	112,523 (80.6%)	27,145 (19.4%)
2024-12	d7f584ee3c24	285,517 (74.0%)	100,631(26.0%)

In summary, for the influence-guided strategy, within the threshold of 60% influence relations, increasing the number of relations can improve its effectiveness and its optimization effect gradually stabilizes as the number of relations continues to grow. For the type-informed strategy, the fixed-size parameters occupy a high proportion in tested system calls, which help reduce the cost of simplifying arguments.

6 Discussion

SyzMini optimizes the minimization stage in kernel fuzzing and does not affect other parts of a coverage-guided kernel fuzzer. Figure 5 shows the extension implemented by SyzMini on top of Syzkaller. In the literature, a number of research work has been conducted to improve different aspects of kernel fuzzing (*e.g.*, generating system call description [8, 9, 39], mutating the seeds [13, 40] and scheduling the fuzzing tasks [20]). SyzMini is orthogonal to these existing work and would not affect any of these improvements. Thus, our optimization strategies are not limited to Syzkaller. Our extensive evaluation in Section 5.5 has also shown the significant effectiveness improvement when integrating our strategies into three kernel fuzzers (SyzVege, Countdown and SyzDirect) for different application scenarios. We believe integrating our optimization strategies to other kernel fuzzers will boost their effectiveness as well.

7 Related Work

Kernel Fuzzing. Since the success of Syzkaller [15], there has been much research work to improve the effectiveness of kernel fuzzing. A number of research work [5, 8, 9, 11, 17, 18, 39, 45] has been devoted to obtaining more comprehensive system call specifications for producing high-quality seeds. Moonshine [35] aims to generate high-quality initial seeds by distilling the collected system call sequences, thereby enhancing the mutation effectiveness. Other work [10, 13, 17, 23, 40] utilizes the relationship between system calls to narrow the mutation space. SyzVegas [44] leverages reinforcement learning to better schedule the fuzzing tasks such as program generation, mutation and minimization. Different from these existing work, our work SyzMini aims to reduce the cost of

minimization in kernel fuzzing, which has not been explored before.

Kernel Fuzzing Acceleration. There are some research work to accelerate kernel fuzzing. Horus [32] improves the communication speed between the host machine and the virtual machine running the tested kernel by replacing the remote procedure calls (RPCs) with direct memory access. Agamotto [12] observes that the kernel driver fuzzers frequently execute similar test cases, so it dynamically creates multiple checkpoints for these test cases to skip redundant parts. Different from these work, our work SyzMini accelerates kernel fuzzing by reducing the minimizaiton cost.

Input Minimization. Zeller *et al.* [47] first propose the concept of delta debugging to minimize the failure-inducing input. Delta debugging has been adapted in many scenarios. For example, AFL-family tools [28, 30, 46] use afl-cmin [26] and afl-tmin [27] to respectively minimize the seed corpus and an input file for improving the effectiveness of fuzzing. Specifically, afl-tmin [27] takes an input file and tries to remove as much data as possible while maintaining the same coverage as observed initially. In compiler fuzzing, C-Reduce [36] is used to reduce the bug-triggering C program by removing redundant code. In GUI testing, SimplyDroid [21] simplifies crash-triggering UI event traces for Android apps. However, these existing minimization techniques based on delta debugging cannot be directly applied to minimize the system call sequences from OS kernels. Because they do not consider the dependencies between the system calls and the type structures of arguments.

8 Conclusion

We have proposed two general and novel optimization strategies to reduce the cost of minimization stage in kernel fuzzing. The influence-guided strategy leverages the influence relation between system calls to optimize the call removal and the type-informed strategy utilizes the argument type information to avoid unnecessary simplification. Our evaluation shows that the minimization cost can be greatly reduced, compared to Syzkaller. The optimization strategies also help improve branch coverage and find more unique bugs. We believe our optimization strategies are general and could benefit many other kernel fuzzers.

9 ACKNOWLEDGMENTS

We thank the anonymous USENIX ATC reviewers and the shepherd for their valuable feedback. This work was supported in part by National Key Research and Development Program (Grant 2022YFB3104002), Shanghai Trusted Industry Internet Software Collaborative Innovation Center, "Digital Silk Road" Shanghai International Joint Lab of Trustworthy Intelligent Software under Grant 22510750100.

References

[1] Syzmini. https://github.com/ecnusse/SyzMini, 2024.

[2] Erin Avllazagaj, Yonghwi Kwon, and Tudor Dumitras. SCAVY: automated discovery of memory corruption targets in linux kernel for privilege escalation. In *33rd USENIX Security Symposium, USENIX Security 2024, Philadelphia, PA, USA, August 14-16, 2024*, 2024.

[3] Shuangpeng Bai, Zhechang Zhang, and Hong Hu. Countdown: Refcount-guided fuzzing for exposing temporal memory errors in linux kernel. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security, CCS 2024, Salt Lake City, UT, USA, October 14-18, 2024*, pages 1315–1329. ACM, 2024.

[4] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 2329–2344, 2017.

[5] Alexander Bulekov, Bandan Das, Stefan Hajnoczi, and Manuel Egele. No grammar, no problem: Towards fuzzing the linux kernel without system-call descriptions. In *30th Annual Network and Distributed System Security Symposium, NDSS 2023, San Diego, California, USA, February 27 - March 3, 2023*, 2023.

[6] Haogang Chen, Yandong Mao, Xi Wang, Dong Zhou, Nickolai Zeldovich, and M. Frans Kaashoek. Linux kernel vulnerabilities: state-of-the-art defenses and open problems. In *APSys '11 Asia Pacific Workshop on Systems, Shanghai, China, July 11-12, 2011*, page 5, 2011.

[7] Hongxu Chen, Yinxing Xue, Yuekang Li, Bihuan Chen, Xiaofei Xie, Xiuheng Wu, and Yang Liu. Hawkeye: Towards a desired directed grey-box fuzzer. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, pages 2095–2108, 2018.

[8] Weiteng Chen, Yu Hao, Zheng Zhang, Xiaochen Zou, Dhilung Kirat, Shachee Mishra, Douglas Lee Schales, Jiyong Jang, and Zhiyun Qian. Syzgen++: Dependency inference for augmenting kernel driver fuzzing. In *IEEE Symposium on Security and Privacy, SP 2024, San Francisco, CA, USA, May 19-23, 2024*, pages 4661–4677, 2024.

[9] Weiteng Chen, Yu Wang, Zheng Zhang, and Zhiyun Qian. Syzgen: Automated generation of syscall specification of closed-source macos drivers. In *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*, pages 749–763, 2021.

[10] Jaeseung Choi, Kangsu Kim, Daejin Lee, and Sang Kil Cha. Ntfuzz: Enabling type-aware kernel fuzzing on windows with static binary analysis. In *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*, pages 677–693, 2021.

[11] Jake Corina, Aravind Machiry, Christopher Salls, Yan Shoshitaishvili, Shuang Hao, Christopher Kruegel, and Giovanni Vigna. DIFUZE: interface aware fuzzing for kernel drivers. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 2123–2138, 2017.

[12] Ren Ding, Yonghae Kim, Fan Sang, Wen Xu, Gururaj Saileshwar, and Taesoo Kim. Hardware support to improve fuzzing performance and precision. In *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*, pages 2214–2228, 2021.

[13] Marius Fleischer, Dipanjan Das, Priyanka Bose, Weiheng Bai, Kangjie Lu, Mathias Payer, Christopher Kruegel, and Giovanni Vigna. ACTOR: action-guided kernel fuzzing. In *32nd USENIX Security Symposium, USENIX Security 2023, Anaheim, CA, USA, August 9-11, 2023*, pages 5003–5020, 2023.

[14] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. Automated whitebox fuzz testing. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2008, San Diego, California, USA, 10th February - 13th February 2008*, 2008.

[15] Google. Syzkaller. https://github.com/google/syzkaller, 2015.

[16] Weining Gu, Zbigniew Kalbarczyk, Ravishankar K. Iyer, and Zhen-Yu Yang. Characterization of linux kernel behavior under errors. In *2003 International Conference on Dependable Systems and Networks (DSN 2003), 22-25 June 2003, San Francisco, CA, USA, Proceedings*, pages 459–468, 2003.

[17] HyungSeok Han and Sang Kil Cha. IMF: inferred model-based fuzzer. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 2345–2358, 2017.

[18] Yu Hao, Guoren Li, Xiaochen Zou, Weiteng Chen, Shitong Zhu, Zhiyun Qian, and Ardalan Amiri Sani. Syzdescribe: Principled, automated, static generation of syscall descriptions for kernel drivers. In *44th IEEE Symposium on Security and Privacy, SP 2023, San Francisco, CA, USA, May 21-25, 2023*, pages 3262–3278, 2023.

[19] Dae R. Jeong, Kyungtae Kim, Basavesh Shivakumar, Byoungyoung Lee, and Insik Shin. Razzer: Finding kernel race bugs through fuzzing. In *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*, pages 754–768, 2019.

[20] Dae R. Jeong, Byoungyoung Lee, Insik Shin, and Youngjin Kwon. Segfuzz: Segmentizing thread interleaving to discover kernel concurrency bugs through fuzzing. In *44th IEEE Symposium on Security and Privacy, SP 2023, San Francisco, CA, USA, May 21-25, 2023*, pages 2104–2121, 2023.

[21] Bo Jiang, Yuxuan Wu, Teng Li, and W. K. Chan. Simplydroid: efficient event sequence simplification for android application. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*, pages 297–307, 2017.

[22] Dave Jones. Trinity: Linux system call fuzzer. https://github.com/kernelslacker/trinity, 2012.

[23] Kyungtae Kim, Dae R. Jeong, Chung Hwan Kim, Yeongjin Jang, Insik Shin, and Byoungyoung Lee. HFL: hybrid fuzzing on the linux kernel. In *27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26, 2020*, 2020.

[24] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, pages 2123–2138, 2018.

[25] Piotr Krysiuk. Linux kernel use-after-free in netfilter nf_tables when processing batch requests can be abused to perform arbitrary reads and writes in kernel memory. https://seclists.org/oss-sec/2023/q2/133, 2023.

[26] Michal "lcamtuf" Zalewski. afl-cmin. https://github.com/google/AFL/blob/master/afl-cmin, 2016.

[27] Michal "lcamtuf" Zalewski. afl-tmin. https://github.com/google/AFL/blob/master/afl-tmin.c, 2017.

[28] Michal "lcamtuf" Zalewski. Aflplusplus. https://github.com/AFLplusplus/AFLplusplus, 2017.

[29] Kyung-suk Lhee and Steve J. Chapin. Buffer overflow and format string overflow vulnerabilities. *Softw. Pract. Exp.*, 33(5):423–460, 2003.

[30] Shaohua Li and Zhendong Su. Accelerating fuzzing through prefix-guided execution. *Proc. ACM Program. Lang.*, 7(OOPSLA1):1–27, 2023.

[31] Zhenpeng Lin, Yuhang Wu, and Xinyu Xing. Dirtycred: Escalating privilege in linux kernel. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022*, pages 1963–1976, 2022.

[32] Jianzhong Liu, Yuheng Shen, Yiru Xu, Hao Sun, and Yu Jiang. Horus: Accelerating kernel fuzzing through efficient host-vm memory access procedures. *ACM Trans. Softw. Eng. Methodol.*, 33(1):11:1–11:25, 2024.

[33] Valentin J. M. Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz, and Maverick Woo. The art, science, and engineering of fuzzing: A survey. *IEEE Trans. Software Eng.*, 47(11):2312–2331, 2021.

[34] Bart Miller. Virtualization without direct execution or jitting: Designing a portable virtual machine infrastructure. in emerging technologies for information systems, computing, and management. http://pages.cs.wisc.edu/~bart/fuzz/CS736-Projects-f1988.pdf, 1988.

[35] Shankara Pailoor, Andrew Aday, and Suman Jana. Moonshine: Optimizing OS fuzzer seed selection with trace distillation. In *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, pages 729–743, 2018.

[36] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. Test-case reduction for C compiler bugs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, pages 335–346, 2012.

[37] Moritz Schloegel, Nils Bars, Nico Schiller, Lukas Bernhard, Tobias Scharnowski, Addison Crump, Arash Ale Ebrahim, Nicolai Bissantz, Marius Muench, and Thorsten Holz. Sok: Prudent evaluation practices for fuzzing. In *IEEE Symposium on Security and Privacy, SP 2024, San Francisco, CA, USA, May 19-23, 2024*, pages 1974–1993, 2024.

[38] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. kafl: Hardware-assisted feedback fuzzing for OS kernels. In *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*, pages 167–182, 2017.

[39] Hao Sun, Yuheng Shen, Jianzhong Liu, Yiru Xu, and Yu Jiang. KSG: augmenting kernel fuzzing with system call specification generation. In *Proceedings of the 2022 USENIX Annual Technical Conference, USENIX ATC 2022, Carlsbad, CA, USA, July 11-13, 2022*, pages 351–366, 2022.

[40] Hao Sun, Yuheng Shen, Cong Wang, Jianzhong Liu, Yu Jiang, Ting Chen, and Aiguo Cui. HEALER: relation learning guided kernel fuzzing. In *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021*, pages 344–358, 2021.

[41] Xin Tan, Yuan Zhang, Jiadong Lu, Xin Xiong, Zhuang Liu, and Min Yang. Syzdirect: Directed greybox fuzzing for linux kernel. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security, CCS 2023, Copenhagen, Denmark, November 26-30, 2023*, pages 1630–1644, 2023.

[42] Google Team. Syscall description language. https://github.com/google/syzkaller/blob/master/docs/syscall_descriptions_syntax.md, 2017.

[43] Syzkaller team. Syzbot's web dashboard. https://syzkaller.appspot.com/upstream, 2020.

[44] Daimeng Wang, Zheng Zhang, Hang Zhang, Zhiyun Qian, Srikanth V. Krishnamurthy, and Nael B. Abu-Ghazaleh. Syzvegas: Beating kernel fuzzing odds with reinforcement learning. In *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, pages 2741–2758, 2021.

[45] Chenyuan Yang, Zijie Zhao, and Lingming Zhang. Kernelgpt: Enhanced kernel fuzzing via large language models. *CoRR*, abs/2401.00563, 2024.

[46] Michal Zalewski. Aamerican fuzzy lop (afl). https://github.com/google/AFL, 2016.

[47] Andreas Zeller and Ralf Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Trans. Software Eng.*, 28(2):183–200, 2002.

[48] Kyle Zeng, Yueqi Chen, Haehyun Cho, Xinyu Xing, Adam Doupé, Yan Shoshitaishvili, and Tiffany Bao. Playing for k(h)eaps: Understanding and improving linux kernel exploit reliability. In *31st USENIX Security Symposium, USENIX Security 2022, Boston, MA, USA, August 10-12, 2022*, pages 71–88, 2022.

IRHash: Efficient Multi-Language Compiler Caching by IR-Level Hashing

Tobias Landsberg
Leibniz Universität Hannover

Johannes Grunenberg
Leibniz Universität Hannover

Christian Dietrich
Technische Universität Braunschweig

Daniel Lohmann
Leibniz Universität Hannover

Abstract

Compilation caches (CCs) save time, energy, and money by avoiding redundant compilations. They are provided by means of compiler wrappers (Ccache, sccache, cHash) or native build system features (Bazel, Buck2). Conceptually, a CC pays off if the achieved savings by cache hits outweigh the extra costs for cache lookups. Thus, most techniques try to detect a cache hit early in the compilation process by hashing the (preprocessed/tokenized) source code, but hashing the AST has also been suggested to achieve even higher end-to-end savings, as the increased accuracy outweighs the additional parsing costs. Technically, all these CCs are currently limited to C or C-style languages.

In this paper we take the conceptual question of the "right" lookup level for compiler caches one step further onto the IR level. We provide IRHash, an IR-level CC for LLVM that not only offers higher accuracy than the previous works but can also support all languages with an LLVM backend.

We evaluate IRHash against Ccache and cHash based on the development history of 16 open-source projects written in C, C++, Fortran, and Haskell. With an average build time reduction of 19% across all C projects, IRHash provides better end-to-end savings than Ccache (10%) and cHash (16%), while additionally supporting more languages.

1 Introduction

Software development involves the frequent (re-)compilation of *translation units (TUs)* throughout the development cycle, consuming a considerable amount of time and energy (for both the developer and the machine).

To reduce these costs, content-based *compilation caches (CCs)*, such as Ccache [1] or sccache [2], aim to reuse build artifacts from previous builds. Both tools calculate a hash fingerprint of the TU (source file and included headers) and query their object-file cache. While effective, the resulting hash is sensitive to *any* source-code change, even superficial changes (i.e., formatting, coding style, comments) in any of the involved files result in a cache miss and the reinvocation of the compiler, even though the resulting object file would not be different. For higher accuracy, Ccache offers an option to calculate the hash after the C preprocessor has run, which normalizes the TU regarding *some* changes (on token level, e.g., white space formatting or commenting), but many superficial changes still lead to a cache miss, resulting in costly compiler invocations.

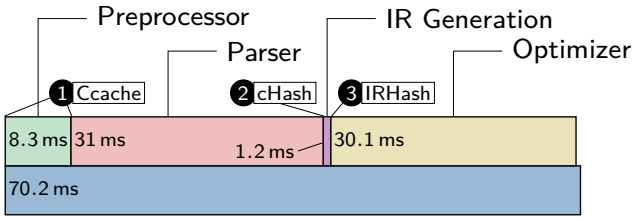


Figure 1: Build-time breakdown for a clean build of OpenSSL, which takes about 187 s and invokes the compiler 2 406 times. A single compiler invocation takes 70.2 ms on average, mainly comprising the duration for preprocessor, parser, IR generation, and optimizer. The markers show when different caching methods would detect redundant recompilations and stop the compiler invocation.

cHash [3] aims to overcome this issue by further normalization of the TU by calculating the hash after parsing, that is, on the level of the *abstract syntax tree (AST)*, where syntactically irrelevant changes are eradicated, and changes in unreferenced AST nodes can simply be ignored (e.g., an unreferenced type declaration). Due to its improved detection accuracy, cHash outperforms Ccache in most cases [3], even though the CC decision overhead is much higher: Fig. 1 shows this for OpenSSL, for which AST generation takes about four times longer than preprocessing. Apparently, further normalization to increase detection accuracy pays off.

About This Paper

In this paper we propose to go even further and solve the CC-problem on the level of the *intermediate representation (IR)*. Modern compilers, such as GCC or LLVM, generate a register-level IR from the AST to decouple language- and target-specific code generation and optimizations. As we can see in Fig. 1, this step takes only negligible extra time after AST generation (at least for OpenSSL). On the other hand, it might provide significant benefits: Besides providing higher *accuracy* (more superficial changes are eradicated), the IR is also relatively simple and language-agnostic: An IR-level CC approach thereby provides *generalizability* over all programming languages supported by the compiler suite. This is in stark contrast to the existing approaches, which are either bound to the C preprocessor (Ccache and sccache) or even the concrete AST-node types emitted by the compiler frontend (cHash). In particular, we claim the following contributions:

- We identify the IR-level as the most suitable level for CC with respect to *effectiveness* (end-to-end savings), *generalizability* (across languages), and *maintainability* (implementation effort).
- We propose IRHash, an IR-level compilation cache for LLVM IR and demonstrate its applicability for four front-end languages (C, C++, Fortran, Haskell).
- We demonstrate its accuracy and end-to-end savings in comparison to Ccache and cHash on the development history of 16 open-source projects. With an average build time reduction of 19% across all C projects, IRHash provides better end-to-end savings than Ccache (10%) and cHash (16%), while additionally supporting more languages.

The remainder of the paper is structured as follows: Sec. 2 describes the problem and limitations of existing caching mechanisms. Sec. 3 outlines the design and implementation of our IR-based caching approach. Sec. 4 evaluates its performance and versatility. Sec. 5 discusses the implications of our findings and potential improvements. Sec. 6 reviews related work, and Sec. 7 concludes this paper.

2 Problem Description

Source-code changes, whether during interactive development or condensed in a patch, are usually local and affect only a small portion of the program. Therefore, it is close at hand to recompile only the changed parts of the program while retrieving the rest from a *compilation cache (CC)* populated by previous compiler invocations. We define the *build-artifact reuse (BAR)* problem as the task of avoiding or speeding up unnecessary compilation steps by querying a CC.

However, in most programming languages, local changes can have a non-lexical, non-local impact on the compilation result. For example, changes to a C `struct`, which is declared in a header file, influence the resulting binary at every usage site. Also, if the BAR is too costly it can outweigh the savings of avoiding compilation steps.

Timestamp-based BAR, as introduced by `make` [4], addresses the problem at the TU level: For each TU, the file system timestamp of all compiler inputs is compared to the timestamp of the last compilation result. If all inputs are older than the last output, the compiler invocation is skipped and the last output is reused. This BAR method induces very low overheads, is entirely language agnostic, and has a high saving potential. However, it comes with two important downsides: (1) it is very sensitive to changes, especially with dense input-dependency graphs, and (2) it is a local strategy that is not able to reuse results across repositories or machines.

Content-based BAR [1]–[3], [5]–[7] addresses both shortcomings by calculating a fingerprint hash of the program that changes whenever the result would change. By keeping and querying a fingerprint–build-artifact CC we can skip compilation (steps).

For example, the Ccache [1] tool hashes either the original (direct mode) or the preprocessed source code (preprocessor mode). These modes differ in their *accuracy* and their *saving potential*: While the direct mode can skip the entire compiler invocation (70 ms for OpenSSL, see Fig. 1), it is sensitive to all textual changes. On the other hand, the preprocessed mode can ignore changes in inactive CPP blocks but requires executing the preprocessor, which takes 8.3 ms. More generally, the relative *end-to-end saving* of a BAR methods is:

$$\frac{T_{\texttt{saved}}}{T_{\texttt{total}}} = \frac{P_{\texttt{opt}} \cdot P_{\texttt{acc}} \cdot T_{\texttt{rem}} - T_{\texttt{over}}}{T_{\texttt{before}} + T_{\texttt{rem}}} \tag{1}$$

The BAR method will run after $T_{\texttt{before}}$ in the compilation process, and it will always introduce an overhead of $T_{\texttt{over}}$. If it can reuse a previous build artifact, it can skip the remaining $T_{\texttt{rem}}$ of the compilation, thereby saving time. However, it can only do so: (a) if the build artifact was produced before, which we introduce by $P_{\texttt{opt}}$, the probability that an optimal BAR[1] strategy with perfect accuracy would show, and (b) if the changed fingerprint does actually indicate a changed compilation result, for which we use the accuracy probability $P_{\texttt{acc}}$. Please note, that $P_{\texttt{opt}}$ depends on the project, the encountered change, the usage scenario, and the CC configuration (cache size). Only $P_{\texttt{acc}}$ is specific to the used BAR method.

A relevant question is, at which point in the compilation process should a BAR method run to maximize the end-to-end saving? For example, cHash [3] occupies another spot in this trade-off space: By hashing at the AST level, cHash can ignore changes to type declarations (e.g., in a header file) if that type was not referenced. However, for OpenSSL, its average $T_{\texttt{rem}}$ is only 31.3 ms per TU, while Ccache's $T_{\texttt{rem}}$ is at 62 ms per TU (89 %).

As the compiler progresses, $T_{\texttt{rem}}$ naturally decreases; however, as we get closer to the final binary, we expect $P_{\texttt{acc}}$ to

[1] Assuming reproducible builds, an optimal BAR method would invoke exactly those compiler runs that would result in a changed binary program.

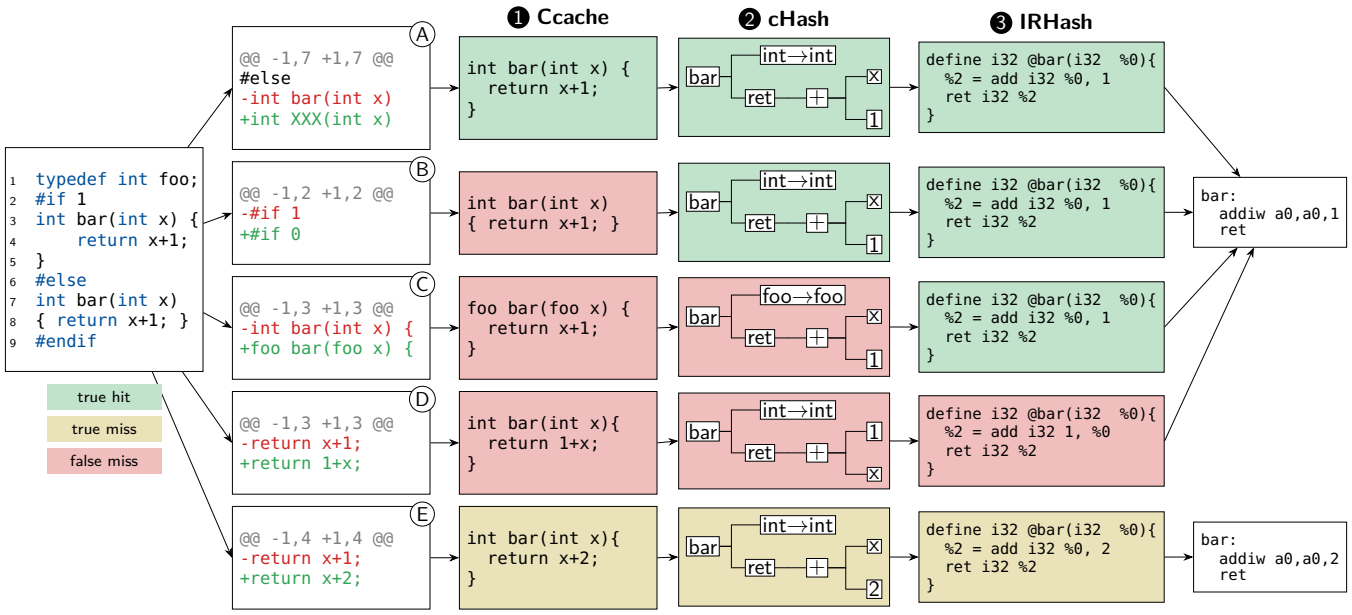


Figure 2: Examples for capabilities and limitations of BAR methods when confronted with different source-code changes

improve accordingly. Therefore, $T_{\texttt{over}}$, which scales with the amount of data and the complexity of the hash procedure, is quite important.

Directly tied to the hash-procedure complexity, and also deciding on its widespread adoption, is the language dependence of a BAR method. For example, Ccache's direct mode only has to hash a byte array, while cHash requires a hash rule for every language-specific AST-node type. This might also be why cHash has not gained widespread adoption and still only supports the C language.

3 IRHash: An IR-Based Compilation Cache

With IRHash, we solve the BAR problem at a different step of the compilation process, right after the IR generation. In the following, we will highlight the conceptual benefits, illustrate situations where IRHash can outperform the other BAR techniques, and touch on the implementation of our prototypical LLVM plugin.

3.1 Conceptual Benefits

Using a hash of a program's IR as a fingerprint to predict changed compilations comes with four key benefits:

Negligible smaller $T_{\texttt{rem}}$ than cHash. Preprocessing and parsing take a considerable share of the compilation process, as we can see in Fig. 1. In comparison, as done by a single tree traversal, generating the IR from the AST is quite fast, reducing $T_{\texttt{rem}}$ only slightly. For example, for OpenSSL, $T_{\texttt{rem}}$ is at 30.1 ms for the IR level and 31.3 ms for the AST-level. We will discuss this in more detail in Sec. 4.3.

Higher $P_{\texttt{acc}}$ At the IR level, many language constructs that are only syntactic sugar (C: `typedef`) or act as type-level safeguards (C++: `explicit`) are removed by IR code generation. The IR generation can also perform some ad-hoc optimizations (e.g., constant folding). Therefore, IR-level should have a higher $P_{\texttt{acc}}$ than AST-level hashing, which already outperforms source-level hashing. We will quantify this higher $P_{\texttt{acc}}$ in Sec. 4.2.

Low $T_{\texttt{over}}$ overhead The data structures that store the IR are designed for frequent traversal to speed up the optimization steps. Also, they are conceptually simpler as they are a sequential program (list of lists) for a virtual machine compared to the AST, which captures the syntactic structure of the program recursively (as a tree). Unlike source-level hashing, which has to hash all (preprocessed) compiler inputs, IR-level hashing can also be faster as only the relevant parts of the input are hashed. In Sec. 4.3, we will demonstrate that IR-level hashing actually has a lower $T_{\texttt{over}}$.

Simple and widely-applicable implementation Due to the conceptual simplicity of the IR, which is comprised of only a few instruction types, IR-level hashing is simpler to implement than AST-level hashing. Since the IR is a language-*independent* abstraction that connects language front-ends, optimization steps, and machine-specific back-ends, IR-level hashing is directly usable for different programming languages. We will argue for IRHash's simplicity in Sec. 3.3 and justify our applicability claim with four programming languages and 16 open-source projects (see Tab. 2).

Given these arguments, we believe that IR hashing is conceptually the most suitable point in time to perform BAR.

C++ Classes	Source Lines
`llvm::Module`	27
├─ `llvm::Function`	25
│ ├─ `llvm::BasicBlock`	4
│ │ ├─ `llvm::Instruction`	71
│ │ │ ├─ `llvm::Constant`	93
├─ `llvm::GlobalVariable`	24
│ ├─ `llvm::Type`	109
Caching Logic	208
Total	561

Table 1: Implementation effort for IR-level hashing

3.2 Reasons for Better Accuracy

To give a better understanding of $P_{\texttt{acc}}$, we will exemplify the capabilities and limitations of the different BAR methods with the five source-code changes Ⓐ-Ⓔ in Fig. 2. We assume that the original code on the left was already compiled and inserted into the CC and we compile for RISC-V, as RISC-V allows us to show a limitation of CCs. Please note, that $P_{\texttt{acc}}$ is $(1 - \frac{\#\text{false miss}}{\#\text{runs}})$.

Ⓐ is a change to the dead preprocessor block (line 6-9). If used in preprocess mode, Ccache can ignore the change and correctly have a cache hit, reusing the previous result. Naturally, all the following steps also have a cache hit.

Ⓑ toggles the `#if` and selects the previously dead block. Since the `bar()` function is formatted differently, Ccache will falsely have a cache miss and continue compilation. As such syntactic differences are not visible in the AST, cHash has a cache hit.

Ⓒ replaces the `int` type with the equivalent `foo` type alias. While C `typedefs` are mere syntactic sugar, they are still visible in the AST, provoking a false cache miss with cHash. Due to type-alias erasure during IR generation, IRHash correctly has a cache hit.

Ⓓ swaps the terms of the sum. This change persists until the IR level, provoking a cache miss for IRHash. However, this is a false miss, as the RISC-V backend uses the add-with-immediate instruction and swaps the operands to the original order.

Ⓔ is an actual semantic change that results in a different binary output, requiring the full compilation process.

The different BAR methods remain, to a differing degree, stable hashes in the presence of irrelevant changes.

3.3 Implementation

Our IRHash prototype comes as an LLVM plugin that introduces a new IR-level compiler pass that has access to LLVM's internal data structures. Thus, IRHash can be loaded into all LLVM-based compilers and our IRHash pass can be scheduled at any point of the compilation process. For BAR, IRHash should run directly after the IR generation to maximize $T_{\texttt{rem}}$. Conceptually, the plugin consists of two parts: IR hashing and CC logic.

IR Hashing LLVM IR captures the program semantic as a set of per-function control-flow graphs with basic blocks as nodes. The IR code generation already flattens all language-level logic constructs (methods, lambdas, co-routines,...) into LLVM functions. Therefore, IR is stored as a shallow hierarchy of nested C++ objects (see Tab. 1): A TU is a list of functions and global variables. A function is a list of basic blocks, which themselves are lists of instructions. Each instruction has an opcode and multiple operands. As LLVM IR is always in SSA format [8], operands are either a literal constant or (the result) of another instruction. Global variables, functions, instructions, and constants have types, which only describe the data layout.

In Tab. 1 we see that it only takes a few hundred source lines to implement the necessary hash operators: For the TU hash, we include the hash of global variables and functions. The global-variable hash includes name, type, initializer, and attributes. For functions we hash the name, the prototype, attributes, and the hash of basic blocks and subsequently instructions. Besides opcode and operands, the instruction hash also includes certain attributes that influence the binary (e.g., allocation alignments, atomicity). With inline assembler we hash the string representation, and complex types are hashed recursively down to their scalar components.

We deliberately chose not to hash a serialized version of the IR (in textual or LLVM bitcode form), as this has two significant drawbacks: (1) We would lose the fine-grained control over the data included in the hash. For example, IRHash by default does not include debug information in the hash, which means purely syntactical changes do not trigger a rebuild. However, in scenarios where up-to-date debug information is required, this could be adjusted easily.

(2) IR serialization takes a considerable amount of time and is not necessary for many compiler runs. Nevertheless, if LLVM serializes the IR anyway, for example, for ThinLTO [9], an IR hash could be calculated using that.

Caching Logic With the IR hash at hand, the IRHash pass performs the CC lookup: Like Ccache, IRHash stores build artifacts (i.e., object files) in a cache directory with the file name matching the hash. On a hit, the cached build artifact is hard linked to the target location and the compilation is terminated. Also, the timestamp of the object file is updated to allow timestamp-based BAR methods to avoid downstream operations (like linking). On a cache miss, the compilation continues, and the resulting object file is put into the cache.

IRHash is implemented with a total 561 lines of C++ code, while cHash requires 1200 source lines for the hash operator alone [3]. Please note that IRHash supports multiple front-

end languages, while cHash only supports C. As Ccache is a production-ready CC with a multitude of features, its 19 000 source-code lines are not comparable. However, IRHash can be combined with Ccache (see Sec. 4.5) or could act as a fingerprinting module.

4 Evaluation

In the evaluation we demonstrate that IRHash works correctly and efficiently across languages on the development history of 16 open-source projects written in C, C++, Fortran, and Haskell. We also show that the IR level is best suited to solve the BAR problem by comparing IRHash with tools working at the source code and AST levels. For this purpose, we chose Ccache for the source-code level due to its popularity and cHash for the AST level, as it is the only tool known which works at this level. We determine the parameters of Equation 1 for these CCs and projects and conclude with a real-world end-to-end-performance benchmark.

4.1 Test Cases

For our evaluation we selected the open-source projects listed in Tab. 2. This selection spans a wide range of project sizes, from smaller codebases, like ShellCheck (16 kLOC) and LAPACK (779 kLOC) to larger ones, such as PostgreSQL (1 019 kLOC) and Linux (26 256 kLOC). These projects also represent diverse domains, including interpreters (Bash, CPython, Lua), database systems (PostgreSQL, SQLite), cryptography libraries (Mbed TLS, OpenSSL), static analysis tools (Clazy, GammaRay, ShellCheck), frameworks and toolkits (Asio, SDL), a mathematical library (LAPACK), a lightweight C library (musl), an x86 emulator (Bochs), and the Linux operating system kernel. For comparison with the textual and AST level, our evaluation focuses on C but to demonstrate applicability to other languages, some of these projects are written in C++ (Bochs, Asio, Clazy, GammaRay), Fortran (LAPACK) and Haskell (ShellCheck).

For our experiments we examined the development history of the projects up to the described version as a source of real-world changes required for incremental builds. The number of commits considered was limited to the 100 most recent ones. Together with the selection of projects, this should result in a diverse and representative set of changes. Commits failing to build in the baseline (without any caches) were filtered out. All measurements were performed on a system with an Intel® Core™ i7-12700 processor with Turbo Boost disabled for reproducibility and 32 GB of memory. The size of the CCs was not limited. All projects were tested using their default configurations, except for replacing the compiler with Clang and enabling the tested CCs. For SQLite we disabled amalgamation[2], a process in which all source code is concatenated into a single file. Amalgamation is mainly done for deployment and optimization purposes and is not intended for development. Asio as a header library is not compilable itself, which is why we have compiled the example code, the default target of the project. For LAPACK, written in Fortran, we compiled using Flang, LLVM's Fortran frontend [10]. ShellCheck, a project written in Haskell, was compiled using the LLVM backend of GHC 9.12.1 [11]. Due to an incompatibility with newer GHC versions of the earlier commits in its history, ShellCheck has only 59 successfully building (and therefore analyzed) commits. In contrast, there are usually no failing (clean) builds for the other projects and at most six for GammaRay. All in all, based on this diverse selection of projects and commits, we are confident that our results are representative of a wide range of software projects.

[2] https://sqlite.org/amalgamation.html

Table 2: Overview of Evaluated Projects

Project	Language	Version	LOC
Linux [12]	C	6.11	26 255 658
OpenSSL [13]	C	3.4.0	621 158
SQLite [14]	C	3.47.0	297 234
Lua [15]	C	5.4.7	21 879
PostgreSQL [16]	C	17.0	1 018 762
CPython [17]	C	3.13.0	672 595
Bash [18]	C	5.2	130 759
Mbed TLS [19]	C	3.6.2	166 172
SDL [20]	C	2.30.8	351 683
musl [21]	C	1.2.5	95 034
Bochs [22]	C++	2.8	289 270
Asio [23]	C++	1.32.0	182 108
Clazy [24]	C++	1.12	25 035
GammaRay [25]	C++	3.1.0	101 798
LAPACK [26]	Fortran	3.12.0	779 202
ShellCheck [27]	Haskell	0.10.0	15 672

4.2 Validation (P_{opt}, P_{acc})

First, we validated the implementation of IRHash by performing clean builds for 100 commits each across 16 projects. We compared the predictions of Ccache, cHash, and IRHash against object file hashes and measure the compilation time with caching disabled to determine the overhead. In total, 1 289 086 translation units were compiled.

Most importantly, IRHash, like the other tools, did not cause any false hits, which means that whenever an object file changed, the IRHash was also different. The property of correctness is mandatory for a CC. If developers cannot be sure that their changes have been compiled, i.e., a CC leads to incorrect results, it is practically useless. Consequently, Ccache and cHash also offer this property. Full completeness, i.e., no false misses ($P_{\mathsf{acc}} = 1$), on the other hand, is desir-

Table 3: The average accuracy ($P_{\texttt{acc}}$) and $P_{\texttt{opt}}$ per commit for the evaluated projects, commits, and CCs. Values not available due to lack of language support by a CC are marked with "-".

		$P_{\texttt{acc}}$ (%)		
Project	$P_{\texttt{opt}}$ (%)	Ccache	cHash	IRHash
Linux	99.91	97.89	98.14	99.99
OpenSSL	99.96	97.63	99.01	100.00
SQLite	98.86	93.66	99.01	99.82
Lua	97.23	85.26	97.03	99.21
PostgreSQL	99.88	97.99	100.00	100.00
CPython	98.97	98.74	99.48	99.78
Bash	97.09	93.23	99.58	99.97
Mbed TLS	99.37	93.62	99.76	99.97
SDL	99.31	95.78	99.94	99.93
musl	98.98	99.35	99.91	100.00
Average (C)	98.96	95.32	99.19	99.87
Bochs	98.95	97.29	-	98.81
Asio	96.08	59.92	-	96.72
Clazy	99.41	95.83	-	99.93
GammaRay	99.55	99.74	-	99.99
Average (C++)	98.50	88.20	-	98.86
LAPACK	99.80	-	-	99.99
ShellCheck	75.26	-	-	100.00

able but not claimed by any approach. As Tab. 3 shows, the accuracy $P_{\texttt{acc}}$ varies; the reasons are discussed in Fig. 2.

The results show that IRHash is generally one to two orders of magnitude more accurate than Ccache, cHash is in between but closer to IRHash. For example, for Mbed TLS with Ccache, 6.38% redundant builds are performed, with cHash 0.24%, and with IRHash only 0.03%. It is noticeable that both Lua and Asio have a high rate of false misses. In the case of Lua this is due to the fact that it was refactored frequently, sometimes in such a way that the code was only the same after optimization. Asio as a header library often introduces functions that are not used, a weakness of Ccache. Also, in the commit range we examined, implicit constructs were made explicit, and `std::move()` was inserted in places where the optimizer already had done so, leading to false misses for IRHash. The fact that many of the false misses are caused by semantic changes, which are only eliminated by optimizations, underlines the high suitability of the IR level for the accuracy of caching.

Another interesting observation is that cHash is more accurate for SDL than IRHash. This is due to an optimization of cHash: Unlike IRHash, cHash does not hash the types of structs themselves but their use only, i.e., the offset of the accesses and the size (for `sizeof()`). If a new field is added at the end of a struct that is not used and the size of the struct is not used or changed in a TU, the hash does not change either. In the future this optimization could be implemented in IRHash as well.

In contrast to the other CCs, IRHash also supports languages apart from C/C++. The last two experiments confirm that IRHash works for other compilers with an LLVM backend (Fortran and Haskell). With a $P_{\texttt{acc}}$ of 99.99% and 100.00%, it performs equally as well as for C/C++.

Note that $P_{\texttt{opt}}$ for ShellCheck (75.26%) is remarkably low. There are two reasons for this. Firstly, ShellCheck has relatively few translation units (27), thus modifying only a few files is already a significant share. Secondly, the compiler does not always work fully deterministically for each TU. The main problem is that internal identifiers are made globally unique (across all TUs). This process is nondeterministic. In addition, the identifiers are enumerated, which can lead to changes in one TU changing identifiers in another. This is further exacerbated in incremental builds where the number of TUs to be compiled varies. The GHC maintainers confirmed this behavior as a known bug and are, at the time of writing, working on a fix. [28]–[31]

All in all, the results demonstrate that the IRHash approach works correctly, effectively, and significantly more accurately than Ccache, especially for C++ projects. Compared to cHash, the difference in accuracy is less pronounced. This is primarily due to two factors: (1) changes in the AST frequently result in changes to the IR, and (2) the evaluated changes, derived from the projects' development history, often genuinely alter behavior. Nonetheless, the experiment confirms the expectation of IR Hashing providing a higher $P_{\texttt{acc}}$.

4.3 Performance Characteristics ($T_{\texttt{over}}$, $T_{\texttt{rem}}$)

We performed clean builds with deactivated caching for the same commits as before to quantify each CC's overhead. Similarly, 1 289 086 translation units were compiled. The resulting $T_{\texttt{over}}$ and $T_{\texttt{rem}}$ are summarized in Tab. 4.

The measurements show that Ccache's overhead is consistently the highest, with an average of 548.8 ms per commit for Linux, followed by cHash (236.1 ms) and IRHash (2.0 ms). The results for Linux highlight the limitations of Ccache. It invokes the preprocessor during cache misses, which in Linux entails resolving a multitude of complex and deeply nested macros. By contrast, cHash benefits from the fact that the macros are already resolved and a lot of code discarded. In the code generation phase, further complexity is resolved, enabling IRHash to process the IR code much faster. This pattern is observed consistently across all tested projects, though the specific performance characteristics vary.

However, it is noticeable that the absolute overhead of IRHash for the C++ projects Asio (14.5 ms) and Clazy (26.1 ms) is higher compared to C projects of their size, e.g., SDL (1.1 ms). This difference stems from the extensive code generated due to the use of templates. For instance,

Table 4: The average overhead ($T_{\texttt{over}}$) and time remaining after hash calculation ($T_{\texttt{rem}}$) per TU for the evaluated projects, commits, and CCs. Values not available due to lack of language support by a CC are marked with "-".

	$T_{\texttt{over}}$			$\frac{T_{\texttt{over}}}{T_{\texttt{total}}}$			$\frac{T_{\texttt{rem}}}{T_{\texttt{total}}}$		
Project	Ccache	cHash	IRHash	Ccache	cHash	IRHash	Ccache	cHash	IRHash
Linux	548.8 ms	236.1 ms	2.0 ms	34.8%	14.7%	0.1%	65.2%	1.4%	15.3%
OpenSSL	97.7 ms	8.7 ms	0.7 ms	38.0%	3.7%	0.3%	61.8%	36.1%	38.9%
SQLite	59.2 ms	4.4 ms	1.5 ms	17.0%	1.4%	0.5%	82.9%	76.5%	76.4%
Lua	52.0 ms	3.8 ms	1.2 ms	18.7%	1.5%	0.5%	81.2%	77.6%	76.7%
PostgreSQL	113.5 ms	9.1 ms	1.4 ms	29.2%	2.5%	0.4%	70.7%	51.9%	53.3%
CPython	151.1 ms	22.3 ms	3.3 ms	19.7%	3.0%	0.4%	80.2%	62.1%	63.2%
Bash	121.7 ms	3.8 ms	1.1 ms	28.8%	1.3%	0.4%	71.1%	62.0%	62.0%
Mbed TLS	63.4 ms	4.8 ms	0.9 ms	29.4%	2.6%	0.5%	70.4%	61.8%	62.9%
SDL	158.4 ms	36.8 ms	1.1 ms	31.9%	7.8%	0.2%	68.0%	31.9%	39.1%
musl	33.9 ms	0.9 ms	0.1 ms	52.9%	2.9%	0.4%	46.2%	50.2%	51.1%
Average (C)				30.0%	4.1%	0.4%	69.8%	51.1%	53.9%
Bochs	86.0 ms	-	1.7 ms	17.3%	-	0.4%	82.6%	-	55.5%
Asio	447.5 ms	-	14.5 ms	6.9%	-	0.2%	93.1%	-	30.7%
Clazy	510.0 ms	-	26.1 ms	5.3%	-	0.7%	94.7%	-	53.5%
GammaRay	272.5 ms	-	2.4 ms	11.4%	-	0.1%	88.6%	-	12.1%
Average (C++)				10.2%	-	0.3%	89.7%	-	37.9%
LAPACK	-	-	0.5 ms	-	-	0.2%	-	-	41.7%
ShellCheck	-	-	60.7 ms	-	-	1.5%	-	-	85.5%

Asio (182 kLOC) generates 387 MiB of IR code, while SDL (352 kLOC) generates only 28 MiB. As this greatly increases the total compiler runtime, it is not reflected in the relative overhead, both for Ccache and for IRHash. Instead, it mainly affects the compiler frontend, which has to expand the templates. The overhead for Asio is 6.9% and 0.2% respectively, for Clazy 5.3% and 0.7% respectively, and for SDL 31.9% and 0.2% respectively. ShellCheck is special in that Haskell's GHC generates textual LLVM IR and then calls LLVM's `opt` to optimize the generated IR. With 439 MiB even more code is generated than with Asio, which explains the high overhead. In addition, IRHash only runs as part of `opt`, which leads to a $T_{\texttt{total}}$ that consists mainly of optimizations, since the compiler frontend has been executed already and the lowering happens in a later step. This leads to a comparatively high relative overhead and the fact that only the optimization phase (not the lowering) can be cached.

Regarding the saving potential $T_{\texttt{rem}}$, Ccache performs best because it starts the hash calculation the earliest. Interestingly, IRHash often completes hash computation earlier than cHash despite the latter starting earlier (as soon as the AST is generated). For example, for OpenSSL $T_{\texttt{rem}}$ is 61.8% when using Ccache, for cHash it is 36.1%, and for IRHash 38.9%. This behavior is due to an implementation detail in modern compilers, particularly Clang: To reduce memory usage, the compilation process handles each top-level declaration independently and sequentially. As a result, the code generation phase for the first declaration begins even before the AST for subsequent declarations is generated. Consequently, cHash starts exactly as much earlier as the code generation phase for the last top-level declaration takes and IRHash often compensates for this by completing the hash computation more quickly. Again, ShellCheck is special: Because its $T_{\texttt{total}}$ consists only of `opt`, the hashing pass runs directly at the beginning, leading to a high $T_{\texttt{rem}}$.

Overall, IRHash offers a trade-off: the hash is calculated later than when using Ccache but is significantly more accurate. Compared to cHash it is equally fast but more precise and not limited to C. The low overhead of the hash calculation also shows that our implementation is sufficiently fast, although there is still plenty of room for improvement. These results are a strong indicator for IR being better suited to solve the BAR compared to the AST.

4.4 End-to-End Performance ($T_{\texttt{total}}$, $T_{\texttt{saved}}$)

The end-to-end performance benchmark is intended to reflect the use case of a typical developer who frequently changes and builds the project. To simulate this scenario, we incrementally built 100 (+1 initial) commits from the development history of each of the evaluated projects. Non-building commits, along

Table 5: The optimization level, number of successfully building commits (in baseline) together with the average build time ($T_{\texttt{total}}$) and relative end-to-end savings per commit for the evaluated projects, commits, and CCs. The fastest cache for each project is highlighted in green. If a combination of caches is faster than this cache for a project, it is highlighted in gray. Values not available due to lack of language support by a CC are marked with "-".

				$\frac{-T_{\texttt{saved}}}{T_{\texttt{total}}}$				
Project	Opt	Commits	$T_{\texttt{total}}$	Ccache	cHash	IRHash	Ccache+cHash	Ccache+IRHash
Linux	O2	100	25.1 s	0.1%	3.0%	−3.1%	3.2%	−3.4%
OpenSSL	O2	100	5.7 s	−16.9%	−9.5%	−20.8%	−18.8%	−20.2%
SQLite	O2	98	42.9 s	−4.8%	−3.8%	−4.4%	−4.3%	−4.7%
Lua	O2	100	1.0 s	−44.9%	−49.6%	−52.8%	−52.4%	−56.1%
PostgreSQL	O2	100	1.8 s	−7.9%	−11.5%	−11.9%	−8.4%	−9.3%
CPython	O3	100	2.3 s	−14.9%	−19.1%	−22.0%	−16.9%	−20.2%
Bash	O2	89	1.4 s	−12.7%	−29.5%	−26.4%	−18.8%	−20.8%
Mbed TLS	O2	98	1.1 s	−2.1%	−25.2%	−23.3%	−22.9%	−21.8%
SDL	O3	88	2.3 s	14.3%	−7.5%	−20.3%	10.6%	−2.1%
musl	O2	100	1.2 s	−6.8%	−4.6%	−5.3%	−7.1%	−7.7%
Average (C)				−9.6%	−15.7%	−19.0%	−13.6%	−16.6%
Bochs	O3	98	1.9 s	3.6%	-	−12.6%	-	−8.0%
Asio	O3	96	41.0 s	−6.8%	-	−28.5%	-	−29.5%
Clazy	O3	95	7.1 s	−0.6%	-	−25.9%	-	−26.5%
GammaRay	O3	94	5.8 s	−20.2%	-	−10.6%	-	−26.4%
Average (C++)				−6.0%	-	−19.4%	-	−22.6%
LAPACK	O2	100	1.7 s	-	-	−12.4%	-	-
ShellCheck	O2	58	142.7 s	-	-	0.8%	-	-

with their subsequent commits used for a new clean build, were excluded from the evaluation. Tab. 5 presents the average performance improvements per commit for Ccache, cHash, and IRHash, compared to builds without any CC.

Across all 16 projects, regardless of optimization level, project size, or programming language, the results show that, with few exceptions, IRHash consistently achieves the fastest build times. In four of these exceptions, the fastest CC is only marginally faster. For SQLite and musl, Ccache, with improvements of 4.8% and 6.8%, is slightly faster than IRHash (4.4% and 5.3%). For Bash and Mbed TLS, cHash marginally outperforms IRHash (29.5% vs. 26.4% and 25.2% vs. 23.3%). In one other case, GammaRay, Ccache can utilize its higher $T_{\texttt{rem}}$ and offer more pronounced improvements (20.2% vs. 10.6%).

ShellCheck with a 0.8% slowdown is the only project where IRHash cannot improve the average build time. As mentioned in Sec. 4.2, GHC is not fully deterministic, which is amplified in incremental builds. This leads to frequent changes and cache misses. In addition, GHC uses fingerprints for the recompilation decision [29], which further increases the probability that a compiler call will actually result in a changed object file. Although IRHash still works absolutely correctly, it does not make sense to use it under these circumstances. Due to very few cache hits, the savings cannot outweigh the overhead.

In the end, it always depends on the time behavior of the cache and whether it is accurate enough for a specific change. If a change can be cached by a technique with higher $T_{\texttt{rem}}$, this will always be the fastest option. However, our evaluation has shown that, for a wide range of software projects and changes, caching at the IR level is the better choice. From these three CCs, IRHash provides competitive results most consistently.

4.5 Combining Approaches ($T_{\texttt{total}}$, $T_{\texttt{saved}}$)

Since a low $T_{\texttt{rem}}$ is a weakness of IRHash, combining it with Ccache by using both Ccache's compiler wrapper and the IRHash compiler plugin at the same time might lead to even better results. Therefore, Tab. 5 also presents the average performance improvements per commit for the combinations of Ccache with IRHash and, for comparison, Ccache with cHash. We did not measure the combination cHash+IRHash, as IRHash is superior to cHash both in terms of $P_{\texttt{acc}}$ and $T_{\texttt{rem}}$. The following applies to the combined cache: A miss in the first cache (Ccache) results in a query for the second cache (cHash or IRHash), which means that the data is stored in both

caches. If there is a cache hit in the first cache, the second cache is not queried.

The results show that whenever IRHash was faster than cHash, the combination with Ccache is also faster. Interestingly, this also applies to Bash, where cHash is faster.

In six out of 16 cases, Ccache+IRHash's improvements are the best compared to all other CCs and combinations thereof. In five of these cases (Linux, Lua, musl, Asio, Clazy) the improvement is small, for example, 3.4% over 3.1% with IRHash alone for Linux.

An interesting case is GammaRay, as Ccache alone speeds up the build by 20.2% and IRHash only speeds up the process by 10.6%, but together these tools achieve a speedup of 26.4%. While the speedup is more pronounced, this behavior is fundamentally no different to the other cases in which the combination is faster than the individual caches. However, when combining caches, only the speed can increase, not the hit rate. This is because a cache hit with Ccache means that after preprocessing the compiler input has not changed. Consequently, a deterministic compiler must always produce the same AST and IR code so that a hit in Ccache must also always mean a hit for cHash and IRHash. The advantage must therefore be that in cases where Ccache has a hit, calculating the more precise hash of cHash or IRHash is skipped. In cases where Ccache has a miss, the other methods may still save time due to their higher accuracy.

In cases where Ccache+IRHash delivers the best improvement, IRHash alone is often already highly effective and, therefore, a safe bet to use. Depending on the project and the change patterns ($P_{\texttt{opt}}$), it may make sense to use Ccache additionally.

4.6 Result Summary

At this point we want to conclude that IR-level hashing is (a) effective (end-to-end savings), (b) generalizable across languages, and (c) easy to implement and maintain.

IRHash outperforms Ccache and cHash or comes close for most projects (see Tab. 5). The reason for this is its higher $P_{\texttt{acc}}$ (see Tab. 3). While it has a comparable $T_{\texttt{rem}}$ than cHash, its $T_{\texttt{over}}$ is significantly lower, allowing it to outperform Ccache (see Tab. 4).

Our evaluation with 16 open-source projects across four programming languages demonstrates its easy and broad applicability. As we argued in Sec. 3.3, IR-level hashing is also easier to implement, for example, in the case of IRHash with a few hundred lines of code.

All in all, we think that IR-level hashing should become a standard extension to modern compiler frameworks.

5 Discussion

We have now described IRHash, our CC operating at the IR level, and have demonstrated the advantages over approaches operating at the source code and AST levels. In this chapter, we will discuss potential threats to the validity of our results, the generalizability of our approach, and open aspects in more detail.

5.1 Threats to Validity

To help ensure the internal validity of our study, we evaluated IRHash using the development histories of 16 open-source projects. The projects were chosen from different application domains: interpreters, database systems, cryptography libraries, static analysis tools, frameworks and toolkits, a mathematical library, a lightweight C library, an x86 emulator, and an operating system kernel. They are implemented in C, C++, Fortran, and Haskell.

Due to the significant differences between them, a large coverage of language constructs is given. In addition, IRHash operates on IR, which is intentionally kept simple, thereby considerably improving coverage further. In combination with the number of 1 289 086 compiled TUs, a diverse set of IR constructs was generated. IRHash has not missed a single change as verified against the most exact metric of binary equivalence.

To ensure external validity, we used the same variety of projects for the performance measurements. The experiments consistently yielded strong results, with IRHash not only being more precise than Ccache and cHash but also more effective in reducing compilation time. Moreover, due to the significant differences between C/C++, Fortran, and Haskell, we successfully demonstrated that the approach is language independent.

Overall, we are confident that our results are both correct and transferable to other projects and languages.

5.2 Open Questions

While we evaluated IRHash thoroughly, there are still possibilities for further research.

Source-code changes have different extents and impacts (reflected in different $P_{\texttt{opt}}$ values). For interactive development, developers will benefit even more from IRHash's faster turnaround times, as usually only a few lines are changed before recompilation. Commits or patches naturally require more recompilation effort (medium $P_{\texttt{opt}}$ values) and induce their costs, most importantly, on continuous-integration servers [32]. While we expect even lower $P_{\texttt{opt}}$ values between software versions, IRHash could help to reuse results between the different variants of statically-configured software product lines [33]. For our evaluation we deliberately chose to test IRHash with commits from existing Git histories. These changes have a medium scope and give us realistic change sets developed by independent third-party developers.

Multiple IR levels are gaining importance in current compiler design. For example, Rust's `rustc` compiler uses

three IR levels (High-, Medium-IR, and LLVM-IR) to perform incremental lowering. With MLIR [34], LLVM now also provides a general means to express different IR levels. However, with multiple IR levels, the gap between AST generation and LLVM-IR hashing increases, reducing $T_{\texttt{rem}}$. Therefore, the question arises at which IR level should results be cached. We consider this a topic for further research.

More fine-grained caching We also considered whether more fine-grained caching (i.e., function level) could lead to even better results. While IRHash currently fingerprints and caches the entire TUs, it can also provide per-function hashes. However, per-function caching is challenging, as interprocedural optimizations, which LLVM's fix-point optimizer schedules early, cause information to flow between functions. For example, if the compiler inlines function A into function B, a change to A will also change the binary form of B. Without fine-grained tracking of such dependencies, per-function caching, as already proposed by Sathyanathan *et al.* [35], risks incorrect compilation results. Such tracking, however, requires substantial compiler modifications, that are not necessary for IRHash, which is less intrusive and easier to validate.

IR normalization Another interesting direction to explore could be to combine *IR normalization*[3] with IRHash. By normalizing the IR (e.g., topological sorting of basic blocks) before hashing, even more non-semantic changes could be equalized, potentially leading to a higher $P_{\texttt{acc}}$. We refrained from enabling normalization as (1) bugs in the IR normalization would result in caching incorrect results, (2) normalization itself induces considerable overhead ($T_{\texttt{over}}$), and (3) our observed $P_{\texttt{acc}}$ was already quite high (see Tab. 3).

6 Related Work

Reducing the end-to-end costs of software recompilation by incremental builds is a decades-old mission that was even once considered a motivating factor towards modularization [36]. Yet the question which modules actually need to be recompiled based on a change in some other module remains an ongoing challenge: C and its descendants delegate modularization to an idiomatic usage of the preprocessor as well as the file system and the task of incremental builds to a language-agnostic build system (e.g., make [4]). Hence, changes are only detected and handled on the granularity of whole files, leading to many redundant recompilations. This has been identified as particularly problematic for so-called "God" header files, which trigger many other modules to be (redundantly) recompiled in case of a change [37], [38].

CCs, such as Ccache [1] and sccache [2] mitigate such recompilation costs by stepping in as a proxy compiler that detects a redundant build early in the compilation process. FASTBuild [39] goes one step further and incorporates a CC directly into the build system. All of these systems work on (preprocessed) C/C++ source-code level and can skip redundant builds only for irrelevant lexical changes. For languages with true module support (i.e., modules as first-class syntactic entities), several solutions have been proposed to prevent redundant builds at higher accuracy [40]–[42] by tracking the actual symbol-level interactions between exporting and importing modules. cHash [3] achieves this for C by shifting the detection onto the AST level. With IRHash, symbol-level dependencies are basically dealt with automatically in a language-independent way, as all symbol-level interactions have already been resolved on the IR level.

Parallelizing the compilation process is another common approach to reduce end-to-end build times. Systems like distcc [43], Icecream [44], CloudBuild [7], Bazel [5], and Buck2 [6] distribute compilation tasks over the local network or cloud. While parallelization reduces waiting times, it does not reduce the overall workload and energy costs. Therefore, all of these systems also provide means to integrate hash-based CCs, usually with the option to share the cache across branches, projects, and developers. It would be relatively easy to integrate IRHash into these settings if the respective compiler uses an IR.

For such larger *continuous integration (CI)* settings, AST-based (i.e., symbol-level) redundancy detection has also been suggested as a means to reduce the post-compilation overhead by preventing redundant regression testing [45]. Here, IRHash might further increase the accuracy and language support.

7 Conclusion

Compilation caches (CCs) reduce the costs of incremental software builds by interrupting redundant compilations early in the process in favor of a cached result from a previous build. Where exactly in the multi-stage process of modern compilers such caches should be implemented is a trade-off between effectiveness (end-to-end savings), generalizability (across languages), and maintainability (implementation effort).

In this paper, we have conceptually and experimentally analyzed (with 16 open-source projects in four different languages) that implementing a CC on the level of the *intermediate representation (IR)* is beneficial for all three properties in comparison to the previous approaches, which work either on source-code level (Ccache) or AST level (cHash).

IRHash, our implementation of IR-level caching for LLVM, provides, with 19% end-to-end savings, not only better effectiveness than Ccache (10%) and cHash (16%) across the analyzed C projects but also generalizability over other languages supported by LLVM (we have shown this for C++, Fortran, and Haskell) out of the box. Due to the structural simplicity of the IR language, the implementation of caching at this level can, furthermore, be kept simple and maintainable. In the case of IRHash, we needed less than 600 lines of code for the complete LLVM plugin. Overall, our results show that

[3] LLVM: `llvm/lib/Transforms/Utils/IRNormalizer.cpp`

the IR level is the most beneficial level for implementing a CC.

> *Please refer to the published artifacts to verify and replicate the experiments [46]. The IRHash source code is also available on GitHub [47].*

Acknowledgments

We would like to thank our reviewers and our shepherd Eric Eide for their constructive feedback. This work has been supported by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – 236869097.

References

[1] J. Rosdahl. "Ccache — Compiler cache." (2010), [Online]. Available: https://ccache.dev.

[2] Mozilla. "sccache - Shared Compilation Cache." (2016), [Online]. Available: https://github.com/mozilla/sccache.

[3] C. Dietrich, V. Rothberg, L. Füracker, A. Ziegler, and D. Lohmann, "cHash: Detection of Redundant Compilations via AST Hashing," in *Proceedings of the 2017 USENIX Annual Technical Conference (USENIX '17)*, (Santa Clara, CA, USA), Santa Clara, CA, USA: USENIX Association, Jul. 2017, ISBN: 9781931971386. [Online]. Available: https://www.usenix.org/conference/atc17/technical-sessions/presentation/dietrich.

[4] S. I. Feldman, "Make — a program for maintaining computer programs," *Software: Practice and experience*, vol. 9, no. 4, pp. 255–265, 1979.

[5] Google. "Bazel." (2015), [Online]. Available: https://bazel.build.

[6] Meta. "Buck2 build system website." (2023), [Online]. Available: https://buck2.build.

[7] H. Esfahani, J. Fietz, Q. Ke, *et al.*, "Cloudbuild: Microsoft's distributed and caching build service," in *Proceedings of the 38th International Conference on Software Engineering Companion (ICSE)*, 2016, ISBN: 978-1-4503-4205-6. DOI: 10.1145/2889160.2889222.

[8] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 13, no. 4, pp. 451–490, Oct. 1991. DOI: 10.1145/115372.115320.

[9] T. Johnson, M. Amini, and X. David Li, "ThinLTO: Scalable and incremental LTO," in *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2017, pp. 111–121. DOI: 10.1109/CGO.2017.7863733.

[10] The Flang Team. "Welcome to Flang's documentation – The Flang Compiler." (2025), [Online]. Available: https://flang.llvm.org.

[11] B. Gamari. "The Glasgow Haskell Compiler." (2024), [Online]. Available: https://www.haskell.org/ghc/.

[12] Linux Kernel Organization, Inc. "The Linux Kernel Archives." (2025), [Online]. Available: https://kernel.org.

[13] OpenSSL. "OpenSSL." (2024), [Online]. Available: https://www.openssl.org.

[14] SQLite. "SQLite Home Page." (2024), [Online]. Available: https://www.sqlite.org.

[15] Lua. "The Programming Language Lua." (2024), [Online]. Available: https://www.lua.org.

[16] The PostgreSQL Global Development Group. "PostgreSQL: The world's most advanced open source database." (2025), [Online]. Available: https://www.postgresql.org.

[17] Python Software Foundation. "The Python programming language." (2025), [Online]. Available: https://github.com/python/cpython.

[18] Free Software Foundation, Inc. "GNU Bash." (2020), [Online]. Available: https://www.gnu.org/software/bash/.

[19] Linaro Limited. "Mbed TLS." (2024), [Online]. Available: https://www.trustedfirmware.org/projects/mbed-tls/.

[20] SDL Community. "Simple DirectMedia Layer." (2025), [Online]. Available: https://www.libsdl.org.

[21] musl libc. "musl libc." (2024), [Online]. Available: https://musl.libc.org.

[22] The Bochs Project. "bochs: The Open Source IA-32 Emulation Project." (2024), [Online]. Available: https://bochs.sourceforge.io.

[23] C. M. Kohlhoff. "Asio C++ Library." (2024), [Online]. Available: https://think-async.com/Asio/.

[24] S. Martins. "Clazy." (2025), [Online]. Available: https://invent.kde.org/sdk/clazy.

[25] KDAB Group. "GammaRay™." (2024), [Online]. Available: https://www.kdab.com/development-resources/qt-tools/gammaray/.

[26] LAPACK team. "LAPACK — Linear Algebra PACKage." (2024), [Online]. Available: https://netlib.org/lapack/.

[27] V. Holen. "ShellCheck – shell script analysis tool." (2024), [Online]. Available: https://www.shellcheck.net.

[28] B. Gamari. "deterministic builds." (2019), [Online]. Available: https://gitlab.haskell.org/ghc/ghc/-/wikis/deterministic-builds.

[29] GuillaumeGen. "recompilation avoidance." (2010), [Online]. Available: https://gitlab.haskell.org/ghc/ghc/-/wikis/commentary/compiler/recompilation-avoidance.

[30] B. Gamari. "Object code produced by GHC is non-deterministic." (2016), [Online]. Available: https://gitlab.haskell.org/ghc/ghc/-/issues/12935.

[31] R. Mesquita. "determinism: Deterministic object code with -fllvm." (2024), [Online]. Available: https://gitlab.haskell.org/ghc/ghc/-/issues/25274.

[32] M. Shahin, M. Ali Babar, and L. Zhu, "Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices," *IEEE Access*, vol. 5, pp. 3909–3943, 2017, ISSN: 2169-3536. DOI: 10.1109/ACCESS.2017.2685629. [Online]. Available: http://ieeexplore.ieee.org/document/7884954/.

[33] M. Azanza, L. Montalvillo, and O. Díaz, "20 years of industrial experience at SPLC: a systematic mapping study," in *Proceedings of the 25th ACM International Systems and Software Product Line Conference*, 2021, pp. 172–183.

[34] C. Lattner, M. Amini, U. Bondhugula, *et al.*, "MLIR: Scaling compiler infrastructure for domain specific computation," in *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, IEEE, 2021, pp. 2–14.

[35] P. W. Sathyanathan, W. He, and T. H. Tzen, "Incremental whole program optimization and compilation," in *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2017, pp. 221–232. DOI: 10.1109/CGO.2017.7863742.

[36] D. L. Parnas, "On the criteria to be used in decomposing systems into modules," *Communications of the ACM*, vol. 15, no. 12, pp. 1053–1058, Dec. 1972, ISSN: 0001-0782. DOI: 10.1145/361598.361623.

[37] Y. Yu, H. Dayani-Fard, and J. Mylopoulos, "Removing false code dependencies to speedup software build processes," in *Proceedings of the 2003 Conference of the Centre for Advanced Studies on Collaborative Research*, ser. CASCON '03, Toronto, Ontario, Canada: IBM Press, 2003, 343–352.

[38] Y. Wang, W. Chang, T. Deng, Y. Zou, and B. Xie, "Decomposing God Header File via Multi-View Graph Clustering," in *2024 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2024, pp. 112–124. DOI: 10.1109/ICSME58944.2024.00021.

[39] F. Fulin. "FASTBuild - High-Performance Build System." (2012), [Online]. Available: https://fastbuild.org.

[40] W. F. Tichy, "Smart recompilation," *ACM Trans. Program. Lang. Syst.*, vol. 8, no. 3, 273–291, Jun. 1986, ISSN: 0164-0925. DOI: 10.1145/5956.5959.

[41] R. W. Schwanke and G. E. Kaiser, "Smarter recompilation," *ACM Trans. Program. Lang. Syst.*, vol. 10, no. 4, 627–632, Oct. 1988, ISSN: 0164-0925. DOI: 10.1145/48022.214505.

[42] Z. Shao and A. W. Appel, "Smartest recompilation," in *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '93, Charleston, South Carolina, USA: Association for Computing Machinery, 1993, 439–450, ISBN: 0897915607. DOI: 10.1145/158511.158702.

[43] M. Pool. "distcc: a fast, free distributed C/C++ compiler." (2002), [Online]. Available: https://www.distcc.org.

[44] SUSE. "icecream - Distributed compiler with a central scheduler to share build load." (2003), [Online]. Available: https://github.com/icecc/icecream.

[45] T. Landsberg, C. Dietrich, and D. Lohmann, "TASTING: Reuse Test-case Execution by Global AST Hashing," in *Proceedings of the 17th International Conference on Software Technologies - ICSOFT*, INSTICC, Lisbon, Portugal: SciTePress, Jul. 2022, pp. 33–45, ISBN: 978-989-758-588-3. DOI: 10.5220/0011139200003266.

[46] T. Landsberg, J. Grunenberg, C. Dietrich, and D. Lohmann, *IRHash: Efficient Multi-Language Compiler Caching by IR-Level Hashing - Artifacts*, version v1.0.1, May 2025. DOI: 10.5281/zenodo.15367568. [Online]. Available: https://doi.org/10.5281/zenodo.15367568.

[47] T. Landsberg, J. Grunenberg, C. Dietrich, and D. Lohmann, *IRHash: Efficient Multi-Language Compiler Caching by IR-Level Hashing - Source Code*, May 2025. [Online]. Available: https://github.com/luhsra/IRHash.

A Artifact Appendix

Abstract

IRHash is a multi-language compilation cache that operates at the IR level. The artifact contains all the source code for IRHash as well as its evaluation and the scripts to reproduce the results of the paper. This comprises all measured values in the paper, including validation and performance measurement. For this purpose, 100 commits are built for each of the 16 projects, both incrementally and as clean builds. This process runs fully automatically. As the procedure can take a very long time (over a week), there are also instructions to reduce the extent of the experiments. To ensure reproducibility, there are also OCI images provided that can be easily executed.

Scope

The artifact is intended to reproduce all findings of the paper by automatically executing the complete evaluation. Specifically, it concerns the results in Fig. 1, Tab. 3, Tab. 4, and Tab. 5, thereby validating the implementation of IRHash, analyzing its performance characteristics compared to Ccache and cHash, and executing the end-to-end-performance benchmark.

Contents

The artifact includes instructions, the source code for IRHash, its evaluation and scripts to reproduce the results of the paper. OCI container images are also included so that the experiments can be executed directly. Alternatively, Dockerfiles are provided to build the images.

Hosting

The artifact together with detailed instructions is available under the DOI 10.5281/zenodo.15367568 on Zenodo.

Requirements

Only the ability to run OCI image container is necessary, e.g., Podman or Docker, since all steps run in containers for reproducibility.

A.1 Installation

There is no installation required. A detailed guide on how to obtain and run the containers can be found together with the artifact.

On-Demand Container Partitioning for Distributed ML

Giovanni Bartolomeo*
Technical University of Munich

Navidreza Asadi*
Technical University of Munich

Wolfgang Kellerer
Technical University of Munich

Jörg Ott
Technical University of Munich

Nitinder Mohan
TU Delft

Abstract

As machine learning (ML) models grow in complexity and scale, distributed deployment across multiple devices has become essential for ensuring performance and scalability. However, the dynamic nature of distributed ML, where models must be frequently retrained, partitioned, and updated, exposes severe limitations in the current de-facto container-based model deployment. Specifically, the layered architecture of container filesystems is not well-suited for handling fine-grained model updates and partitioned ML deployments, leading to inefficient rebuilds and long delays. In this paper, we present 2DFS, a novel two-dimensional filesystem that enables independent updates, caching, and distribution of ML model components. We design and develop a complete ecosystem, including a builder, registry, and cache hierarchy, to streamline the build and deployment processes of ML models leveraging 2DFS. Our comprehensive evaluation of 14 real-world ML models demonstrates that 2DFS achieves up to 56x faster build times, 25x better caching efficiency, while providing on-demand image partitioning with negligible overhead. 2DFS is fully OCI-compliant and integrates seamlessly with existing infrastructures and container workflows.

1 Introduction

In recent years, machine learning (ML) has emerged as a dominant paradigm in modern applications, from real-time video analytics [40, 42, 80, 82, 92] to intelligent transportation/operations [14, 51], and has radically transformed the field of computing [5, 48, 64, 65]. The increasing complexity and size of the models powering these applications has created a demand for high-performance warehouse computing infrastructures offering significant resources [18, 67, 104]. As the applications become more nuanced, latency has emerged as a driving operational factor, which has led to the development of distributed ML frameworks [1, 15, 20, 74], ML model optimizations [22, 34, 51, 89, 92], and split computing [3, 32, 39–41, 43, 50, 54–56, 63, 66, 95], also enabling executions on relatively constrained devices outside of traditional data center environments [7, 51, 103]. ML models can be partitioned and distributed independently to fit real-time hardware availability. This also guarantees improved resource utilization in shared infrastructures as well as better scalability to fulfill increasing user requests.

Despite all the advances, *the practical deployment and management of these "optimized" models across distributed infrastructures remains a challenge* [9, 19, 23, 66, 68, 81]. Popular Machine Learning Operations (MLOps) frameworks, such as MLflow [71], TFX [94], and KubeFlow [52], offer robust support for the training, versioning, and lifecycle management of models. However, when it comes to actual deployment, these frameworks typically rely on container-based virtualization (more specifically, Docker) to package models and their dependencies. That is because containers encapsulate both runtime environments and application dependencies in a lightweight and portable manner [14, 21, 40, 40, 51, 88, 94, 108]. We argue that while containers, backed by the Open Container Initiative (OCI) [45], provide portability and isolation, they limit the real-world applicability of many distributed ML optimizations and model partitioning techniques. In fact, when model weights are embedded in the container images, model updates caused by continuous re-training require a new image build every time [14, 21, 51]. This limitation also affects solutions based on live model partitioning. These approaches optimize inference by splitting and parallelizing the model based on real-time infrastructure changes. Unfortunately, new containers must be built for each generated partition and each partition update [40, 108]. Solutions managing model parameters using on-demand download via MLOps frameworks are extremely difficult to cache and optimize in edge environments. Instead, approaches using container volumes require considerable manual effort to provision the necessary external storage. These solutions are expensive to realize in edge infrastructures, where hardware availability is scarce, networking is limited, and infrastructure ownership is heterogeneous.

In this work, we address these limitations by introducing `2DFS`, a two-dimensional container filesystem tailored to dis-

* Both authors contributed equally to this research.

tributed ML model packaging and deployment. The proposed solution exploits the well-consolidated container registries and runtimes technologies for caching and distribution of the image layers, the ML model parameters, and further any large file required by the container. This prevents developers from reinventing the wheel with no need to manually provision and manage their own storage solutions or use third party tooling. `2DFS` introduces a new `2dfs.field` container image layer type that decouples model weights and partitions from the traditional container layered filesystem. By organizing data into a two-dimensional matrix of independent "allotments", `2DFS` enables model components to be built, cached, and updated independently. Furthermore, it also enables on-demand partitioning of the images, enabling flexible split-computing deployments. Specifically, our contributions are as follows.

I. We propose `2DFS`, a novel extension to the OCI container format. It adopts a two-dimensional filesystem design where model components, encapsulated as `2dfs.field` layer, can be processed, cached, and updated independently. This eliminates the inefficiencies of traditional hierarchical layering, where even minor updates would trigger expensive rebuilds across the entire container.

II. We design and develop the complete `2DFS` ecosystem that includes several core components: a builder utility for construction of `OCI+2DFS` images, a partitioning operator for on-demand image partitions, and an OCI-compliant registry supporting the new layer type and functionalities.

III. We conduct a comprehensive evaluation of `2DFS` using 14 real-world ML models. The results demonstrate significant improvements in build times and caching efficiency compared to traditional OCI images built with Docker. Specifically, `2DFS` achieves up to **56× faster** build times and a **25× faster** image updates; and in some cases, such as with large EfficientNet-V2L model which allows 82 splits, 2DFS builds were up to **120× faster**.

Note that while `2DFS` is well-suited to distributed ML, it is not limited to such but has broader applicability in distributed computing field. For example, it can be used to efficiently package binaries in container images §5. The `2DFS` project is open-source and available on GitHub [10].

2 Background and Motivation

Containers have emerged as a de facto standard for packaging applications and dependencies and have seen widespread adoption in the industry. Furthermore, with the increasing sophistication of container management platforms, thanks to Open Container Initiative (OCI) standardization [46], the creation, execution, and sharing of container images has become more streamlined. Even in the context of ML applications, containers have become the preferred method for packaging and deploying ML models [13, 101, 102], as they provide a consistent runtime environment and simplify management, especially when dealing with diverse hardware and software configurations at the edge [24, 77, 88]. However, the current containerization methods have significant overhead for supporting distributed/split ML model deployment – particularly for model partitioning and continuous model updating tasks. To grasp these limitations, we first provide a brief overview of containers under the hood.

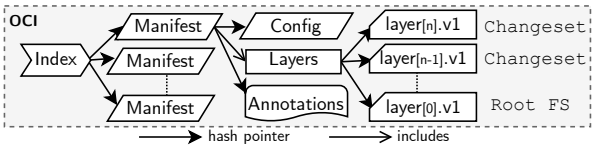


Figure 1: Cross-section of OCI image format.

2.1 Container Dissection and Lifecycle

Figure 1 shows the cross-section of an OCI image, which contains all necessary files to run a container. Internally, the image is structured as a Directed Acyclic Graph (DAG) composed of an `index`, a set of `manifests`, `config` files, and `layers`. A `manifest` represents an image tailored to a specific platform and contains references to the filesystem `layers` and a `config` file. A layer describes a filesystem change; starting from the `RootFS`, each layer updates, adds, or removes files from the previous layer. Layers are stacked on top of each other and joined together to form the container filesystem (e.g., OverlayFS [30]) A layer is stored as a compressed tarball called *blob* and referenced via a hash of their contents (the `digest`). When multiple containers are created from images that use identical layers, the layers are shared, reducing the storage overhead. Every change in a layer invalidates its hash, causing a cascade effect that propagates up to the `index`. The `config` file holds the container's configuration details, such as execution parameters and environment variables, along with the root filesystem changes expressed as a list of `DiffIDs`. A `DiffID` is the hash of an uncompressed image layer, not to be confused with the `digest` mentioned above.

A container image build process is handled by a builder component, which is composed of a front end and a back end. The front end parses the build file descriptor and converts it into a set of instructions that can be used by the backend. Let us take the example of Docker's builder component, `buildx`, which uses BuildKit [72] as the default backend. The `dockerfile` frontend takes Docker's build descriptor file (called Dockerfile [27]) and translates it to high-level `LLB` definitions. The BuildKit backend then creates the image's build graph from the definitions, and the `solver` component builds the container by executing the graph nodes one after another. At this stage, container layers are created as a result of file operations in the definition, such as copying files from the host to the container. Note that despite BuildKit being known for its optimized builder backends that can parallelize image builds, the process of layer generation is still sequential due to the nature of the filesystem operations.

2.2 The Rise of Distributed ML

Deploying ML applications can be challenging due to the high computational resources required by their underlying ML models [77, 82]. This challenge is more pronounced in edge computing environments, where machines are of heterogeneous hardware configurations and multiple applications share the resources [58, 61, 69, 92]. As a result, large ML models with several layers and parameters are often found unsuitable for edge infrastructures [68, 80, 91].

One approach focuses on designing compact models that are smaller and more efficient than their larger counterparts. Such models are developed using a combination of techniques such as using fewer network blocks/layers, blocks with fewer parameters, reducing the precision of model weights through quantization, removing unnecessary connections through pruning, and transferring knowledge from large models to smaller ones through distillation [14, 60, 66, 87, 89]. However, compact models often struggle to adapt to dynamic data distributions as effectively as larger models, making them susceptible to data drift and requiring regular updates [14, 22, 51, 81]. As such, these models require *frequent periodic retraining and updating* (as often as 30-50 seconds [51, 92]) to ensure they remain accurate and adaptable to edge environments.

Another approach is to split the large ML model into multiple parts and distribute them across multiple devices for parallel execution [39, 40, 50, 54, 66]. More specifically, the model is partitioned into several slices, each of which is deployed on a separate machine, forming an execution pipeline. An "early-exit" head can optionally be added to some of the splits [98, 100], providing the possibility of not processing all the splits until the end, trading accuracy with better response time. Table 2 shows the splitting potential of popularly used ML models, where some models can be split into as many as 82 layers. Recent work has explored its applicability in edge-cloud computing, where privacy or performance concerns dictate that the first few splits be processed at the edge, and intermediate information is sent to nearby and distant cloud infrastructure for further processing [54, 79, 95, 96]. However, model partitioning introduces new challenges, particularly in dynamic edge environments where the number of available devices can change over time due to failures, poor connections, or resource sharing with multiple applications [7, 61, 99]. Furthermore, since the hardware configurations of these devices are heterogeneous, the subset of model partitions that can be deployed on a device can vary [40, 103] – requiring flexible methods for distribution and deployment.

2.3 Containers for Distributed ML?

Almost all popular ML management frameworks, such as MLflow [71, 102], Flower [13], and SkyPilot [101], rely on containers for model deployment due to their agility and ability to replicate the runtime environment across different hardware configurations. The typical packaging process involves frameworks creating a container image that includes the ML model and its dependencies, which is then pushed to a central storage (registry) for distribution [13, 17, 44, 49, 51, 70, 71, 73, 76]. Each device periodically connects to the registry and fetches the updated images specific to their requirements.

# Images	Build Time (s)
1	81.67
20	113.78
41	157.64
62	149.92
82	242.66

Table 1: Docker build time with increasing splits of EfficientNet (`ENv2L`) across different images.

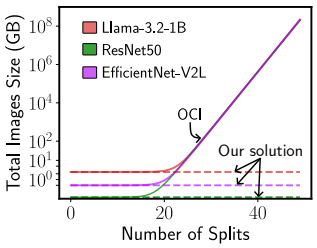


Figure 2: OCI image size explodes with increasing model splits, unlike our solution.

However, model optimization and partitioning techniques noted earlier do not function well for containerized models. Let's consider how split ML models can be distributed using containers. The obvious approach is to create a separate container image for each model partition (e.g., a model with 60 splits can be stored as 60 images). However, bigger devices might be able to compute larger chunks of the model or might provide better hardware acceleration for specific layers, therefore requiring a special image configuration. An ML cluster may also experience resource losses, and the splits must be re-computed to reduce the model fragmentation. The number of possible image configurations can grow exponentially with the number of splits. Additionally, computing these images for AI models can be an extremely time-consuming operation.

To demonstrate this, we conducted an experiment with the EfficientNetv2L (`ENv2L`) model, where we built an increasing number of Docker images while increasing the number of splits of a model. For example, 2 splits would result in 2 images, 3 splits would result in 3 images, and so on. Table 1 shows a significant increase in Docker image build time with increasing model splits – leading up to a 3× longer build compared to the 1 split use case. To combat this, one option could be pre-building all possible container images, ensuring that any requested subset of partitions is readily available for devices to pull from the registry. However, this approach becomes prohibitively expensive in terms of build time as well as the required storage space at the registry, as illustrated in Figure 2. Assuming we can map each model layer to a container image layer $l_i \in L$. Then, a model split is a container image where for each i, layer l_i is either included in the image or not. Each combination is an image manifest referencing the included layers, leading to 2^n combinations, where n is the number of splits. Thus, the combination of manifests required, as shown in the figure, explodes, leading to significant space overhead. We leave it to the reader to imagine the build time necessary to compute such a set of images. Moreover, if the build time is not enough of a concern, these models are also

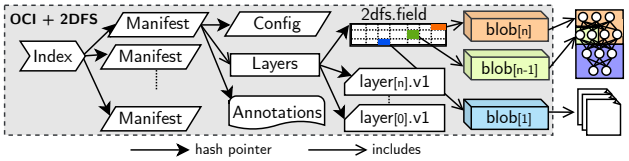


Figure 3: Cross-section of a 2DFS image.

frequently re-trained and updated – thus, the image build procedure must be repeated.

Alternative state-of-the-art solutions distribute model parameters by (a) independently storing and then mounting them to the containers as volumes or (b) downloading them on the fly. The former approach prevents data sharing across untrusted parties and requires considerable engineering efforts in edge environments where object stores are not widely available. The latter approach is transparent to the container runtimes, preventing efficient parameters caching and sharing across workloads but instead forcing the applications to download the parameters at startup every time. Local model registry caches can be used to mitigate bandwidth usage at the cost of additional resource usage.

2.4 Why 2DFS?

We believe that the limitations of containerized ML model distribution lie in the layered filesystem structure of container images, which is well-conceived for code and dependencies but not intended for large files, libraries, and drivers, resulting in severely bloated images [105]. This structure prevents the retrieval of on-demand data subsets and forces continuous container image rebuild for fine-grained file updates, which is expensive and time-consuming. For example, driver and library updates usually invalidate the cache of entire container images even if the underlying code did not change. On the other hand, by addressing content by digest, container registries offer an excellent mechanism for object retrieval. With this work, we aim to extend the OCI image format to recycle the well-known and commonly used container image registries to distribute not only code and dependencies but also binaries, model parameters, and any large file without affecting the complexity of the CI/CD pipelines. By building a specialized container layer offering independent file pointers, the ML model splits, drivers and dependencies can be independently cached in image registries and downloaded on-demand by container runtimes. Registries can then provide each client with image partitions, including only the necessary files, to avoid downloading bloated images.

3 2DFS System Design

2DFS is a two-dimensional filesystem build and distribution framework for containers. The first building block is the extension of the conventional layered filesystem used in container architectures, which often restricts optimization due to the linked nature of layer changesets. 2DFS introduces an additional layer type called `2dfs.field`. Figure 3 shows a cross-section of an OCI container image with the new `2dfs.field` layer type (OCI+2DFS). A field is a sparse hash-pointer matrix of *allotments* representing a self-contained, non-overlapping, and independent filesystem space. Each allotment links one or more files or, for instance, a split of a neural network. This design relaxes intra-layer dependencies, enabling more efficient filesystem distribution (see §3.1) and fast image builds with extremely high parallelism (see §3.2). The second important building block of the proposed system is the image partitioning operation, supported by the 2DFS compliant registry (see §3.5). This operation allows field partitions that can be retrieved independently. The sub-fields are serialized as regular OCI layers and are compliant with any existing container runtime. The two-dimensional design is well suited for flexibly partitioning ML models, allowing for more efficient and on-demand "best-fitting" deployments (see §3.4).

Figure 4 shows the end-to-end workflow of 2DFS container image specification, build, and distribution. At step ①, the developer specifies a base image and the allotments content (e.g., different model splits) with their position in the field via `2dfs.json` descriptor file. For instance, model splits `s1` and `s2` are assigned to the allotments `<0.0>` and `<0.1>`, respectively. 2DFS builder (§3.2) extends a regular OCI image with a `2dfs.field` (§3.1) layer and is responsible for (i) creating the filesystem, (ii) filling the field with allotments, (iii) attaching field to the image and (iv) caching the allotments. The extended image format, referred to as OCI+2DFS, is stored locally in the builder cache (§3.3) and can be pushed to a 2DFS-compliant registry (§3.5) in step ②. Developers can fetch partitioned images from the registry on-demand (step ③) by specifying the semantic tag in the pull request (§3.4). Note that the registry image partitioner only performs manifest operations on the pre-built allotments, restricting the partitioning overhead as much as possible. Upon receiving the image partition pull request (step ④), the registry serializes the image partition into a fully OCI-compliant image and serves it to the runtime *which can be used by any OCI-compliant runtime without modifications*.

3.1 2dfs.field layer

We extend the OCI Image Specification [46] to introduce `2dfs.field` layer type as a new media type. As previously discussed, `2dfs.field` is represented as a sparse matrix that organizes allotments in a grid structure. Among all the possible geometrical structures, this work represents a field as a 2-dimensional plane to simplify the definition of partitions and to easily map the structure of neural networks. In such a plane, allotments can be easily defined to contain model splits or can be graphically organized based on different use cases. Check §5 and Appendix A for further details. Inside the field, each allotment contains: (i) an index in the form of `<row.column>`, (ii) a `digest`, which serves as a hash-pointer reference to an OCI artifact, and (iii) the `DiffID`, represent-

`application/vnd.oci.image.layer.v1.2dfs.field`

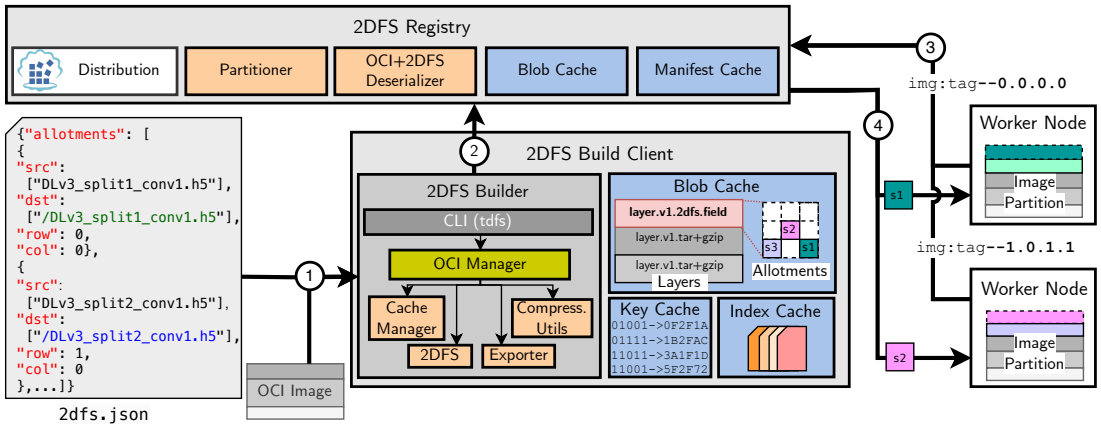


Figure 4: 2DFS end-to-end workflow and component architecture.

```
"allotments":[
    {
      "row": int,
      "col": int,
      "digest": str,
      "diffid": str,
    },
  ]
  "tot_rows": int
  "owner": str
```

Listing 1: Description of the 2dfs.field manifest

ing the hash of the uncompressed content as defined by the OCI specification. Listing 1 illustrates the manifest description of 2dfs.field inside an image. OCI artifacts referenced by allotments are standard .tar.gz blobs that contain user-defined compressed content and are indistinguishable from regular OCI layer objects. The 2DFS builder (see §3.2) makes sure these blobs satisfy the following three properties: (i) they are commutative and can be applied in any order to a container filesystem without altering the final state, (ii) they can be cross-referenced across multiple fields, supporting efficient data sharing and caching across images, and (iii) they can be serialized as conventional OCI layers, ensuring seamless integration with existing OCI runtimes. Changeset operations in 2DFS closely resemble those in OCI layers. When a blob referenced by an allotment is updated, a cascade of hash invalidations occurs, requiring reconstruction of the field data structure. However, unlike intermediate layers in conventional container systems, where linked changesets force rebuilding all dependent layers, only allotments referencing modified files are updated, while the rest are served from the cache (see §3.3). For instance, if model split s1 in Figure 4 is updated, s2 allotment remains unaffected and will not be rebuilt in the new image. The next section details the 2DFS builder.

3.2 2DFS Builder

Starting from a regular OCI image and a 2dfs.json file descriptor, the builder composes the 2-dimensional field to create an extended OCI+2DFS image. The 2DFS builder consists of several components: (a) CLI utility, (b) image manager, (c) 2DFS filesystem serializer/deserializer, (d) cache manager, (e) exporter, and (f) compression utility. *CLI Utility* provides a range of commands to build, push, and manage both OCI and OCI+2DFS images locally. The core of the utility is the following build command:

```
tdfs build [base-image] [target-image] [flags]
```

Here, the base-image is a fully-compliant OCI image descriptor in the format registry/repository/image:tag. In case the registry or the tag is omitted, the tool defaults to Docker registry and latest, respectively. The command expects a descriptor file called 2dfs.json. The descriptor file contains a list of allotments defined by the user. For each allotment, the user describes the source files list and the destination path in the target image, as well as the row and column of the allotment in the field (see fig. 4).

The *OCI Manager* component is responsible for the lifecycle of an image within the builder – from its initial state as base-image to its final form as extended OCI+2DFS image. The build command utilizes the *OCI Manager* to retrieve the base-image, either from the local cache or from a remote registry, and then extend it with the 2dfs.field layer.

The *Cache Manager* handles the storage of image indexes, manifests, and blobs in the local caches, detailed in §3.3.

The 2DFS component generates the field data structure and its allotments based on the 2dfs.json descriptor. Its descriptor is parsed, and the referenced files are copied into temporary directories, which are structured according to the path hierarchy defined by the user. Each temporary directory is first hashed to generate a DiffID—the digest of the uncompressed layer, as per the OCI specifications.

The *Compression Utilities* compress the directories generated by the filesystem component into a .tar.gz file (referred to as a blob), which is cached and hashed. The blob hash, along with the DiffID, column, and row information, is used by the 2DFS serializer to create the allotments for the 2dfs.field layer. After the 2dfs.field layer is serialized, and the referenced blobs are successfully cached, the layer is appended to

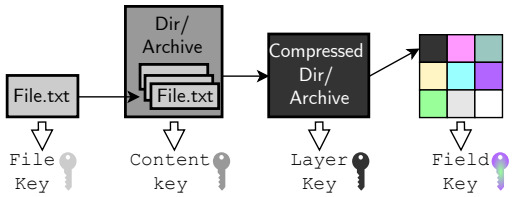


Figure 5: Caching layers architecture.

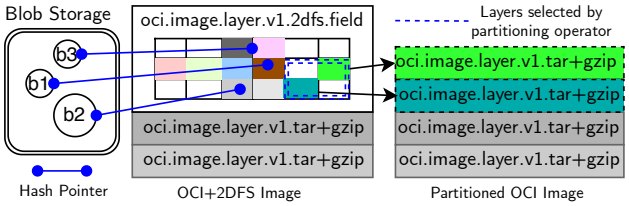


Figure 6: Example showcasing image partitioning.

each manifest referenced by the index. Since allotment generation is a platform-independent operation, the builder appends the new field structure to all manifest variants, regardless of OS or architecture – ensuring cross-platform compatibility. The corresponding `config` for each image is also updated to include all the `DiffIDs` for the allotments as if they are layered. This step is necessary to ensure compliance with the OCI runtime specification. If certain layers are removed from the image, e.g., due to image partitioning, the `config` is updated accordingly to reflect the modified image structure. The builder dynamically re-computes new config files using the `DiffIDs` embedded in the field's allotments. Modifying the manifest's layers and config file invalidates its hash-pointer reference at the index level, so the builder generates a new index file for the `OCI+2DFS` image to maintain integrity.

The *OCI Manager Exporters* use the updated index files to retrieve cache entries and package the new image. `2DFS` supports two exporters: tarball and registry. The tarball exporter can package either a full extended `OCI+2DFS` image or a partitioned image in plain `OCI` format. The registry exporter, on the other hand, pushes all image layers, fields, allotments, and blobs to a remote registry as OCI artifacts, utilizing standard REST API calls defined in the OCI distribution spec [45] to ensure compatibility with existing registries.

One key advantage of the `2DFS` build process is that it removes the need to mount the files into a running container to build and extract layers. This is possible because the `2dfs.field` layer only references static files used at runtime. As a result, the build process is significantly faster and scalable, particularly for large files, while minimizing disk I/O. Additionally, due to the independence of each file (allotment), the blobs can be built in parallel, further accelerating the build process.

3.3 Cache Hierarchy

The builder employs a multi-layer cache system to efficiently store files, allotments, and fields. Every time the *Cache Manager* modifies or creates an object, a hash of its bytes is generated, known as a cache *key*. This *key* uniquely identifies the object, allowing the builder to skip upcoming build stages if the *key* already exists. At each stage, the builder works with dedicated caching layers, generating or interacting with objects from previous stages.

Definition 1 *In a caching hierarchy, ranging from C_0 to C_j, the following rules apply when building a generic object o_i^k for each caching layer i with $0 < i < j$:*

R1. *o_i^k and key K_i^k are built from at least one object of C_{i-1}.*

R2. $$\begin{cases} C_i \leftarrow C_i \cup \{K_i^k : o_i^k\}; i \leftarrow i+1 & \text{if } K_i^k \notin C_i \\ i \leftarrow j & \text{otw} \end{cases}$$

For allotment generation, the build stages and corresponding cache operate as shown in Figure 5. At any level i in the hierarchy, a change has a cascade effect on the layers above. For instance, if a file within an allotment is updated, the following steps occur: (i) the file's *key* is invalidated, the builder copies the new file in a temporary directory used for the allotment build and the *key* of the uncompressed folder is generated (ii) the updated directory is compressed into a blob, generating another *key*; and (iii) the field is re-constructed. If a *key* matches an existing one during this process, the builder skips to the last cache layer and proceeds with the final stages of the build. Since allotments can be built independently, the `2dfs.field` is constructed from separate parallel build processes. Changes in the file structure trigger reconstruction of only affected allotments, resulting in significant build performance improvements (see §4).

The *Cache Manager* module is used by the *OCI Manager* component of `2DFS` builder. It employs *three* different cache storages for different object categories. The *Key Cache* stores intermediate keys for the file copying, allotment creation, and compression, which only serve as metadata for cache hierarchy. Note that the cache only stores the *keys* and hash-pointers referencing the blobs, which are instead stored in the *Blob Cache*. This cache contains every object in the builder except for the image indexes. Each object in this cache is named based on the `sha256sum` of its content, ensuring efficient content validation and easy retrieval using hash pointers. The media type for each blob in this cache is defined by its hash-pointer reference. This means that when a blob is retrieved from the cache, its media type is already known for deserialization, avoiding the need to duplicate this information elsewhere. Finally, the *Index Cache* stores the OCI index of all the images that are referenced via the hashed fully-qualified OCI name: `sha256sum(registry/repository/image:tag)`. Extended `OCI+2DFS` images are annotated with the name of the base image and a `"2dfs"` label indicating that the `2dfs.field` layer is present in the image DAG hierarchy.

3.4 Image Partitioning

The design of `2DFS` focuses on flexibility and ease of use. In a two-dimensional space, users are free to arrange ML models, files, libraries, and other resources to fit any area of

the field (Appendix A discusses this further). This design makes it easy for the user to select a subset of the field with only the desired functionalities. We call this process image partitioning, and it must allow retrieving only the necessary parts of an image on-demand, without needing to re-build or re-push any additional manifest. In 2DFS, partitions from the two-dimensional field can be easily extracted and serialized as regular OCI layers.

Definition 2 *Given a field F composed of allotments A where $F = A^{m \times n}$, a partition P is the union of subfields $P = \bigcup_i f_i$ where $f_i = A[f_i^{x_1} : f_i^{x_2}, f_i^{y_1} : f_i^{y_2}]; i, x, y \in \mathbf{N}; x \leq m, y \leq n$.*

Given a partition P, a 2DFS exporter can create an OCI bundle where the allotments composing the union of the subfields f_i are flattened into regular OCI layers. Figure 6 illustrates an example of the operation. The center shows the selected partition P of the 2DFS+OCI image composed of subfield $A[1:2, 4:5]$ spanning from <row 1.column 4> to <row 2.column 5>. This subfield only has two allotments (shown as different green shades). The image partitioning operation (i) fetches the corresponding allotment's blob from the local blob store, (ii) serializes the blob as regular OCI layers in the image manifest, and (iii) updates the manifest config file with the layer's DiffIDs. The result is the base image from the 2DFS build phase, without the special 2dfs.field layer, but with additional flattened allotments selected by the partitioning operator. To enable seamless image partitioning without additional registry endpoints, we define the partitioning operator as a semantic image tag, i.e., a string containing a special suffix defining the image operation.

```
registry/repository/image:tag--x1.y1.x2.y2
```

x1 and y1 are the coordinates of the top-left corner, and x2 and y2 the bottom-right corner of sub-field f_i (e.g., tag for fig. 6 partition is tag−−1.4.2.5). While only a 2DFS compliant registry can correctly parse the semantic tag, any runtime can use semantic tags to retrieve a partitioned image. The partitioning operators can be concatenated, allowing for an unlimited number of sub-fields for each partition. The sub-fields are parsed by the exporter or a compliant registry and transformed into the actual partition. The partition is processed to generate the requested image.

3.5 2DFS Registry

2DFS registry is an OCI compliant registry with support for .2dfs.field media type and partitioning operator. Starting from the open-source OCI implementation, we introduced the following updates:

1. *Partitioner*: Serializes the partition of an OCI+2DFS into an OCI manifest.
2. *Partitioning Operator Support*: Enables parsing of smantic tags for partitioning at the image/pull endpoint.
3. OCI+2DFS *Deserializer*: Provides support the 2dfs.field media type.

The *partitioner* efficiently generates OCI-compliant manifests and configs by fetching the OCI+2DFS image, decoding the 2dfs.field layer, and applying the selected allotments as OCI layers. It then removes the 2dfs.field layer, replacing it with standard layer descriptors, and it applies the layers DiffIDs to the rootfs of the image's config file. Finally, the new manifest is cached. When a semantic tag with a partition is detected at the /v2/<name>/manifests/<reference> endpoint, the tag parser triggers the *partitioner*, which fetches the manifest and generates the new partitioned image based on the requested subfields. The OCI+2DFS *Deserializer* extends the registry to handle 2dfs.field media types, leveraging distribution.UnmarshalFunc [25] type to deserialize and access the manifest content.

4 Evaluation

We evaluate the performance of 2DFS and compare it against Docker – the most popular container-building tool [28]. Since 2DFS is a completely new approach to layering, in most scenarios, multiple Docker images were used to simulate the same functionality of an OCI+2DFS image. For example, for *three* partitions of 2DFS image, we require *three* separate Docker images. In this evaluation, we focus on the continuous model updating and split computing use cases, where a ML model is stored as several partitions or splits. We compare the performance of 2DFS while varying the amount of splits of a model.

4.1 Experimental Setup

Build Infrastructure. All the build experiments are carried out with Docker runtime version 27.3.1 with containerd snapshotter and all default settings, such as buildx builder and BuildKit backend. The base image used for all the builds is the tensorflow/tensorflow image version 2.17.0. The image build is performed on a bare-metal server with 2× AMD EPYC 7302 16-Core Processors, 256GB DDR4 RAM, and 2TB 12Gb/s Solid State Drive. For the build, we compare the 2DFS builder (in §3.2) against the default Docker build command. The 2DFS compliant registry is hosted on a server running 1× Intel Core i9-9820X CPU, 128GB DDR4 RAM, 1TB NVMe 3.5Gb/s Solid State Drive. The machines are interconnected with a 1Gb/s Ethernet connection.

Edge Infrastructure. We created a distributed edge computing infrastructure consisting of 10 Raspberry Pi 4B (RPi) devices, each equipped with four ARM Cortex-A72 cores and 8GB of RAM. Additionally, we utilized a Linux PC with an Intel Core i7-4790 4-Core Processor and 16GB RAM as the gateway for our experiment in §4.5. All machines were interconnected via a 1Gb/s Ethernet switch. The gateway served as the controller for dynamically deploying different model splits on varying numbers of RPis, generating load, and aggregating output to calculate end-to-end response time. Notably, the gateway did not execute any model splits itself. The RPis rely on the remote server for cross-device clock synchronization.

Model	Label	Application	#Parameter	Total Size (MB)	#Splits	Avg #Params per Split $\pm\sigma$
ResNet50	RN50	Recognition	25.6 M	97.8	18	1.39 $\pm$ 1.76 M
MobileNetV2	MNv2	Recognition	3.5 M	13.5	19	0.13 $\pm$ 0.15 M
MobileNetV2$_{\alpha=1.4}$	MNv2L	Recognition	6.2 M	23.5	19	0.24 $\pm$ 0.29 M
DeepLabv3+*	DLv3	Segmentation	39.1 M	150	19	2.17 $\pm$ 3.67 M
EfficientNet-V2-B0	ENv2B0	Recognition	7.2 M	27.5	24	0.26 $\pm$ 0.22 M
EfficientNet-V2-B1	ENv2B1	Recognition	8.2 M	31.3	30	0.24 $\pm$ 0.22 M
EfficientNet-V2-B2	ENv2B2	Recognition	10.2 M	38.8	31	0.29 $\pm$ 0.26 M
EfficientNet-V2-B3	ENv2B3	Recognition	14.5 M	55.2	35	0.38 $\pm$ 0.33 M
ResNet101	RN101	Recognition	44.7 M	170.5	35	1.25 $\pm$ 1.25 M
YOLOv3	YOLOv3	Detection	62 M	236.3	36	1.68 $\pm$ 2.29 M
EfficientNet-V2S	ENv2S	Recognition	21.6 M	82.5	43	0.48 $\pm$ 0.39 M
ResNet152	RN152	Recognition	60.4 M	230.5	52	1.14 $\pm$ 1.06 M
EfficientNet-V2M	ENv2M	Recognition	54.4 M	207	60	0.90 $\pm$ 0.99 M
EfficientNet-V2L	ENv2L	Recognition	119 M	454	82	1.45 $\pm$ 1.60 M

Table 2: Overview of our Model Zoo which includes popularly used models for different application categories.

To ensure accurate timing measurements, we deployed a network time protocol (NTP) server on the gateway. This setup results in a maximum clock offset of approximately $\pm 100\mu s$ and a jitter less than $\pm 100\mu s$, allowing for precise device-to-device response time reporting. We perform all experiments 10 times and report the median across runs.

Model Zoo. Table 2 presents the model zoo utilized in our evaluation, comprising 14 models across three diverse application scenarios, including Object Recognition, Detection, and Segmentation. These models offer varying splitting options based on their architectural characteristics (cf. §4.1.1). The number of splits per model ranges from 18 to 82. The results are obviously expected to generalize to much larger models.

4.1.1 Generating Model Splits

To accommodate a wide range of models and offer a general solution, we employ the Keras API due to its ability to leverage multiple backends, such as PyTorch, TensorFlow, or JAX, catering to different hardware needs. We automatically generate the `2dfs.json` descriptor as a direct output of the model split generation. For all the models in the zoo, a descriptor shows the position of splits' weights and architecture in the two-dimensional structure. Our approach draws inspiration from previous work [40, 54, 92] on model slicing and is guided by three key principles:

1. **Minimizing output size.** We avoid slicing at branch or shortcut points, as this would require passing multiple tensors to the next device. Instead, we identify inherent information bottlenecks, typically located after aggregation layers (e.g., Add or Concatenation), which serve as suitable split points.
2. **Maximizing flexibility.** By introducing as many splits as possible while adhering to the first principle, we increase deployment flexibility and enable more efficient utilization of available resources.
3. **Balancing split inference times.** While not as critical as the first two principles, we strive to achieve a fair distribution of inference time across each split, ensuring that no single device is disproportionately burdened. Our end-to-end results on `MNv2L` (fig. 14) supports this goal.

Guided by these principles, our slicing strategy targets bottleneck layers whenever possible or, alternatively, those with relatively smaller output sizes compared to their neighbors as candidate split points. Each split was designed to contain two or three computationally expensive operations, such as various forms of convolution or fully connected layers. In certain applications, like object detection, the output of multiple splits may need to be transmitted and utilized in a later stage. We restructured these model graphs to transmit the necessary information directly to the specific split(s) that require it, thereby avoiding unnecessary data passing through all splits. Furthermore, whenever possible, we identified opportunities for parallel computation without tensor sharing and created separate splits to leverage these possibilities. For instance, a few blocks in `YOLOv3` have such a characteristic. By exploiting parallelism in the model graph, we can accelerate inference and improve scalability.

Our model slicing approach yields a varying number of splits across the 14 models in our Model Zoo (Table 2), ranging from 18 (e.g., `RN50`) to 82 (e.g., `ENv2L`). Each split can be deployed and containerized as a standalone application, enabling flexible deployment scenarios. We include in fig. 18 the parameter count and size distribution of different splits, showing the variability among models. Note that each split comprises multiple model layers. Each layer potentially has weights and biases that may change values after model updates or have distinct versions (e.g., `FP32`, quantized `INT8`) to accommodate diverse use cases. To comply with this dynamic nature, we detangle the split model architecture (computation graph) from the weights themselves to improve flexibility. We then apply `BZip2` compression to the splits and store them as separate allotments in the `OCI+2DFS` format.

4.2 Image Build

We begin by measuring the build performance of 2DFS and Docker builder. For each model in the zoo, we define a split capacity ranging from 0% to 100%. We define split capacity as the percentage of the splits we divide the model compared to the maximum allowed splits. For example, at 50% split capacity for ENv2L, we only generate 82×0.5=41 splits and divide the model into 41 portions instead of 82. With 0% split capacity, we do not split the model at all resulting in *one* large "split" representing the entire model. Each split is allocated in a separate 2DFS allotment or OCI layer. Therefore, an image containing YOLOv3 at 100% split capacity features 37 layers (RootFS + 36 splits), while an image with 100% split of ENv2L has 83 layers. We perform the experiments for all the models in the zoo, but only report results for a subset for brevity. We refer interested readers to Appendix B for the remaining results.

Figure 7 shows the results of build-time of 1 image with respect to split capacity %. On average, 2DFS builds OCI+2DFS images 16× faster than Docker with standard OCI images. Even in the base case of 0% split capacity, 2DFS is already outperforming Docker. For deep models with larger layer sizes, such as ENv2L, we observe a counterintuitive pattern in the Docker build. The build is slower when either the layers are too big (e.g., 0% split capacity) or the layers are too many (e.g., 100% split capacity), with a sweet spot at 50% split capacity. For smaller models, instead, Docker build time decreases somewhat linearly with the number of layers. This behavior hints that there is a tradeoff between layer size and layer number. Generally, it looks faster to build smaller layers until they are confined in numbers (E.g., less than 50 layers.). With 2DFS, we observe that the more we split, the faster the build time. The performance gap between the two build systems is related to how Docker handles the layer creation. Docker uses a build container to which all the files are copied during the build. Mounting the model weights into the build container and extracting a layer changeset is an expensive and time-consuming operation. Moreover, each layer operation is executed using the input mount point from the previous phase sequentially since Docker cannot know if the layer's applications are commutative. On the other hand, 2DFS exploits the allotment independence to parallelize the build process, thus escalating the cache hierarchy described in §3.3 for all the allotments simultaneously. The 2DFS builder runs under the assumption that a `2dfs.field` layer is immutable and only contains static files. Therefore, a build container is not necessary, and the allotments are computed locally, minimizing disk I/O utilization also for the 0% split capacity scenario.

With OCI+2DFS image format, we can retrieve on-demand any possible image split with a random partitioning. In fig. 8, we compare the build time needed to pre-compute multiple OCI images for each split. Because the total number of split and partition combinations would be unfeasible to build, we simplify the experiment by only building a small subset of possible image splits for Docker. In particular, we compute one Docker image for each split based on the Split Capacity. E.g., ENv2L50% split capacity in the context of this plot means that we partition the model in 41 splits, and we generate 41 different Docker images, one for each split, while only one OCI+2DFS image with 41 equal partitions from the original 82 allotments. Note that an OCI+2DFS image can actually generate up to 2^{82} different partitions with such split capacity, but it would be unreasonable to build so many Docker images as demonstrated in fig. 2. The results show an average of 56× faster build time for 2DFS, with a peak of 120× faster builds for the ENv2L model with 100% split capacity. The build time of OCI+2DFS in this experiment is constant and does not depend on the split capacity because we always build one image including all the allotments, while to achieve the same result with traditional OCI images, we need to build a new image for each split, increasing the build time manyfold.

Figure 9 shows the CPU and Memory usage of the builder machine measured during the experiment shown in fig. 7. The results show that while 2DFS and Docker have similar memory usage, 2DFS has higher CPU consumption, especially with higher image split capacity. With 0% split capacity, we observe similar consumption. The more we increase the split capacity, the higher the values of CPU usage. While 2DFS builder computes each allotment's DiffID and blob simultaneously, which are both CPU-intensive operations, Docker performs these operations in sequence, resulting in lower CPU usage but longer builds. It is worth noting that even if, in this experiment, Docker shows up to 20% lower CPU consumption over the build time, 2DFS build time was on average 16*x* faster, resulting in an overall lower total CPU time.

4.3 Image Download and Partitioning

We now evaluate the overhead caused by the partitioning operation at the registry compared to pull of traditional OCI images. Figure 10 compares the download time of four different partitions for the ENv2L and RN50 images, each containing respectively 25, 50, 75, and 100% of the total splits. For Docker, we pre-compute the splits, and we push all four images to the registry independently. For 2DFS, we just push to the registry a single OCI+2DFS image and ask for an on-demand partition every time. The pull requests are performed on a separate worker machine using the Docker CLI. We observe how the on-demand partitioning is performed by the registry with minimal overhead, leading to a comparable total download time with on average only 20 *ms* additional latency.

Figure 11 shows the bandwidth usage at the client side during the download of the pre-computed OCI images built using Docker versus the OCI+2DFS partitioned image. The result shows comparable bandwidth usage across the two approaches. Therefore, serialized allotments bring negligible overhead in terms of download time and data rate.

Figure 12 shows the CPU usage at the registry side during the download of traditional OCI images built using Docker and

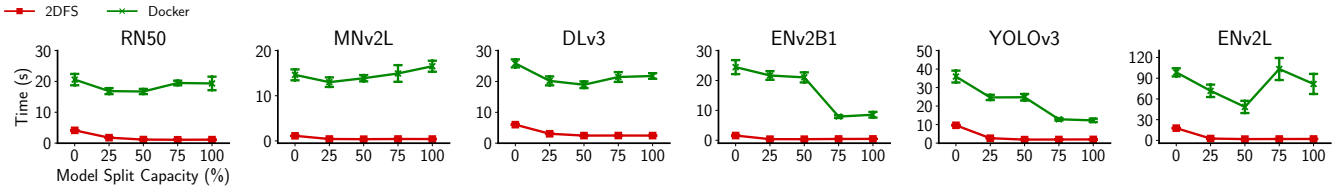


Figure 7: Build time for a single image with increasing number of splits per layer.

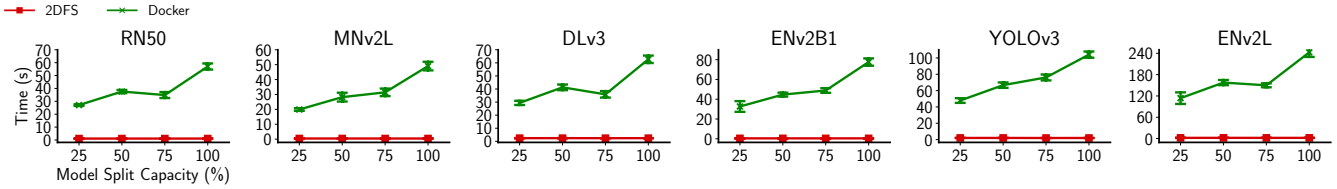


Figure 8: Build time for different split partitions where each partition is packaged as a separate image.

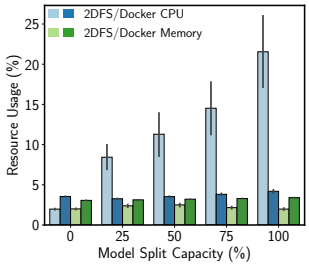


Figure 9: Resources consumption during image build.

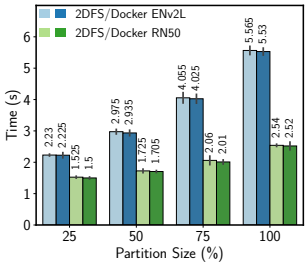


Figure 10: Download of partitioned vs prebuilt images.

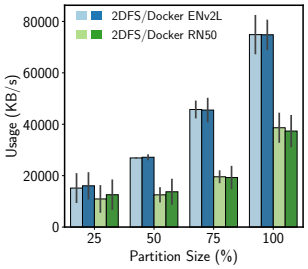


Figure 11: Download bandwidth for partitioned images.

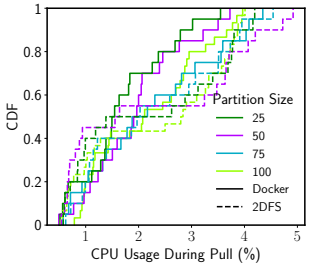
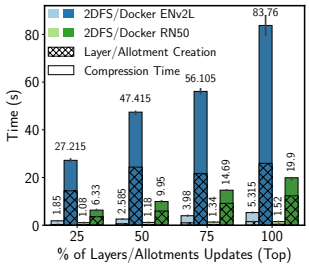
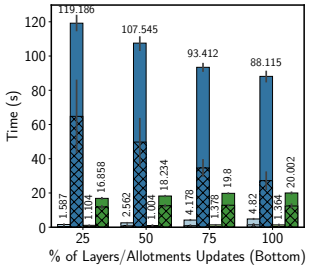


Figure 12: Registry resource utilization during image pull.



(a) Top-down update strategy.



(b) Bottom-Up Update Strategy.

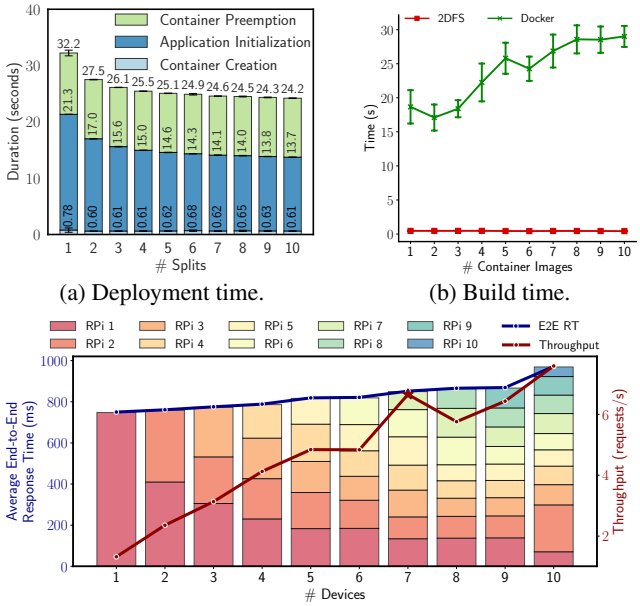
Figure 13: Build time after model updates with image caching.

2DFS images. With these measurements, we try to quantify the computational overhead of the on-demand partitioning operation compared to the download of pre-computed images. The results show that the registry CPU usage is comparable across the two approaches, with a slight increase for the 2DFS images of 1% after the 50-th percentile. This gap represents the CPU overhead of the on-demand partitioning operation and the serialization of the field's allotments into OCI layers. We also measure the memory overhead at the registry, but we omit the results as we do not observe any significant difference between the two approaches.

4.4 Model Updates

As mentioned in §2.2, some parts (i.e., splits) of a model, particularly in smaller models, may receive frequent updates to comply with the accuracy demands as they experience distribution drift. In this section, we evaluate the effect of model updates on the image rebuild. For each image in the model zoo, we build a baseline version where each split is placed on a different layer. Then, we update increasingly 25%, 50%, 75%, and 100% of the model splits with newer versions and re-perform the build of the images. This operation commonly happens after model re-training. This experiment measures the caching efficiency of each builder. Better caching strategies lead to lower build times. In 2DFS, the position of the allotment that needs to be updated does not affect the build time due to the independence property. Each allotment that has not been updated will skip the cache hierarchy as described in §3.3. On the other hand, in Docker, the position of the layer that needs to be replaced affects the cache invalidation and drastically increases the build time. Therefore, we compare two different configurations, one where the updated layers are placed at the top of the image and one where they replace the bottom layers of the image.

Figure 13 shows the results of the experiment. On the left side, we have the results obtained by starting the updates from the top layers, while on the right, the results while updating the layers from the bottom. As expected, Docker build time is relatively more efficient when updating the layers from the top, as Docker is able to cache the bottom layers successfully. However, the build time is, on average, 25× slower than 2DFS. The worst case scenario for Docker is when the layers are updated from the bottom, resulting in a 75× slower build time compared to 2DFS. Interestingly, in this latter scenario,



(a) Deployment time. (b) Build time.

(c) Response time & throughput.

Figure 14: Performance of distributed `MNv2L` over increasing number of devices and model splits.

the more layers we update, the faster the build gets. This is related to the way Docker invalidates the cache for the layers above and re-computes one layer after the other. We observe that for the bottom-up update scenario, the compression time increases with the number of updated layers while the layering time decreases. The layering time includes all the operations prior to the layer compression, such as cache invalidation, content copy, and changeset computation. We conclude that in the current Docker implementation, it is faster to rebuild the image from scratch than to perform a bottom-up update. When updating the image layers from the top, the compression and layering time grows linearly as expected. The `2DFS` build time is growing linearly with the number of updated layers but at a slower pace, showing a better caching efficiency.

4.5 Split Computing Performance

Another scenario of interest besides continuous model updates is the split computing phenomenon §2.2, where due to different reasons such as privacy and/or scalability, a model is split into multiple parts, and each device serves only a subset of the model. This inherently provides an opportunity for pipeline parallelism as we enable it by default. In this section, we evaluate the performance of the `MNv2L` model in a distributed Edge Computing testbed. We deploy the model on a set of Raspberry Pi devices, ranging from 1 to 10, each running a different split of the model. Every time a device joins the experiment, the image splits are re-computed uniformly to fit the new number of devices. For each new device, a new image split must be computed for traditional `OCI` images, while for `OCI+2DFS` images, we can simply request the new partition from the registry. We measure (i) E2E RT, which refers to the average end-to-end response time, (ii) throughput of the deployed system, which is the number of requests per second that it is able to fulfill, (iii) deployment time and (iv) build time of the images for the different splits.

Figure 14c shows the inference performance results. `RPi x` refers to the average time a particular Raspberry Pi device takes to process its split, including the communication and queuing delay. Observe how the end-to-end response time increases linearly with the number of devices. The model fragmentation and the additional networking overhead are the main contributing factors. On the other hand, the total throughput of the system increases steeply from less than 1.3 to 7.6 requests/s. Model fragmentation allows constrained devices to compute their splits, avoiding CPU throttling and memory exhaustion issues. Moreover, model splitting can improve the overall system scalability.

Figure 14a is showing the overall deployment time breakdown for the `MNv2L` model. This experiment does not include the download and partitioning time as it assumes that Docker images were pre-computed. The results show a logarithmic decrease in build time, mainly caused by the application initialization and the container creation. Smaller model sizes are faster to decompress, especially on constrained hardware, resulting in an overall speed-up of the deployment process.

Figure 14b shows the time to build the new image split for each device joining the experiment. Build time for `2DFS` is constant as only one image is generated at the beginning, and the new splits are requested on-demand. On the other hand, the build time for the Docker use case increases based on the number of images that need to be generated every time we need a new split. On some occasions, such as when generating images for 2 and 6 devices, the build time results are lower than expected, probably related to a discrepancy in the split sizes, which can lead to faster build times. The relative build time difference between the two approaches shows that `2DFS` build is 55.9× faster. We omit the total build time because it is not comparable across the two approaches; in fact, for this simple example, we had to build 55 OCI images but only a single image for `2DFS`.

5 Discussion and Future Directions

While our primary focus is on providing an efficient and standard packaging framework for the mainstream distributed ML, there are additional application scenarios for `2DFS` that are available straight out of the box. Conducting separate evaluation experiments of these applications is unlikely to provide new insights beyond those already presented in the previous section; thus, we discuss them here and refer to relevant evaluations when applicable.

Other ML Architectures. `2DFS` is not limited to the models provided in the Model Zoo. Indeed, it is a general solution, and one can use any model or set of models together with `2DFS` to streamline data propagation and parallel executions in a standard containerized way. This includes autoencoders, recurrent neural networks, and generative AI such as diffusion models

and large language models (LLMs). For instance, LLMs heavily rely on distributed inference infrastructure across several machines [4, 35, 83, 84, 107], a practice essential for scalability and for supporting massive models like Llama 3.1 with up to 405 billion parameters [38]. Integration of `2DFS` and LLM libraries like vLLM [53], DeepSpeed [4, 86, 93] and the Kubernetes-native `llm-d` [62] comes with foreseeable advantages such as facilitating LLM distributed serving pipelines by leveraging (i) `OCI+2DFS` images for efficient packaging of inference code and model parameters, and (ii) on-demand partitioning for distributing model shards. This combines LLM serving with the simplicity and scalability of self-contained microservice applications.

Additional ML Usecases. In the context of ML applications, several scenarios emerge where `2DFS` is advantageous. For instance, in ML serving systems, dynamic application demands (e.g., varying latency and accuracy requirements), different backends, and numerous model sizes lead to thousands of *variants* of the same base model and corresponding Docker images [88]. This calls for a more efficient packaging framework that can decouple these different models; this is where `2DFS` excels. In fact, each variant can be treated as a separate set of allotments, and significant gains can be achieved as demonstrated in §4.4.

Along similar lines, the concept of bloated vs. debloated ML containers [105] highlights the issue of oversized Docker containers with potential vulnerabilities due to the inclusion of various packages and toolchains during different stages (design, training, serving) of an ML application's lifecycle. `2DFS` offers a straightforward solution by facilitating the separation of these stages into distinct partitions. Appendix A illustrates ML packaging in more detail.

Further, another line of work [7, 85, 97] aims to reduce the communication overhead in tensor parallelism by constructing sparsely distributed sub-models. These works along with multi-model pipelines where an application is constructed of multiple ML tasks present another compelling use case for `2DFS`. Instead of relying on several docker images, or worse, a bulky image with redundant blobs, it packages everything into a single image. Each device only pulls the needed parts, resulting in reduced build, download, and deployment times §4.3.

Frequent ML Model Updates. One of the most critical aspects of containerized ML is the container build time, hindering the propagation of ML model updates. Often, platforms prefer to rely on object stores [36], just to avoid frequent container re-builds, which are slower. However, this practice requires customized release pipelines and has severe limitations in edge environments, as discussed in §2. Khani et al. [51] show how real-world continuous re-training for object-detection models such as MobileNetV2(`MNv2`) and ResNet50(`RN50`) can lead to model updates arriving every 30 seconds or even faster. For `RN50`, a relatively small model, in fig. 13, we show how traditional container updates on production-grade hardware can span between 6 and 20 seconds, depending on where the layer update happens, slowing the effectiveness of continuous re-training manyfold. On the other hand, `2DFS`, is able to build such images in 1 to 1.5 seconds, up to 13*x* faster. For larger models such as EfficientNetV2L(`ENv2L`), the gap increases further up to 75*x* faster rebuild. These results highlight how `2DFS` opens an unprecedented opportunity to utilize containers for model packaging and caching even in highly dynamic environments with frequent model re-train.

Broader Applications. By design, `2DFS` is a general-purpose solution applicable to diverse scenarios that require flexible and efficient packaging. An interesting use case is the management of fat images [2], where containers are built with extensive binaries, libraries and other application requirements, e.g. analytics utilities to full-blown networking libraries like ServiceRouter [90]. In traditional systems, any update to a single binary requires rebuilding the entire container, which is not only a resource-intensive process but can also take several hours to complete. With `2DFS`, separate allotments can contain different binaries and libraries, allowing teams to update specific components independently, while improving deployment efficiency. This capability significantly accelerates the DevOps workflows, enabling continuous integration and deployment practices to operate at a much finer granularity.

Future Directions. In addition to the above-mentioned scenarios, `2DFS` has potential operational benefits in microservice packaging/deployment in edge computing environments [12] and for federated learning workloads [13]. However, `2DFS` also has several limitations to be considered. `2DFS` is currently limited to supporting only OCI images, which, despite being an established standard, restricts its compatibility with other formats such as the conventional `vnd.docker`. While the number of `2DFS` allotments is potentially unlimited, leaving developers total freedom in how to architect the field, traditional OCI runtimes can only handle a maximum of 127 container layers. Therefore, to maintain compatibility with runtimes like Docker, the `2dfs.json` descriptor must be designed in a way that limits `2DFS` partitions to a maximum of 127 allotments per partition. This still allows fine-grained parameter caching but can be a limitation for developers dealing with extremely large models. In the future, we plan to address these limitations and extend the system by (i) exploring alternative field form factors and layouts to optimize both performance and usability, (ii) developing a `2DFS` storage driver to overcome the layer limitations imposed by current file systems, and (iii) designing a system-specific runtime that dynamically retrieves model partitions based on file system access patterns. For instance, a future direction can aim at extending container runtimes to better support use cases requiring more than 127 layers and optimize how allotments are handled within the container runtime storage driver [29].

6 Related Work

Distributed ML serving needs model partitioning. Related work on distributed ML serving can be categorized into two main avenues. One focuses on splitting models between edge devices and cloud servers to reduce data transmission latency and improve privacy [3, 32, 39–41, 43, 50, 54–56, 63, 66, 95]. Techniques vary in their partitioning strategies, often driven by specific objectives such as energy efficiency [50, 59], bandwidth, latency, or privacy [47, 79]. Some works employ early-exit mechanisms for generating predictions on edge devices [63, 92, 95]. Others propose creating multiple versions of partitions to adapt to changing network conditions [22, 43] or utilize idle machines over the Internet for large language models [16]. The second line of research targets deploying small models on distributed edge computing devices, often considering scenarios like self-driving cars [14, 51] or multiple audio and video analytics applications [22, 92]. These works address the capacity gap between smaller and larger models through model adaptation. Some proposals involve deploying models or splits on various devices and aggregating results to achieve desired accuracy [6, 7, 97, 100, 103], while others focus on efficient model update strategies, such as training only the final layers of models [51], freezing 70-90% of layers during retraining [14, 22], or dynamically selecting a portion of model partitions to process and applying early-exit to maintain Service-Level Objectives (SLO) requirements [92].

ML Serving meets containers. Containers have become ubiquitous as they offer isolated runtime environments, ensuring consistent behavior across different dependencies and platforms [14, 21, 40, 51, 88, 108]. However, most MLOps frameworks overlook the unique challenges of Distributed ML [8, 9, 11, 13, 19, 31, 33, 37, 57, 71, 75, 78, 106]. These frameworks assume that packaging is a solved problem, neglecting the complexities of dividing ML models into multiple parts for efficient adaptation and distribution across devices.

Our work highlights the need for a more comprehensive and efficient approach to packaging in Distributed ML. While there are numerous proposals addressing various aspects of Distributed ML, we found no existing packaging tool that natively supports its requirements. This gap underscores the importance of our solution.

7 Conclusion

In this paper, we introduced `2DFS`, a two-dimensional filesystem build and distribution framework designed to address the inefficiencies of existing container-based systems for distributed ML. Our design enables independent updates of model partitions by decoupling them from the traditional layered structure, offering substantial improvements in both build and deployment efficiency. Through our comprehensive evaluation, we demonstrated that `2DFS` provides significant performance gains – achieving on average *56× faster build times* and *25× better caching efficiency* compared to Docker and regular OCI images.

Acknowledgments

We would like to thank the anonymous ATC reviewers and the shepherd for their helpful comments and insights during the review process of this paper. We would also like to thank the ATC Artifact Evaluation Committee for their meticulous examination and efforts to reproduce our results. This work is supported by the Federal Ministry of Education and Research of Germany (BMBF) project 6G-Life (16KISK002) and by the National Growth Fund through the Dutch 6G flagship project “Future Network Services".

References

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*, 2016.

[2] Dimitri Aivaliotis. Tupperware: Container deployment at scale. 2015.

[3] Mario Almeida, Stefanos Laskaridis, Stylianos I Venieris, Ilias Leontiadis, and Nicholas D Lane. Dyno: Dynamic onloading of deep neural networks from cloud to device. *ACM Transactions on Embedded Computing Systems*, 21(6):1–24, 2022.

[4] Reza Yazdani Aminabadi, Samyam Rajbhandari, Ammar Ahmad Awan, Cheng Li, Du Li, Elton Zheng, Olatunji Ruwase, Shaden Smith, Minjia Zhang, Jeff Rasley, et al. Deepspeed-inference: enabling efficient inference of transformer models at unprecedented scale. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15. IEEE, 2022.

[5] Ganesh Ananthanarayanan, Victor Bahl, Landon Cox, Alex Crown, Shadi Nogbahi, and Yuanchao Shu. Video analytics-killer app for edge computing. In *Proceedings of the 17th annual international conference on mobile systems, applications, and services*, pages 695–696, 2019.

[6] Navidreza Asadi and Maziar Goudarzi. An ensemble mobile-cloud computing method for affordable and accurate glucometer readout. *arXiv preprint arXiv:2301.01758*, 2023.

[7] Navidreza Asadi and Maziar Goudarzi. Variant parallelism: Lightweight deep convolutional models for distributed inference on iot devices. *IEEE Internet of Things Journal*, 11(1):345–352, 2024.

[8] AWS. Deep Learning Container. https://github.com/aws/deep-learning-containers. [Online; accessed 26-Jan-2024].

[9] Mariam Barry, Albert Bifet, and Jean-Luc Billy. Streamai: Dealing with challenges of continual learning systems for serving ai in production. In *2023 IEEE/ACM 45th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 134–137. IEEE, 2023.

[10] Giovanni Bartolomeo, Navidreza Asadi, Wolfgang Kellerer, Jörg Ott, and Nitinder Mohan. 2dfs. https://github.com/2DFS, 2025. Accessed: 22 May 2025.

[11] Giovanni Bartolomeo, Jacky Cao, Xiang Su, and Nitinder Mohan. Characterizing distributed mobile augmented reality applications at the edge. In *Companion of the 19th International Conference on emerging Networking EXperiments and Technologies*, pages 9–18, 2023.

[12] Giovanni Bartolomeo, Mehdi Yosofie, Simon Bäurle, Oliver Haluszczynski, Nitinder Mohan, and Jörg Ott. Oakestra: A lightweight hierarchical orchestration framework for edge computing. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 215–231, Boston, MA, July 2023. USENIX Association.

[13] Daniel J Beutel, Taner Topal, Akhil Mathur, Xinchi Qiu, Javier Fernandez-Marques, Yan Gao, Lorenzo Sani, Kwing Hei Li, Titouan Parcollet, Pedro Porto Buarque de Gusmão, et al. Flower: A friendly federated learning research framework. *arXiv preprint arXiv:2007.14390*, 2020.

[14] Romil Bhardwaj, Zhengxu Xia, Ganesh Ananthanarayanan, Junchen Jiang, Yuanchao Shu, Nikolaos Karianakis, Kevin Hsieh, Paramvir Bahl, and Ion Stoica. Ekya: Continuous learning of video analytics models on edge compute servers. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 119–135, 2022.

[15] Alexander Borzunov, Dmitry Baranchuk, Tim Dettmers, Max Ryabinin, Younes Belkada, Artem Chumachenko, Pavel Samygin, and Colin Raffel. Petals: Collaborative inference and fine-tuning of large models. *arXiv preprint arXiv:2209.01188*, 2022.

[16] Alexander Borzunov, Max Ryabinin, Artem Chumachenko, Dmitry Baranchuk, Tim Dettmers, Younes Belkada, Pavel Samygin, and Colin A Raffel. Distributed inference and fine-tuning of large language models over the internet. *Advances in Neural Information Processing Systems*, 36, 2024.

[17] Ryan Chard, Zhuozhao Li, Kyle Chard, Logan Ward, Yadu Babuji, Anna Woodard, Steven Tuecke, Ben Blaiszik, Michael J Franklin, and Ian Foster. Dlhub: Model and data serving for science. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 283–292. IEEE, 2019.

[18] Sarah Chasins, Alvin Cheung, Natacha Crooks, Ali Ghodsi, Ken Goldberg, Joseph E Gonzalez, Joseph M Hellerstein, Michael I Jordan, Anthony D Joseph, Michael W Mahoney, et al. The sky above the clouds. *arXiv preprint arXiv:2205.07147*, 2022.

[19] Zhenpeng Chen, Yanbin Cao, Yuanqiang Liu, Haoyu Wang, Tao Xie, and Xuanzhe Liu. A comprehensive study on challenges in deploying deep learning based software. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2020, page 750–762, New York, NY, USA, 2020. Association for Computing Machinery.

[20] Will Constable. Pytorch distributed overview. https://pytorch.org/tutorials/beginner/dist_overview.html, 2024. Accessed: 2024-10-23.

[21] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J Franklin, Joseph E Gonzalez, and Ion Stoica. Clipper: A {Low-Latency} online prediction serving system. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 613–627, 2017.

[22] Harshit Daga, Yiwen Chen, Aastha Agrawal, and Ada Gavrilovska. Clue: Systems support for knowledge transfer in collaborative learning with neural nets. *IEEE Transactions on Cloud Computing*, 2023.

[23] Yingnong Dang, Qingwei Lin, and Peng Huang. Aiops: real-world challenges and research innovations. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 4–5. IEEE, 2019.

[24] Jim Davis, Philbert Shih, and Alex Marcham. State of the edge: A market and ecosystem report for edge computing. https://www.stateoftheedge.com, 2023.

[25] distribution. distribution/distribution repository. https://github.com/distribution/distribution, 2024. [Online; accessed 15-Oct-2024].

[26] Docker. Dockefile best practices. https://docs.docker.com/build/building/best-practices/, 2024. Accessed: 22 October 2024.

[27] Docker. Dockefile reference. https://docs.docker.com/reference/dockerfile/, 2024. Accessed: 21 October 2024.

[28] Docker. Most-used developer tool. https://www.docker.com/blog/docker-stack-overflow-survey-thank-you-2023/, 2024. Accessed: 22 October 2024.

[29] Docker. Storage driver. https://docs.docker.com/engine/storage/drivers/, 2024. Accessed: 22 May 2025.

[30] Docker Docs. Use the overlayfs storage driver. https://docs.docker.com/storage/storagedriver/overlayfs-driver/, 2020.

[31] Domino. Role of Containers. https://domino.ai/blog/role-of-containers-on-mlops-and-model-production. [Online; accessed 26-Jan-2024].

[32] Xin Dong, Barbara De Salvo, Meng Li, Chiao Liu, Zhongnan Qu, Hsiang-Tsung Kung, and Ziyun Li. Splitnets: Designing neural architectures for efficient distributed computing on head-mounted systems. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 12559–12569, 2022.

[33] DVC. Model Registry. https://dvc.ai/model-registry. [Online; accessed 26-Jan-2024].

[34] Ali Farhadi and Joseph Redmon. Yolov3: An incremental improvement. In *Computer vision and pattern recognition*, volume 1804, pages 1–6. Springer Berlin/Heidelberg, Germany, 2018.

[35] Yao Fu, Leyang Xue, Yeqi Huang, Andrei-Octavian Brabete, Dmitrii Ustiugov, Yuvraj Patel, and Luo Mai. {ServerlessLLM}:{Low-Latency} serverless inference for large language models. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 135–153, 2024.

[36] Saeid Ghafouri, Kamran Razavi, Mehran Salmani, Alireza Sanaee, Tania Lorido Botran, Lin Wang, Joseph Doyle, and Pooyan Jamshidi. Ipa: Inference pipeline adaptation to achieve high accuracy and cost-efficiency. In *Companion of the 16th ACM/SPEC International Conference on Performance Engineering*, pages 2–3, 2025.

[37] Noah Gift and Alfredo Deza. *Practical MLOps*. " O'Reilly Media, Inc.", 2021.

[38] Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

[39] Yitian Hao, Wenqing Wu, Ziyi Zhang, Yuyang Huang, Chen Wang, Jun Duan, and Junchen Jiang. Deft: Slo-driven preemptive scheduling for containerized dnn serving. In *Symposium on Networked Systems Design and Implementation*, 2023.

[40] Ke-Jou Hsu, Ketan Bhardwaj, and Ada Gavrilovska. Couper: Dnn model slicing for visual analytics containers at the edge. In *Proceedings of the 4th ACM/IEEE Symposium on Edge Computing*, pages 179–194, 2019.

[41] Chuang Hu, Wei Bao, Dan Wang, and Fengming Liu. Dynamic adaptive dnn surgery for inference acceleration on the edge. In *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*, pages 1423–1431. IEEE, 2019.

[42] Yitao Hu, Rajrup Ghosh, and Ramesh Govindan. Scrooge: A cost-effective deep learning inference system. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 624–638, 2021.

[43] Jin Huang, Colin Samplawski, Deepak Ganesan, Benjamin Marlin, and Heesung Kwon. Clio: Enabling automatic compilation of deep learning pipelines across iot and cloud. In *Proceedings of the 26th Annual International Conference on Mobile Computing and Networking*, pages 1–12, 2020.

[44] Hugging Face. The model hub. https://huggingface.co/docs/hub/en/models-the-hub, 2024. Accessed: October 23, 2024.

[45] Open Containers Initiative. Oci specification: Distribution specification. https://github.com/opencontainers/distribution-spec/blob/main/spec.md, 2024. Accessed: 20 October 2024.

[46] Open Containers Initiative. Oci specification: Image specification. https://github.com/opencontainers/image-spec/blob/main/spec.md, 2024. Accessed: 20 October 2024.

[47] Hyuk-Jin Jeong, InChang Jeong, Hyeon-Jae Lee, and Soo-Mook Moon. Computation offloading for machine learning web apps in the edge server environment. In *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, pages 1492–1499. IEEE, 2018.

[48] Junchen Jiang, Ganesh Ananthanarayanan, Peter Bodik, Siddhartha Sen, and Ion Stoica. Chameleon: scalable adaptation of video analytics. In *Proceedings of the 2018 conference of the ACM special interest group on data communication*, pages 253–266, 2018.

[49] Wenxin Jiang, Nicholas Synovic, Matt Hyatt, Taylor R Schorlemmer, Rohan Sethi, Yung-Hsiang Lu, George K Thiruvathukal, and James C Davis. An empirical study of pre-trained model reuse in the hugging face deep learning model registry. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 2463–2475. IEEE, 2023.

[50] Yiping Kang, Johann Hauswald, Cao Gao, Austin Rovinski, Trevor Mudge, Jason Mars, and Lingjia Tang. Neurosurgeon: Collaborative intelligence between the cloud and mobile edge. *ACM SIGARCH Computer Architecture News*, 45(1):615–629, 2017.

[51] Mehrdad Khani, Ganesh Ananthanarayanan, Kevin Hsieh, Junchen Jiang, Ravi Netravali, Yuanchao Shu, Mohammad Alizadeh, and Victor Bahl. {RECL}: Responsive {Resource-Efficient} continuous learning for video analytics. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 917–932, 2023.

[52] Kubeflow. Kubeflow: The machine learning toolkit for kubernetes, 2021. Accessed: 2024-10-23.

[53] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*, 2023.

[54] Stefanos Laskaridis, Stylianos I Venieris, Mario Almeida, Ilias Leontiadis, and Nicholas D Lane. Spinn: synergistic progressive inference of neural networks over device and cloud. In *Proceedings of the 26th annual international conference on mobile computing and networking*, pages 1–15, 2020.

[55] En Li, Liekang Zeng, Zhi Zhou, and Xu Chen. Edge ai: On-demand accelerating deep neural network inference via edge computing. *IEEE Transactions on Wireless Communications*, 19(1):447–457, 2019.

[56] He Li, Kaoru Ota, and Mianxiong Dong. Learning iot in edge: Deep learning for the internet of things with edge computing. *IEEE network*, 32(1):96–101, 2018.

[57] KeDi Li and Ning Gui. Cms: A continuous machine-learning and serving platform for industrial big data. *Future Internet*, 12(6):102, 2020.

[58] Tian Li, Jie Zhong, Ji Liu, Wentao Wu, and Ce Zhang. Ease. ml: Towards multi-tenant resource sharing for machine learning workloads. *Proceedings of the VLDB Endowment*, 11(5):607–620, 2018.

[59] Xiangjie Li, Chenfei Lou, Yuchi Chen, Zhengping Zhu, Yingtao Shen, Yehan Ma, and An Zou. Predictive exit: Prediction of fine-grained early exits for computation- and energy-efficient inference. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 37, pages 8657–8665, 2023.

[60] Zhuohan Li, Eric Wallace, Sheng Shen, Kevin Lin, Kurt Keutzer, Dan Klein, and Joey Gonzalez. Train big, then compress: Rethinking model size for efficient training and inference of transformers. In *International Conference on machine learning*, pages 5958–5968. PMLR, 2020.

[61] Qianlin Liang, Walid A Hanafy, Noman Bashir, Ahmed Ali-Eldin, David Irwin, and Prashant Shenoy. Dělen: enabling flexible and adaptive model-serving for multi-tenant edge ai. In *Proceedings of the 8th ACM/IEEE Conference on Internet of Things Design and Implementation*, pages 209–221, 2023.

[62] llm d. Kubernetes-native distributed inference at scale. https://llm-d.ai, 2025. Accessed: 22 May 2025.

[63] Jiachen Mao, Xiang Chen, Kent W Nixon, Christopher Krieger, and Yiran Chen. Modnn: Local distributed mobile computing system for deep neural network. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*, pages 1396–1401. IEEE, 2017.

[64] Markets and Markets to 2023. Intelligent video analytics market, 2024. Accessed: 2024-10-23.

[65] MarketsandMarkets. Intelligent video analytics market - global forecast to 2028, 2024. Accessed: 2024-10-23.

[66] Yoshitomo Matsubara, Marco Levorato, and Francesco Restuccia. Split computing and early exiting for deep learning applications: Survey and research challenges. *ACM Computing Surveys*, 55(5):1–30, 2022.

[67] Peter Mattson, Vijay Janapa Reddi, Christine Cheng, Cody Coleman, Greg Diamos, David Kanter, Paulius Micikevicius, David Patterson, Guenther Schmuelling, Hanlin Tang, et al. Mlperf: An industry standard benchmark suite for machine learning performance. *IEEE Micro*, 40(2):8–16, 2020.

[68] Ruben Mayer and Hans-Arno Jacobsen. Scalable deep learning on distributed infrastructures: Challenges, techniques, and tools. *ACM Computing Surveys (CSUR)*, 53(1):1–37, 2020.

[69] ETSI Industry Specification Group (ISG) MEC. Mobile-edge computing (mec) - introductory technical white paper. Technical report, European Telecommunications Standards Institute (ETSI), September 2014. Version 1.

[70] Hui Miao, Ang Li, Larry S Davis, and Amol Deshpande. Modelhub: Deep learning lifecycle management. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*, pages 1393–1394. IEEE, 2017.

[71] MlFlow. Deep Learning Container. https://mlflow.org/docs/latest/model-registry.html. [Online; accessed 26-Jan-2024].

[72] Moby. Buildkit. https://github.com/moby/buildkit, 2024. Accessed: 21 October 2024.

[73] ModelHub Contributors. Modelhub: Empowering ai researchers to share fully reproducible and portable model implementations. http://modelhub.ai, 2024. Accessed: October 23, 2024.

[74] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I Jordan, et al. Ray: A distributed framework for emerging {AI} applications. In *13th USENIX symposium on operating systems design and implementation (OSDI 18)*, pages 561–577, 2018.

[75] Neptune. Deep Learning Container. https://neptune.ai/blog/ml-model-registry. [Online; accessed 26-Jan-2024].

[76] Neptune.ai. Ml model registry: What it is, why it matters, and how to implement it. https://neptune.ai/blog/ml-model-registry, 2024. Accessed: October 23, 2024.

[77] Chanh Nguyen, Amardeep Mehta, Cristian Klein, and Erik Elmroth. Why cloud applications are not ready for the edge (yet). In *Proceedings of the 4th ACM/IEEE Symposium on Edge Computing*, pages 250–263, 2019.

[78] David Nigenda, Zohar Karnin, Muhammad Bilal Zafar, Raghu Ramesha, Alan Tan, Michele Donini, and Krishnaram Kenthapadi. Amazon sagemaker model monitor: A system for real-time insights into deployed machine learning models. In *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 3671–3681, 2022.

[79] Seyed Ali Osia, Ali Shahin Shamsabadi, Sina Sajadmanesh, Ali Taheri, Kleomenis Katevas, Hamid R Rabiee, Nicholas D Lane, and Hamed Haddadi. A hybrid deep learning architecture for privacy-preserving mobile analytics. *IEEE Internet of Things Journal*, 7(5):4505–4518, 2020.

[80] Arthi Padmanabhan, Neil Agarwal, Anand Iyer, Ganesh Ananthanarayanan, Yuanchao Shu, Nikolaos Karianakis, Guoqing Harry Xu, and Ravi Netravali. Gemel: Model merging for {Memory-Efficient},{Real-Time} video analytics at the edge. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 973–994, 2023.

[81] Andrei Paleyes, Raoul-Gabriel Urma, and Neil D Lawrence. Challenges in deploying machine learning: a survey of case studies. *ACM computing surveys*, 55(6):1–29, 2022.

[82] Misun Park, Ketan Bhardwaj, and Ada Gavrilovska. Pocket: ml serving from the edge. In *Proceedings of the Eighteenth European Conference on Computer Systems*, pages 46–62, 2023.

[83] Pratyush Patel, Esha Choukse, Chaojie Zhang, Aashaka Shah, Íñigo Goiri, Saeed Maleki, and Ricardo Bianchini. Splitwise: Efficient generative llm inference using phase splitting. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, pages 118–132. IEEE, 2024.

[84] Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, Jacob Devlin, James Bradbury, Jonathan Heek, Kefan Xiao, Shivani Agrawal, and Jeff Dean. Efficiently scaling transformer inference. *Proceedings of Machine Learning and Systems*, 5:606–624, 2023.

[85] Minghai Qin, Chao Sun, Jaco Hofmann, and Dejan Vucinic. Disco: Distributed inference with sparse communications. In *Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision*, pages 2432–2440, 2024.

[86] Samyam Rajbhandari, Conglong Li, Zhewei Yao, Minjia Zhang, Reza Yazdani Aminabadi, Ammar Ahmad Awan, Jeff Rasley, and Yuxiong He. Deepspeed-moe: Advancing mixture-of-experts inference and training to power next-generation ai scale. In *International conference on machine learning*, pages 18332–18346. PMLR, 2022.

[87] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. Enabling ai at the edge with xnor-networks. *Communications of the ACM*, 63(12):83–90, 2020.

[88] Francisco Romero, Qian Li, Neeraja J Yadwadkar, and Christos Kozyrakis. {INFaaS}: Automated model-less inference serving. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 397–411, 2021.

[89] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4510–4520, 2018.

[90] Harshit Saokar, Soteris Demetriou, Nick Magerko, Max Kontorovich, Josh Kirstein, Margot Leibold, Dimitrios Skarlatos, Hitesh Khandelwal, and Chunqiang Tang. {ServiceRouter}: Hyperscale and minimal cost service mesh at meta. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 969–985, 2023.

[91] Mahadev Satyanarayanan, Wei Gao, and Brandon Lucia. The computing landscape of the 21st century. In *Proceedings of the 20th International Workshop on Mobile Computing Systems and Applications*, pages 45–50, 2019.

[92] Sudipta Saha Shubha and Haiying Shen. Adainf: Data drift adaptive scheduling for accurate and slo-guaranteed multiple-model inference serving at edge servers. In *Proceedings of the ACM SIGCOMM 2023 Conference*, pages 473–485, 2023.

[93] DeepSpeed Team. Deepspeed: Accelerating deep learning training. https://github.com/deepspeedai/DeepSpeed, 2025. Accessed: 2025-05-24.

[94] TensorFlow Team. Tensorflow serving with docker, 2024. Accessed: 2024-10-23.

[95] Surat Teerapittayanon, Bradley McDanel, and Hsiang-Tsung Kung. Branchynet: Fast inference via early exiting from deep neural networks. In *2016 23rd international conference on pattern recognition (ICPR)*, pages 2464–2469. IEEE, 2016.

[96] Surat Teerapittayanon, Bradley McDanel, and Hsiang-Tsung Kung. Distributed deep neural networks over the cloud, the edge and end devices. In *2017 IEEE 37th international conference on distributed computing systems (ICDCS)*, pages 328–339. IEEE, 2017.

[97] Guanhua Wang, Zhuang Liu, Brandon Hsieh, Siyuan Zhuang, Joseph Gonzalez, Trevor Darrell, and Ion Stoica. sensai: Convnets decomposition via class parallelism for fast inference on live data. *Proceedings of Machine Learning and Systems*, 3:664–679, 2021.

[98] Xiaofang Wang, Dan Kondratyuk, Eric Christiansen, Kris M Kitani, Yair Alon, and Elad Eban. Wisdom of committees: An overlooked approach to faster and more accurate models. *arXiv preprint arXiv:2012.01988*, 2020.

[99] Carole-Jean Wu, David Brooks, Kevin Chen, Douglas Chen, Sy Choudhury, Marat Dukhan, Kim Hazelwood, Eldad Isaac, Yangqing Jia, Bill Jia, et al. Machine learning at facebook: Understanding inference at the edge. In *2019 IEEE international symposium on high performance computer architecture (HPCA)*, pages 331–344. IEEE, 2019.

[100] Guoxuan Xia and Christos-Savvas Bouganis. Window-based early-exit cascades for uncertainty estimation: When deep ensembles are more efficient than single models. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 17368–17380, 2023.

[101] Zongheng Yang, Zhanghao Wu, Michael Luo, Wei-Lin Chiang, Romil Bhardwaj, Woosuk Kwon, Siyuan Zhuang, Frank Sifei Luan, Gautam Mittal, Scott Shenker, et al. {SkyPilot}: An intercloud broker for sky computing. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 437–455, 2023.

[102] Matei Zaharia, Andrew Chen, Aaron Davidson, Ali Ghodsi, Sue Ann Hong, Andy Konwinski, Siddharth Murching, Tomas Nykodym, Paul Ogilvie, Mani Parkhe, et al. Accelerating the machine learning lifecycle with mlflow. *IEEE Data Eng. Bull.*, 41(4):39–45, 2018.

[103] Liekang Zeng, Xu Chen, Zhi Zhou, Lei Yang, and Junshan Zhang. Coedge: Cooperative dnn inference with adaptive workload partitioning over heterogeneous edge devices. *IEEE/ACM Transactions on Networking*, 29(2):595–608, 2020.

[104] Hong Zhang, Yupeng Tang, Anurag Khandelwal, and Ion Stoica. {SHEPHERD}: Serving {DNNs} in the wild. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 787–808, 2023.

[105] Huaifeng Zhang, Mohannad Alhanahnah, Fahmi Abdulqadir Ahmed, Dyako Fatih, Philipp Leitner, and Ahmed Ali-Eldin. Machine learning systems are bloated and vulnerable. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 8(1):1–30, 2024.

[106] Yizhen Zhao. Machine learning in production: A literature. Technical report, Tech. Rep, 2021.

[107] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. {DistServe}: Disaggregating prefill and decoding for goodput-optimized large language model serving. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 193–210, 2024.

[108] Li Zhou, Hao Wen, Radu Teodorescu, and David HC Du. Distributing deep neural networks with containerized partitions at the edge. In *2nd USENIX Workshop on Hot Topics in Edge Computing (HotEdge 19)*, 2019.

Appendix A Packaging ML Models

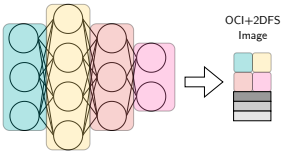


Figure 15: ML Model Packaging Example.

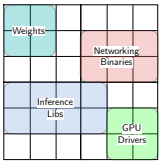


Figure 16: High Level Space Partitions.

As for traditional containers, also in 2DFS the build and caching efficiency depends on the best practices on how to design the 2D space. When building a traditional Docker image the user uses a Dockerfile to define the operations composing the layers of the image. The order of the instruction heavily impacts the performance of the build as well as the wasted space in the final image. Docker provides a set of best practices to follow when designing a Dockerfile [26]. 2DFS performance heavily on the allotment sizes. The smaller the alltoments, the better the parallelization at build time and caching efficiency. For large models, we suggest splitting the model layers into separate files even if the model does not support partitions. This way, selective model updates invalidate only part of the 2D field, speeding up the build process. Figure 15 shows an example of a very simple model split technique that can significantly improve the 2DFS performance. By simply splitting the fully connected layers into separate files, we can package these files as independent allotments.

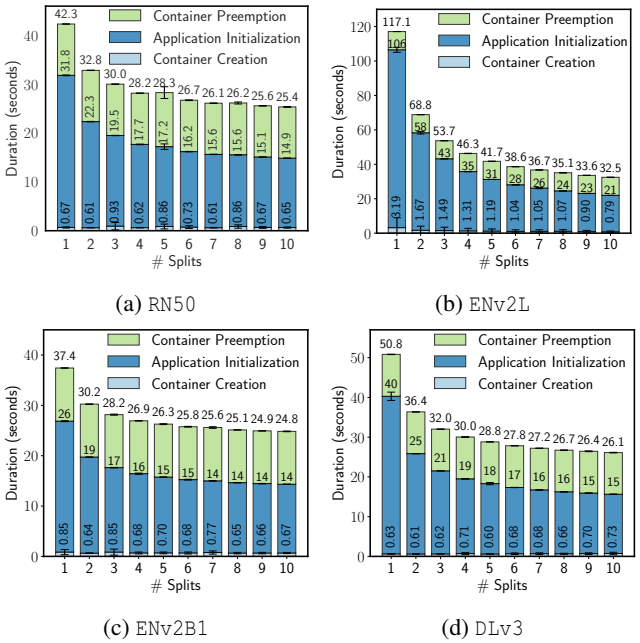
In more advanced usecases, the 2D space can be organized into scope-based partitions. E.g., in fig. 16, we place the model weights in the top-left corner, the networking binaries in the middle, the inference libs in the bottom left, and the GPU drivers in the bottom-right corner. Suppose the model is executed on a device without GPU; the bottom right corner can be left out of the partition. Alternatively, during training, we can remove the inference libs and networking binaries.

Appendix B Additional Results

		Split Capacity %				
Model	Format	0	25	50	75	100
ENv2L	Docker	451.84	451.88	451.92	451.97	452.00
	2DFS	451.68	451.69	451.70	451.72	451.73
RN50	Docker	124.87	124.88	124.89	124.89	124.90
	2DFS	124.64	124.64	124.65	124.65	124.65

Table 3: Exported Image Sizes Comparison with Different Split Capacities.

Figure 18 shows the parameter count and size distribution of different splits, showcasing the variability among the models of the zoo (Table 2). Note that although the splits closer to the end of a model comprise more parameters, this does not translate 1:1 to additional computation.



(a) RN50 (b) ENv2L (c) ENv2B1 (d) DLv3

Figure 17: Deployment time breakdown for different numbers of splits / available devices for two small (RN50 and DLv3), one medium (ENv2B1) and one large (ENv2L) models.

Figure 17 complements the experiments in §4.5 and it's showing the deployment time of RN50, ENv2L, ENv2B1 and DLv3for different numbers of devices. The number of splits equals the number of devices, and the image layers are homogeneously distributed across the splits. The deployment time is broken down into the time required to download the image, the time required to unpack the image, and the time required to start the application. The deployment time decreases logarithmic with the number of splits as shown also in fig. 14a. ENv2Lis showing a steeper performance improvement because, on the constrained Edge devices, the initialization phase is much slower when too many parameters are handled in a single node. With ten splits, we observe a 3.5× faster deployment time.

Figure 19 complements the results from fig. 7 with the models omitted for brevity. The results show the same Docker trends, with the small models showing little variation, the medium-sized models showing a build time decrease with a higher number of splits, the large model showing their best build performance at around 50% split capacity, and then higher build times towards 75% and 100%. As explained in §4.2, for Docker, the is a tradeoff between the number of layers and the size. 2DFS performance improves consistently with the number of allotments, plateauing at around 50%, hitting the I/O disk capacity.

Figure 20 complements the results from fig. 8 with the models omitted for brevity. The results show the same Docker trends, with build time consistently increasing when more

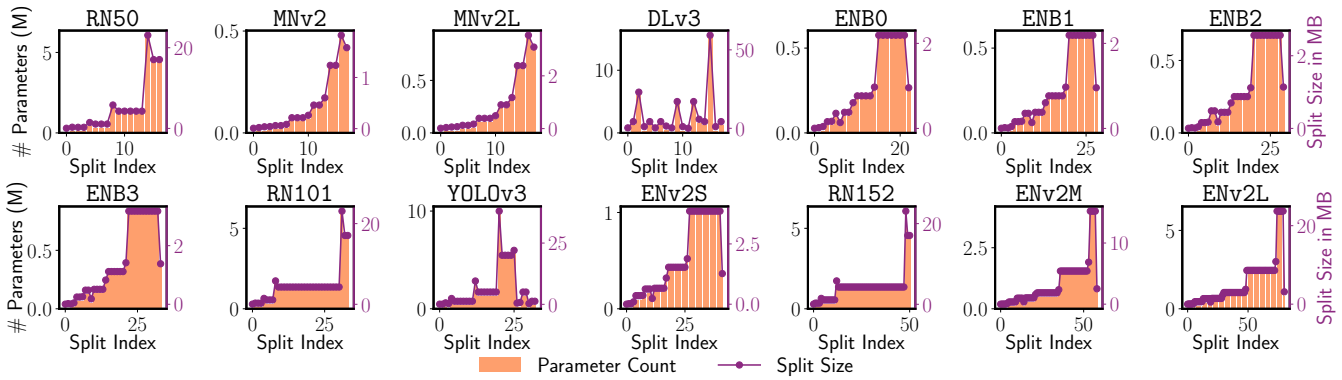


Figure 18: Per-Split Parameter Count and Byte Size.

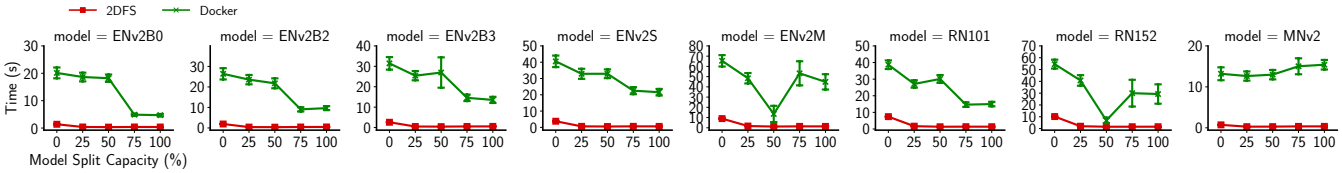


Figure 19: Build Time of 1 Image with Different Amount of Splits per Layer

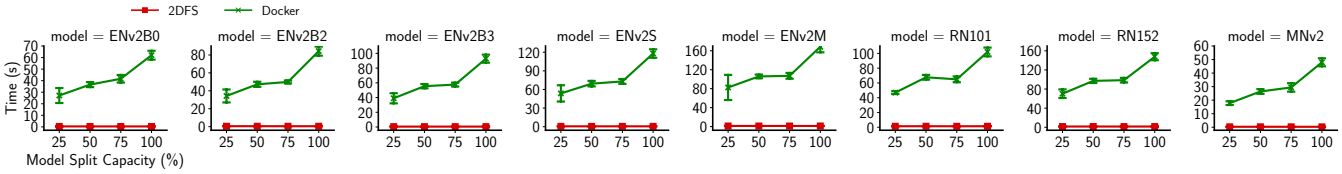


Figure 20: Images Build Time for different Split Partitions - 1 Image per Partition with Uniform Partitions.

images are generated. `2DFS` shows almost constant build time because it always generates the same `OCI+2DFS` image.

In §B compare the size of the exported `OCI+2DFS` image and the traditional `OCI` images to measure the space overhead of the new additional `2dfs.field` layer. We build the images for the `ENv2L`and `RN50` models with different split capacities, the same as previously done in fig. 7. For each split capacity, we build and export in `tar.gz` format one OCI image using Docker and one `OCI+2DFS`. The images show comparable sizes, with no additional space overhead for the `2dfs.field` layer. This happens because the two-dimensional data structure introduced is serialized as an array of hash-pointers + metadata, efficiently replacing the traditional OCI layers.

ARTIFACT EVALUATED
usenix ASSOCIATION
AVAILABLE

PathWeaver: A High-Throughput Multi-GPU System for Graph-Based Approximate Nearest Neighbor Search

Sukjin Kim Seongyeon Park Si Ung Noh
Junguk Hong Taehee Kwon Hunseong Lim Jinho Lee

Seoul National University

Abstract

Graph-based Approximate Nearest Neighbor Search (ANNS) is widely adopted in numerous applications, such as recommendation systems, natural language processing, and computer vision. While recent works on GPU-based acceleration have significantly advanced ANNS performance, the ever-growing scale of datasets now demands efficient multi-GPU solutions. However, the design of existing works overlooks multi-GPU scalability, resulting in naïve approaches that treat additional GPUs as a means to extend memory capacity for large datasets. This inefficiency arises from partitioning the dataset and independently searching for data points similar to the queries in each GPU. We therefore propose PathWeaver, a novel multi-GPU framework designed to scale and accelerate ANNS for large datasets. First, we propose pipelining-based path extension, a GPU-aware pipelining mechanism that reduces prior work's redundant search iterations by leveraging GPU-to-GPU communication. Second, we design ghost staging that leverages a representative dataset to identify optimal query starting points, reducing the search space for challenging queries. Finally, we introduce direction-guided selection, a data selection technique that filters irrelevant points early in the search process, minimizing unnecessary memory accesses and distance computations. Comprehensive evaluations across diverse datasets demonstrate that PathWeaver achieves 3.24× geomean speedup and up to 5.30× speedup on 95% recall rate over state-of-the-art multi-GPU-based ANNS frameworks.

1 Introduction

Various fields, such as computer vision [11, 34, 39], recommendation systems [13], natural language processing [35, 58], and information retrieval [17], utilize datasets consisting of multi-dimensional vectors representing larger data entities (e.g., image, text). These representations are used to efficiently retrieve data relevant to the input queries in various applications. One solution to finding k-closest data points (vectors) is k-Nearest Neighbor Search, where the k-closest data points are selected from the entire dataset based on the user-defined similarity metric, commonly the L2 distance. However, as the dataset grows, finding the exact solution becomes increasingly difficult due to the curse of dimensionality [15, 28, 38]. To address this challenge, Approximate Nearest Neighbor Search (ANNS) has been widely adopted, especially by products such as vector databases [2, 3, 5, 6, 18, 55]. ANNS performs nearest neighbor search on large datasets with reasonable execution times while maintaining high accuracy.

Among various ANNS solutions [23, 27, 32, 48], graph-based methods [20, 21, 40, 41, 54] have gained significant attention due to their ability to effectively represent neighbor relationships between data points. This representation allows for evaluating fewer data points while achieving higher accuracy compared to alternative methods [56]. Consequently, there has been a growing body of work on optimizing graph-based ANNS performance across diverse hardware platforms. While CPU-based optimizations [9, 12, 16, 22, 42, 43, 61, 64] offer algorithms that maintain accuracy, they are limited by the number of available cores, hindering their scalability when processing large, high-dimensional datasets that demand extensive computation and memory access.

To address such a computational burden, recent works [25, 31, 41, 44, 50, 53, 65] propose GPU-optimized search techniques to leverage the high-performance capabilities of GPUs. A prominent example is CAGRA [44], which accelerates single-GPU search by efficiently computing the L2 distance between queries and data points. It further leverages hash tables to eliminate redundant computations while optimizing for the GPU's thread and memory hierarchy. Another example is GGNN [25], which partitions large datasets into smaller, independent graphs that fit within a single GPU. This enables data-parallel searches across multiple GPUs.

While these approaches demonstrate significant performance improvements over CPU-based solutions, we identify key bottlenecks that hinder their ability to fully exploit the potential of GPUs:

- **Existing multi-GPU solutions suffer from low scale efficiency.** Prior works [25, 31] divide the dataset across GPUs to support ANNS on large datasets with GPUs, with each GPU independently processing queries (i.e., graph sharding). While this approach enables handling large datasets on GPUs, it requires each query to be processed multiple times across different GPUs, resulting in a low scale efficiency.
- **Majority of random initial nodes turn out to be unnecessary later.** To quickly search for near-optimal data points, existing approaches [41, 44, 50] rely on starting searches from numerous random initial points. However, due to the underlying beam search method, most of the neighbors explored from those random initial points are quickly discarded within a few iterations, leaving only descendants of the few best points. This incurs too much computational and memory overhead for the search, which is unnecessary for the final result.
- **Each iteration mandates too much overhead.** Visiting a vertex in a proximity graph yields distance computation with all its neighbors. Given that the typical degree of proximity graphs is around a few tens, this is a huge burden to the search. However, many of these neighboring data points are not selected as part of the top-k results, leading to unnecessary memory access and computation that hinders overall efficiency.

Based on these observations, we introduce **PathWeaver**, a multi-GPU framework designed to support graph-based ANNS on large datasets with scalable performance with minimal accuracy loss. PathWeaver is carefully designed to address the limitations above while exploiting the parallelism of GPU resources.

To address the scalability limitations of sharding-based multi-GPU search, we leverage our finding that each shard's seemingly independent search operations can be optimized by sharing intermediate search results with the following shard. We propose **pipelining-based path extension**, a mechanism that passes the search results of each shard to the next GPU in a pipelined manner. This enables subsequent GPUs to start their search from data points closer to the query within their shard, effectively reducing the number of search iterations only with negligible inter-GPU communication cost. One limitation of pipelining-based path extension lies in the first stage, whose search still has to be done from scratch. Our second scheme **ghost staging** addresses this issue by selecting more optimal starting points for each shard by prioritizing data points closer to the query, thereby improving the efficiency of the first stage. Finally, we propose **direction-guided selection**, which skips distance calculations for neighbors whose direction from the parent node significantly deviates from the direction toward the query. This approach further minimizes unnecessary distance computations, reducing both computational and memory overhead.

We implement the proposed techniques in **PathWeaver** and conduct extensive experiments on large datasets. PathWeaver improves the performance by 3.24× geomean speedup compared to the state-of-the-art GPU-based ANNS baselines at 95% recall rate, demonstrating its scalability and effectiveness.

2 Background

2.1 Approximate Nearest Neighbor Search

Given a dataset $\mathcal{D} = \{x_0, x_1, \cdots, x_{n-1} \mid x_i \in \mathbb{R}^d\}$ with n data points represented as d-dimensional vectors, and a query $q \in \mathbb{R}^d$, k-Nearest Neighbor Search (k-NNS) identifies k data points $N_k(q)$ that satisfies the following:

$$N_k(q) = \underset{\mathcal{S}\subset\mathcal{D},|\mathcal{S}|=k}{\operatorname{argmin}} \sum_{x_i\in\mathcal{S}} dist(q, x_i), \tag{1}$$

where $dist(q, x_i)$ is the similarity measure between the query and data point, typically the L2 distance. As the size of $\mathcal{D}$ grows, performing exact k-NNS becomes computationally expensive due to the substantial memory access and computation overhead. To address this, many applications adopt approximate nearest neighbor search (ANNS), which employs efficient algorithms to approximate k-NNS results with significant reductions in execution time [37].

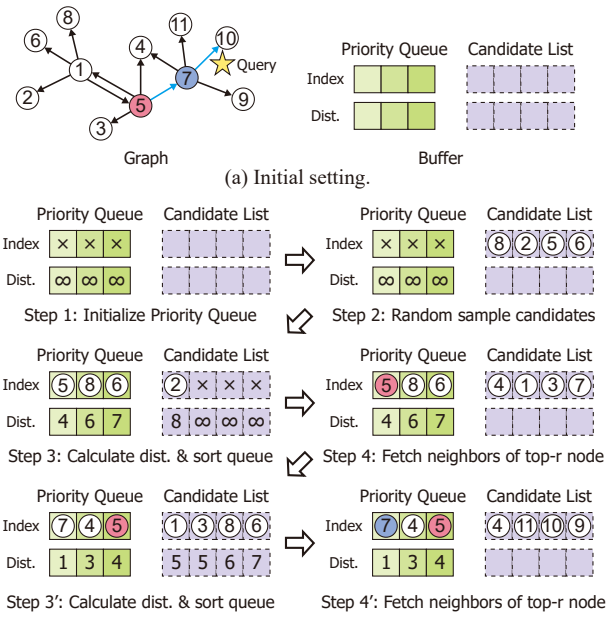
Among the various ANNS algorithms, graph-based ANNS [20, 21, 40, 41, 54] has been widely used due to its ability to effectively capture the relationships (distances) between data points [56]. Graph-based methods construct a proximity graph $\mathcal{G}$ where nodes represent data points and edges encode distances between them, allowing for compact representations of relationships and facilitating efficient graph traversal to approximate k-NNS results. Assuming that $\mathcal{G}$ is pre-constructed, the proposed PathWeaver targets accelerating the graph search process itself, addressing key inefficiencies and scaling challenges associated with large-scale datasets.

2.2 Approximate Nearest Neighbor Search Method on GPUs

Graph-based Approximate Nearest Neighbor Search (ANNS) algorithms have gained significant traction due to their ability to efficiently navigate complex datasets by leveraging graph structures. We briefly explain the overall search algorithm [44] on a constructed graph $\mathcal{G}$, as illustrated in Fig. 1a, to provide context for our optimizations. Note that many graph-based search algorithms [25, 53, 65] follow a similar workflow.

The graph-based search operates on two key data structures as shown in Fig. 1a:

- A *priority queue* $p = \{p_0, p_1, \ldots, p_{l-1}\}$ of size l, which stores the top-l ($k \le l$) intermediate nodes sorted by their distances to the query q.



(a) Initial setting.

(b) Stepwise execution of GPU-based ANNS.

Figure 1: Overview of approximate nearest neighbor search on GPUs.

- A *candidate list* $c = \{c_0, c_1, \ldots, c_{m-1}\}$ of size m, which acts as a buffer to hold the neighbors of the nodes being processed.

Given these structures, the algorithm iteratively refines the search process as depicted in Fig. 1b:

Step 1 The algorithm begins by initializing the priority queue p with dummy indices with ∞ distances.

Step 2 The candidate list c is populated with m randomly selected nodes from the graph.

Step 3 The distance $dist(q, c_i)$ is computed for each candidate node $c_i \in c$ using:

$$dist(q, c_i) = \|q - c_i\|_2, \tag{2}$$

where elements of the priority queue p and the candidate list c are sorted together based on the distances, ensuring that the closest nodes appear at the top of p.

Step 4 The neighbors of the top-r ($r \leq l$) nodes in the priority queue are fetched from $\mathcal{G}$ and the candidate list c is populated with these nodes.

Steps 3 and 4 are repeated until the priority queue receives no new entries or the pre-defined maximum number of iterations is reached. Then, the top-k nodes in p are returned as the approximate nearest neighbors.

This algorithm is based on carefully built proximity graphs, which encourage reachability (all vertices are reachable starting from any vertex) and convexity (avoids falling into local minima) [25, 44]. Based on such characteristics, the search algorithm aims for faster convergence towards the global minima by maintaining the priority queue that stores data points with small L2 distances and quickly dropping long-distance points.

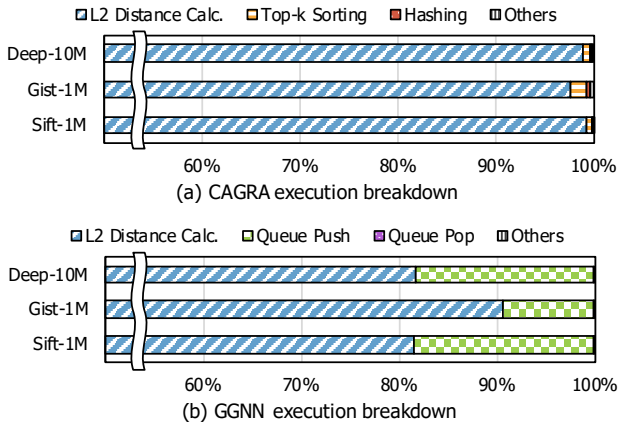


Figure 2: Execution time breakdown analysis of baseline ANNS.

Because the vector dimension typically ranges from a few tens to several hundreds, L2 distance calculation dominates the search time due to the overhead of loading and processing high-dimensional vectors. According to our measurements depicted in Fig. 2a, when using the current state-of-the-art method CAGRA [44], L2 distance calculation accounts for over 95% of the total search time, regardless of the dataset. To further analyze this trend across different methods, we also performed a breakdown of GGNN [25] and depicted in Fig. 2b found that over 80% of the time was still spent on distance calculations. Therefore, the key to high-throughput ANNS is to reduce the number of L2 distance calculations. For a search that converged in i iterations on a j-proximity graph (i.e., each vertex has j neighbors), the rough count of L2 distance calculation is $i \times j \times r$ (including duplicates). To achieve high throughput, PathWeaver mainly aims to reduce the number of search iterations i and number of examined neighbors out of j, while not sacrificing the search accuracy.

3 PathWeaver Design

3.1 Pipelining-based Path Extension

3.1.1 Scalability of Performance on Multiple GPUs

To diagnose the existing multi-GPU search method based on sharding [25], we compare the total number of search iterations between single- and multi-GPU setups using a multi-GPU extension of CAGRA and GGNN, as shown in Fig. 3a. The results indicate that the sharding-based approach scales inefficiently across multiple GPUs in both methods. For the

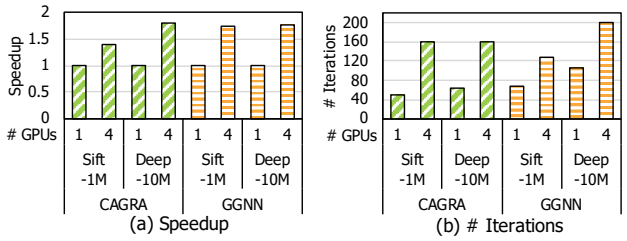


Figure 3: Performance scalability of prior work on Sift-1M and Deep-10M datasets with 4 GPUs.

Sift-1M dataset, using 4 GPUs yields only a 1.39× speedup in CAGRA, resulting in a scaling efficiency of approximately 35%. GGNN shows similar trends, achieving about a 1.7× speedup on both datasets, which corresponds to a scaling efficiency of around 43%. This inefficiency arises from the fact that the multi-GPU baseline performs a separate local search in each GPU's shard for every query. While this simple query-parallel design provides an easy parallelized solution for large datasets, the number of iterations needed does not reduce linearly with the size of a shard. As a result, the number of total iterations to complete a search increases with more number of shards, as depicted in Fig. 3b. In the case of the Sift-1M dataset in CAGRA method, the total iteration over all the shard scales with the number of shards results in 4× total iteration. In GGNN, increase of total iterations is slightly better at around 2×, but not enough to compensate for the overall scalability loss. This highlights the need for an alternative solution to achieve efficient performance.

3.1.2 Pipelining-based Path Extension Design

To amend the issue identified above, we propose *pipelining-based path extension*, a multi-GPU solution that enhances scalability by leveraging intermediate search results from other GPUs in a pipelined manner. As illustrated in Fig. 4a, the key idea is to create inter-shard edges from each node to the closest node in the adjacent shard.

The baseline sharding [25] (Fig. 4b) begins by randomly partitioning the input dataset into independent shards, with the number of shards matching the number of GPUs (4 shards in this example). After allocating these shards to each GPU, independent graphs are built for each shard/GPU. Note that in this baseline approach, there is no edge between the nodes in different graphs. Once the graphs are built, the baseline performs independent ANNS for all queries by starting from randomly selected nodes within each graph. Upon completing the search for all queries, each GPU stores the top-k data points for each query specific to its graph. For instance, after the searches are finished, $N \times k$ candidate data points are outputted across N GPUs. These results are then offloaded to the CPU for reduction to get the final top-k data points for each query. While this method provides an easily parallelizable solution with almost no communication, it requires more search iterations as revealed in Section 3.1.1 and thus scales inefficiently.

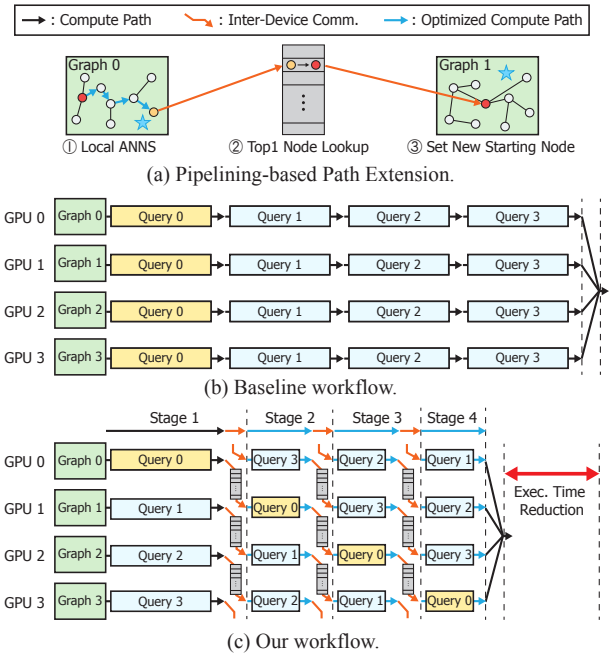


Figure 4: Illustration of the Pipelining-based path extension design.

In the proposed pipelining-based path extension design illustrated in Fig. 4c, the dataset is randomly partitioned in the same way as the baseline sharding approach to build independent graphs G_i for the shard on GPU i. However, after the graph construction, pipelining-based path extension takes a step further by creating uni-directional node-to-node connections between the shards of adjacent GPUs as shown in Fig. 4a. We define G_i to be adjacent to shards graph $G_{(i+1)\%N}$ for N GPUs, which form a ring topology. From each vertex in a shard, the nearest neighbor vertex is found in the adjacent shard, and an inter-shard edge is added between them. For example, the 0th GPU's graph connects all of its nodes to the closest node in the 1st GPU's graph, and nodes in the 3rd GPU to the 0th GPU, forming a ring-like uni-directional connection among graphs/shards. These inter-shard edges can be expressed as a mapping I:

$$I(u) = \underset{w \in G_{(i+1)\%N}}{\operatorname{argmin}} \; dist(u,w) \qquad \forall u \in G_i. \tag{3}$$

These connections are constructed once during the graph build phase and can be reused across searches. As detailed in Section 5.7, the additional overhead for creating these inter-shard edges is minimal.

To perform a search with the interconnected graph, each GPU is assigned a $|Q|/N$ chunk from the set of queries Q and starts the search as in the baseline sharding. After each query

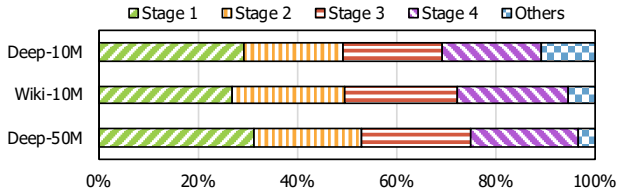
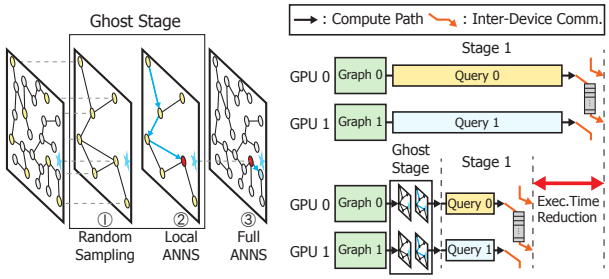


Figure 5: Execution time breakdown analysis with 4 GPUs after applying the pipelining-based path extension design according to the pipeline stages.



(a) Ghost Staging. (b) Pipeline with Ghost Stage.

Figure 6: Illustration of the ghost staging design.

has converged in all GPUs, a part of the final local result Z is taken. From each of its vertices $z \in Z$, it continues searching from $I(z)$, which naturally takes place in the adjacent shard as if the search path has been extended. Because $I(z)$ has been designed to be close to the query, the search converges significantly faster than the baseline in fewer iterations. For example, in Fig. 4c, when searching with query batch 0 (yellow box), the search process begins on GPU 0 with graph 0. After the search completes on GPU 0, the result is passed to GPU 1, which continues the search on graph 1. As shown in the figure, the search time in stage 2 (blue arrow) is reduced compared to that of stage 1 (black arrow). This process is executed simultaneously on all GPUs and is repeated in a pipelined manner until every GPU has processed the queries originating from all other GPUs. Then the results are reduced in the CPU, similarly to the baseline.

As discussed in Section 6.4, because the only data transfer involved is the local result between the adjacent GPUs, the communication overhead is almost negligible compared to the substantial reduction in search time.

3.2 Ghost Staging

3.2.1 Execution Time Breakdown after Pipelining-based Path Extension

Fig. 5 illustrates the execution time breakdown of searching with pipelining-based path extension. As expected, the first search stage of pipelining-based path extension is the primary bottleneck. For example, the first search stage in the Deep-50M dataset takes up to 31%, while the other stages consume only up to 22%.

This bottleneck arises as the first search stage does not benefit from pipelining-based path extension and starts from random, potentially distant data points. As a result, more iterations are required to converge on the appropriate top-k candidates. As all GPUs are performing the initial stage of their own query, reducing this overhead could bring additional speedup.

3.2.2 Ghost Staging Design

Motivated by pipelining-based path extension's ability to identify better starting points for subsequent GPU search stages, we propose *ghost staging*, a kind of hierarchical search [40] solution for improving the initial search stage on each GPU. Ghost staging adds a small auxiliary shard before the first stage, such that a high-quality data point close to the query can be located only in a few iterations. This approach significantly reduces the number of iterations needed during the initial search stage.

Fig. 6 illustrates how ghost staging reduces the number of search iterations during the first stage while maintaining accuracy. To create an auxiliary shard, ① ghost staging randomly samples a fixed number of random data points within a shard, referred to as ghost nodes. These ghost nodes are connected based on distances, creating a lightweight network. In addition, the inter-shard edges are added between the ghost nodes and the original nodes. Due to the relatively small number of ghost nodes, the preparation for ghost staging only adds negligible overhead to the graph build process (see Section 5.7). During ghost staging, ② a fixed number of starting points are chosen among the ghost nodes and the search algorithm from Section 2.2 is applied to locate ghost nodes near the query within a few iterations. ③ Once the relevant ghost nodes are identified, the search transitions to the original graph similar to pipelining-based path extension.

The high performance of ghost staging stems from ghost nodes acting as central hubs and their interconnections as highways within the expansive original graph. Moreover, ghost staging maintains high accuracy by locally searching the original graph for the remaining iterations. This is highlighted by comparing the search process of the baseline method and ghost staging in Fig. 6b. To reach the same data point close to the query, the baseline has to traverse many vertices due to the original graph's size. On the other hand, during ghost staging, taking a single step forward among ghost nodes has a similar effect of passing through multiple original graph data points. Even if ghost staging overshoots and bypasses the query, it can backtrack by the remaining search on the original graph.

Table 1: Unused Distance Calculation

Dataset	#Total Visits	#Discarded Visits	Ratio
Sift-1M [8, 32]	2.32E+7	2.00E+7	86.2%
Gist-1M [8, 32]	3.17E+6	2.81E+6	88.9%
Deep-10M [10]	2.68E+7	2.28E+7	85.0%

3.3 Direction-Guided Selection

3.3.1 Analysis on Unused Distance Calculation

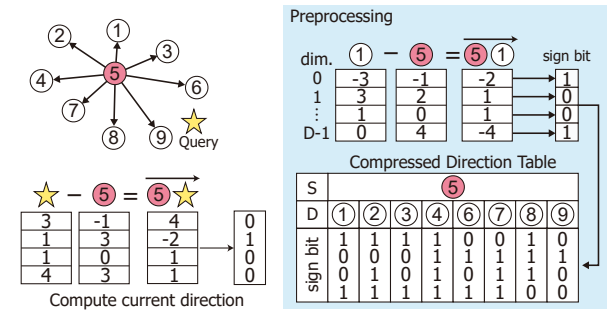
Despite reducing the search iterations with pipelining-based path extension and ghost staging, the search procedure remains heavily dominated by the numerous L2 distance calculations. This partially stems from having to add all neighbors of each top-r node in the priority queue (Section 2.2), where the number of neighbors is usually in the order of a few tens [44]. While such a large number of neighbors is necessary to ensure reachability and convexity, it often introduces unnecessary computational overhead. During the search, we found that the majority of the data points added to the priority queue never reached the top-k region of the priority queue and were eventually discarded (unused) without being considered again because they were too far from the query compared to the other candidates.

In Table 1, we quantitatively measured the portion of the considered nodes, where **Total Visits** denotes the total number of nodes in the proximity graph that are accessed during the search, and **Discarded Visits** represents the nodes that are visited but never remain in the candidate buffer until the end of the search process to be included in the final top-k results. As depicted in the table, we find that the ratio of discarded node visits, which lead to unnecessary distance calculations, can exceed 80%. This indicates an interesting opportunity for additional optimization, where we could remove a substantial portion of the unused calculations if we can identify them in advance.

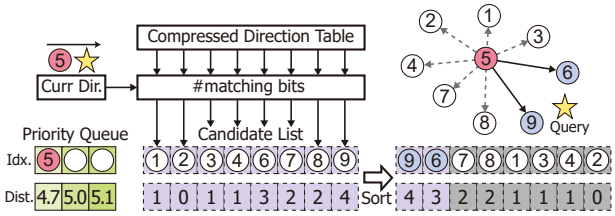
3.3.2 Direction-Guided Selection Design

We propose *direction-guided selection*, an optimization technique designed to accelerate the search process within each iteration. Direction-guided selection allows the search algorithm to bypass data points that are significantly unaligned with the query's direction relative to the current target node, as shown in the top graph in Fig. 7. By leveraging a set of lightweight operations including direction table reads and local sorting, direction-guided selection significantly reduces expensive vector reads and distance calculations.

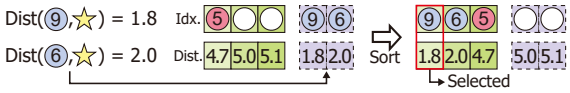
The direction-guided selection process operates as follows: (a) During the off-line preprocessing phase, a compressed direction table is generated, storing the sign bits of the differences between the source node and its neighbors in the proximity graph. Each sign bit vector encodes the rough direction of an edge [46] and serves as a fast proxy to guide traversal during run-time search on the GPU. (b) On each iteration of the search, the sign bits of the direction between the query and the visiting node (node 5 in the example) are calculated. Then the number of matching bits is counted for each neighbor using the compressed direction table. The neighbors are sorted by their number of matching bits in descending order, and the top-n candidates are selected. In the example, with $n = 2$, nodes 9 and 6 are selected. (c) The distance is calculated between the query and the selected neighbors (node 9 and 6 in the example), and the priority queue elements are sorted by the order of their distances (e.g., nodes 9, 6, 5). Finally, top-r nodes from the priority queue are selected (e.g., node 9 with $r = 1$) and they become the next visiting nodes. To maintain accuracy, the last few iterations are regarded as cool-down phase, and the search is conducted without the selection. We use 30% of the max iteration as the default value.



(a) Compute direction to query via sign bits.



(b) Sort & select candidates by directional similarity.



(c) Compute distance to query & select top-r nodes.

Figure 7: Illustration of the direction-guided selection design.

Direction-guided selection effectively reduces the number of distance calculations while maintaining high accuracy. This is achieved by prioritizing candidates most aligned with the query's direction, significantly narrowing the search space. There is a small chance that direction-guided selection can drop meaningful candidates, because of the compression error in the direction table. However, due to the reachability and

convexity of the proximity graphs, this would typically result in an increased number of iterations until convergence instead of accuracy. We empirically found that the accuracy drop is almost negligible while the gain from the reduction in candidates dwarfs the increase in the number of iterations.

4 Implementation Details

PathWeaver's search kernel has been implemented based on the CAGRA search kernel, adopting its query-per-threadblock approach along with GPU-friendly bitonic sorting and hash tables. To maximize the benefits of fast register communication within a warp, we set the threadblock size to be equal to that of a warp (i.e., 32 threads).

To minimize the communication overhead in pipelining-based path extension, the total set of queries is sent to all devices, which has a negligible size compared to the dataset. Each device processes independent graphs generated from its respective shard and maintains a unique inter-shard edge table for its linked shard. At runtime, only a partial chunk of the query batch is processed, and the result is forwarded to the next GPU device.

While the number of results sent per query is a tunable parameter, we empirically choose to send only one to minimize communication overhead. Additionally, for pipelining-based path extension, a shard-to-shard lookup table needs to be precomputed. Every node within a single shard acts as a query to perform a search in the adjacent shard. The top-1 result from the search is stored in the lookup table for pipelining-based path extension.

Ghost staging can be implemented similarly to pipelining-based path extension, except that instead of connecting neighboring shards, it connects the extracted smaller auxiliary shard with the original shard.

In direction-guided selection, during processing of the sign bit table, all 1-bit signs are packed into one uint32. To efficiently compress the vector between the current node and the query, each thread subtracts the vector's elements and converts them into 1-bit values. During the runtime search process, the `__shfl_xor_sync()` intrinsic is used within a warp to enable efficient processing by performing fast intra-warp shuffle operations. Next, the precomputed sign bit table is looked up to retrieve the approximate sign bits of neighboring vectors. The similarity with the query vector is then computed by performing bit-wise XOR operations on uint32 units, followed by a `__popcll()` intrinsic call to count the total number of different bits. Finally, a min-sort is performed to find the node with the highest similarity. As mentioned above, direction-guided selection requires a precomputed compressed sign bit table generated on the CPU using multi-threading, where each thread handles the edges of a single parent node. For each neighbor, element-wise comparisons are performed, and the results are compressed into a uint32 using bit-wise shift and OR operations.

Table 2: Datasets used in evaluation

Target	Dataset	Dim. (d)	Size (n)	Type
Single GPU	Sift-1M [8, 32]	128	1M	float
	Gist-1M [8, 32]	960	1M	float
	Deep-10M [10]	96	10M	float
Multi-GPU	Wiki-10M [7]	768	10M	float
	Deep-10M [10]	96	10M	float
	Deep-50M [10]	96	50M	float

5 Evaluation

5.1 Experiment Setup

Environment. We have implemented and evaluated PathWeaver on a server with four NVIDIA RTX A6000 GPUs and an AMD EPYC 9124 16-Core CPU, where two GPUs are connected via NVLink Bridge [4], and each GPU pair is linked through a host PCIe switch. All environments run on Ubuntu 22.04 with CUDA version 12.1 and PyTorch 2.4.1.

Baselines. The following CPU and GPU baselines were chosen for evaluation.

- **CAGRA** [44] is the state-of-the-art GPU framework for graph-based ANNS. CAGRA proposes a heuristically optimized proximity graph for parallel search operations and incorporates techniques such as warp splitting and forgettable hashing to enhance search performance. We used the official implementation of the authors, where we extended it for multi-GPU settings (denoted as 'CAGRA w/ Sharding').

- **HNSW** [40] is a popular graph-based ANNS implemented for CPUs. HNSW introduces a hierarchical proximity graph, with each layer structured as an NSW [41] graph representing a subset of points. The search operation begins at the top layer and traverses through the hierarchy, achieving improved performance and accuracy. Because HNSW only supports execution on CPUs, we evaluated HNSW only in the single-GPU environment for fair comparison. We utilized 64 threads on CPUs for evaluation.

- **GGNN** [25] is another GPU implementation for graph-based ANNS. GGNN builds on the HNSW-inspired proximity graph, specifically optimizing it to leverage massive parallelism during the graph construction phase. Additionally, GGNN enhances search performance by efficiently utilizing shared memory and enabling parallel operations on data structures. To the best of our knowledge, GGNN is the only graph-based ANNS framework that supports multiple GPUs out of the box, where it uses the sharding method.

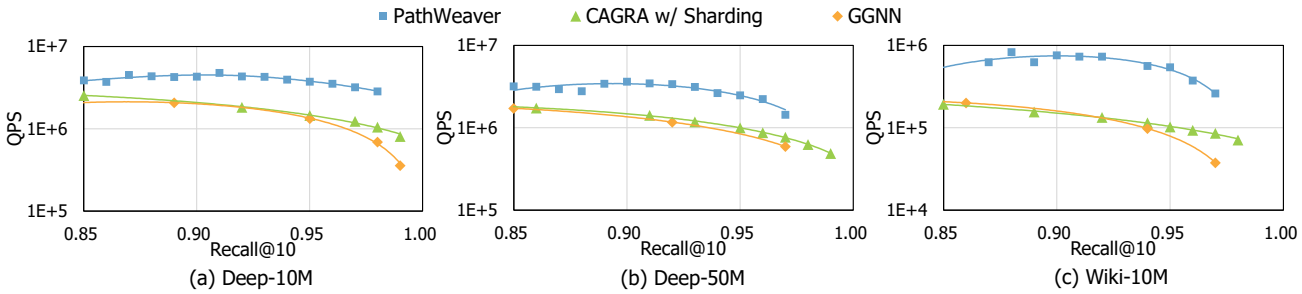


Figure 8: Performance comparison on multi-GPU environment.

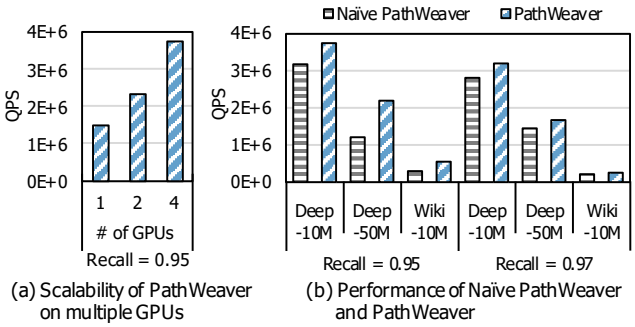


Figure 9: Performance comparison of Recall@10 between the naïve and pipeline method on multiple GPUs.

Datasets. We evaluate our method on a total of six datasets in which the vector dimension ranged from 96 to 960 as reported in Table 2. We analyzed single-GPU performance using datasets of varying sizes (1M, 10M) and dimensions (96, 128, 960) to demonstrate our method's effectiveness. For multi-GPU experiments, we selected larger datasets, scaling up to 50M in size and 768 in dimension. We took the first 10 million and the first 50 million part to create the Deep-10M and Deep-50M datasets from the Deep-1B dataset [1]. For graph building of PathWeaver, we used CAGRA's graph build algorithm, which offers the fastest build speed on GPUs. To ensure fairness, the out-degree of graphs was fixed to 64 for all datasets for PathWeaver's search and the CAGRA baseline evaluation.

Query Batch Size. We evaluated the performance and accuracy of PathWeaver and baselines using a query batch size of 10,000 for single-GPU tests. The exception was Gist-1M, where we used a batch size of 1,000. For multi-GPU evaluations, we employed a batch size of 60,000 to achieve high throughput and fully utilize multiple devices.

Performance Metrics. We tested two key metrics for the evaluation, following the typical performance metrics in the previous ANNS frameworks. The two key metrics and the details are as follows:

1. **Recall**: This metric quantifies the accuracy of the ANNS results by comparing them to the ground truth k-NNS results. For a query q, recall is defined as:

$$Recall@k = \frac{|\mathcal{N}_k^{KNNS}(q) \cap \mathcal{N}_k^{ANNS}(q)|}{|\mathcal{N}_k^{KNNS}(q)|}, \quad (4)$$

where $\mathcal{N}_k^{KNNS}(q)$ represents the k-NNS result, and $\mathcal{N}_k^{ANNS}(q)$ is the result obtained from the ANNS framework. Similar to prior work [25, 44, 53, 65], we target 95% recall@10 for most of the analyses unless otherwise noted.

2. **Queries Per Second (QPS)**: This metric captures the throughput of the framework by measuring the number of queries processed per second.

The primary objective of ANNS frameworks is to balance these two metrics—achieving high recall to ensure accuracy while maximizing QPS to deliver fast query processing.

5.2 Evaluation on Multiple GPUs

5.2.1 Performance Comparison on Multiple GPUs

Fig. 8 presents the performance comparison of PathWeaver with other multi-GPU baselines in a QPS-recall plot. Because CAGRA [44] does not support a multi-GPU environment out-of-box, we extended it with the sharding method as discussed by the authors. As shown in the results, PathWeaver outperforms all baselines. At a high recall of 95%, PathWeaver achieves 3.24× geomean speedup over CAGRA, the best-performing baseline. Moreover, PathWeaver achieves at most 5.30× speedup at the same recall rate in the Wiki-10M dataset. A similar advantage is maintained at a moderate recall 88%–92%, where PathWeaver achieves 3.36× speedup over CAGRA.

Among datasets, Deep-10M and Deep-50M both show similar QPS of around 3×10^6 for PathWeaver and 1.1×10^6 for both CAGRA and GGNN. However, for Wiki-10M, the throughput is an order of magnitude lower in all frameworks because Wiki-10M is composed of very wide vector indices of 768. This aligns with our finding that the ANNS throughput is less impacted by the size of the graphs. Even though

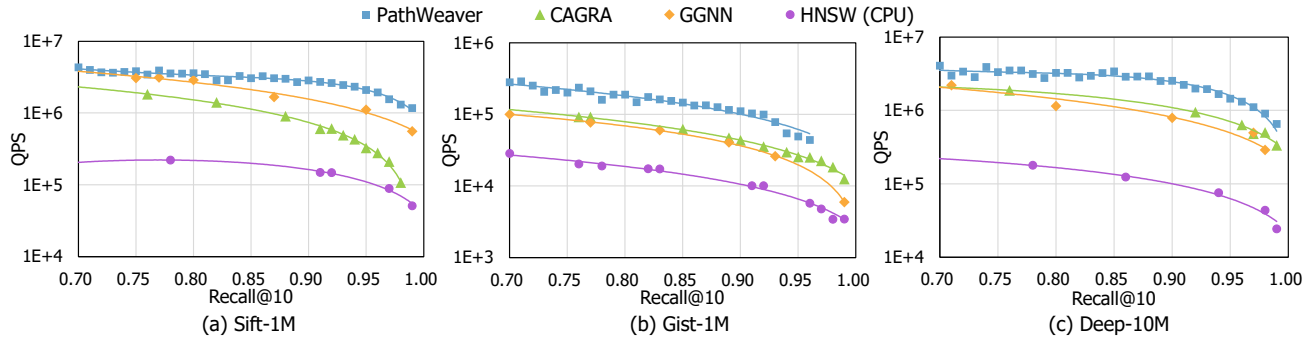


Figure 10: Performance comparison on a single GPU.

Deep-50M has five times more vertices than that of Deep-10M, the number of iterations until convergence is similar, which supports the motivation for pipelining-based path extension. However, having a wider vector per vertex in Wiki-10M dataset directly affects the performance. In this case, PathWeaver achieves more speedup than the other two datasets mainly due to direction-guided selection.

To demonstrate the efficiency and scalability of PathWeaver, we evaluated its performance while increasing the number of GPUs from 1 to 4, at a recall of 95%. The results are shown in Fig. 9a. Compared to that of using a single GPU, using four GPUs achieves 2.47× more speedup when comparing the fastest cases of each GPU setting, which represents 62% scale efficiency. Compared to that of the baselines shown in Fig. 3 for the same dataset Deep-10M, the scale efficiency is improved by 17%.

Fig. 9b further evaluates the impact of pipelining-based path extension on multiple GPUs. Compared to PathWeaver using the sharding method of baseline (denoted as 'Naïve PathWeaver'), the advantage of pipelining-based path extension is maintained across various datasets and target recall values. This indicates that PathWeaver scales stably on various settings.

5.3 Performance Comparison on a Single GPU

While PathWeaver provides a significant speedup with multiple GPUs, its benefit remains in single-GPU settings. Except for pipelining-based path extension, PathWeaver can still benefit from ghost staging and direction-guided selection on a single-GPU setting. We plotted the performance of PathWeaver and other baselines in Fig. 10. In addition to the GPU-based methods, we additionally study HNSW [40], a baseline method implemented on a CPU.

Overall, PathWeaver achieves a speedup of 3.43× over CAGRA. This speedup is primarily attributed to reduced distance computations due to direction-guided selection and fewer iterations resulting from the application of ghost staging. Similar to the observations from multi-GPU experiments, Gist-1M results in the slowest throughput due to its larger vector dimension of 960. In all cases under evaluation, PathWeaver exhibited better QPS-recall trade-off compared to the baselines.

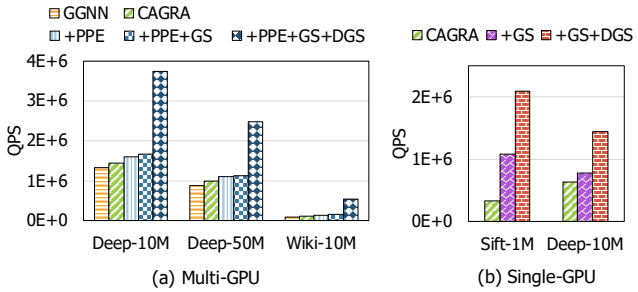


Figure 11: Ablation study of PathWeaver on (a) multi-GPU and (b) single-GPU settings. '+PPE', '+GS', and '+DGS' represent PathWeaver with pipelining-based path extension, ghost staging, and direction-guided selection, respectively.

5.4 Ablation Study

To demonstrate the impact of each scheme in PathWeaver, we conducted an ablation study using the Deep-10M, Deep-50M, and Sift-1M datasets on four GPUs, as well as Deep-10M and Sift-1M on a single GPU. In the results shown in Fig. 11, PPE, GS, and DGS represent pipelining-based path extension, ghost staging, and direction-guided selection, respectively.

For the ablation study on multiple GPUs, we used GGNN and CAGRA as baselines, and gradually applied PPE, GS, and DGS to analyze the effect of each, as shown in Fig. 11a. On top of the baselines, each component of PathWeaver adds consistent speedup across datasets with similar trends. In addition, Fig. 11b shows the ablation study on the single-GPU setting. Although pipelining-based path extension cannot be applied in this case, ghost staging provides higher speedup. This is because in the single-GPU setting, the entire search can be regarded as the first stage, which benefits from the ghost stage.

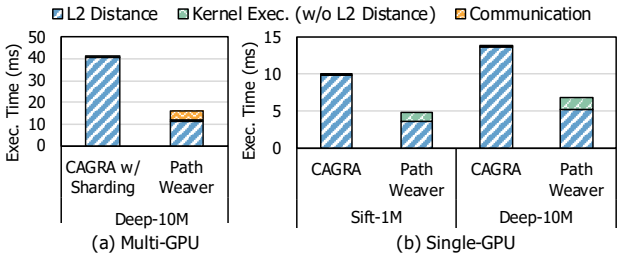


Figure 12: Search execution time breakdown on (a) multi-GPU and (b) single-GPU setting.

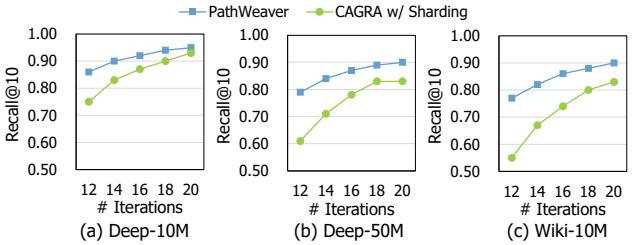


Figure 13: Accuracy comparison on different number of iterations.

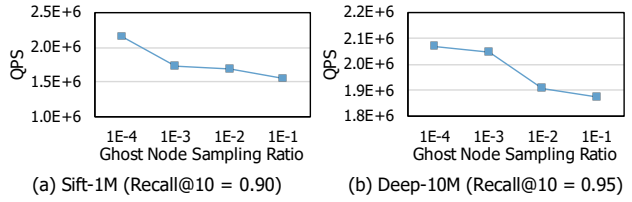


Figure 14: Sensitivity study on the relationship between the ghost node sampling ratio and QPS.

5.5 Execution Time Breakdown

To investigate how the execution time is spent in PathWeaver, we further broke down the execution time into three components: L2 distance calculation, rest of the kernel execution time, and inter-GPU communication time. The rest of the kernel includes random number generation, neighbor fetching, distance sorting, and hash table management for avoiding duplicate node visits. They are evaluated at 95% recall, with Deep-10M dataset for multi-GPU setting and an additional Sift-1M dataset for the single-GPU setting. To obtain the breakdown, we used `clock64()` function to separate the portion of L2 distance calculation within a kernel. For breakdowns of multi-GPU execution time, we measured the execution time with a certain portion of the code disabled, similar to the CPI stack [19] method.

The multi-GPU results are shown in Fig. 12a. We compare PathWeaver with extended CAGRA implementation for multi-GPUs (CAGRA w/ Sharding). As shown in the result, the L2 distance calculation is the dominating factor in both the CAGRA w/ Sharding and the fully optimized PathWeaver. In CAGRA w/ Sharding, no execution time is attributed to inter-GPU communication, since all searches are performed independently without any data exchange across GPUs. In PathWeaver, the communication time is incurred because of inter-stage data transfer of pipelining-based path extension. However, this results in fewer iterations required until convergence in subsequent stages, leading to a smaller L2 distance calculation. The rest of the kernel's portion slightly increases with optimizations in PathWeaver, due to the additional direction flag lookup process introduced by direction-guided selection. However, its portion is almost negligible in both CAGRA w/ Sharding and PathWeaver, making its impact minimal.

Fig. 12b depicts the breakdown on a single GPU. Because there is no communication, most of the time is spent on the L2 distance calculation. Thus, the speedups come from ghost staging and direction-guided selection. Instead of communication, the miscellaneous kernel execution time slightly increases in PathWeaver, which accounts for the ghost stages and the reduction of the dominant L2 distance calculation.

5.6 Detailed Analysis

In this subsection, we further investigate the effect of PathWeaver in terms of pipelining-based path extension, ghost staging, and direction-guided selection.

5.6.1 Pipelining-based Path Extension

To demonstrate the effect of pipelining-based path extension, we analyzed how the recall rate evolves by increasing the max number of search iterations in Fig. 13 on three datasets (Deep-10M, Deep-50M, and Wiki-10M). PathWeaver achieves high recall values with significantly fewer iterations. This is due to pipelining-based path extension, which enables each GPU to initiate the search process from a data point closer to the query using the search results from other GPUs. In contrast, the baseline requires substantially more iterations to reach comparable recall rates. For example, in the Deep-10M dataset, the baseline reaches a recall rate 0.90 with 18 iterations, while PathWeaver achieves this with only 14 iterations.

5.6.2 Ghost Staging

To evaluate the impact of the sampling ratio of ghost nodes on search performance, we conducted search operations on the Sift-1M and Deep-10M datasets using a single GPU, while varying the sampling ratio for building the ghost shard. As shown in Fig. 14, higher QPS was observed at lower sampling ratios of ghost nodes. For example, the QPS is 1.39× higher for sampling ratio 0.0001 compared to 0.1 in the Sift-1M dataset. We attribute this to the connections among the smaller set of ghost nodes, which facilitate larger iteration jumps

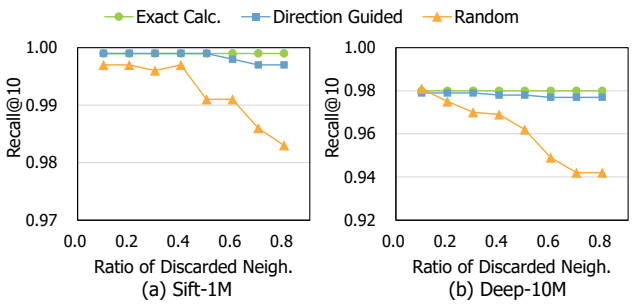


Figure 15: Comparison of neighbor selection strategies by varying the ratio of discarded neighbors.

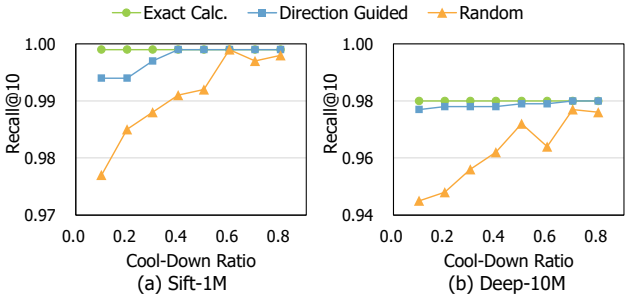


Figure 16: Comparison of neighbor selection strategies by varying the cool-down ratio of search iterations against the total iteration. A small cool-down ratio indicates that direction-guided selection or random selection is applied during the majority of the iterations.

within the original graph. The result indicates that a small-sized ghost shard is often sufficient, which provides another explanation for why ghost staging can bring speedup.

5.6.3 Direction-Guided Selection

To demonstrate the effectiveness of using direction bits as the neighbor discarding metric in direction-guided selection, we compared the performance of direction-guided selection against exact calculation (no discarding) and random neighbor discarding (randomly selecting neighbors).

Initially, the recall rate was evaluated for various discarded neighbor ratios while using a cool-down ratio of 0.5, as shown in Fig. 15. Compared to exact calculations, which achieve ideal recall, neighbor discarding in direction-guided selection results in a slight recall drop of at most 0.003, whereas random neighbor discarding causes a significant recall degradation of at most 0.038 for the Deep-10M dataset. Notably, PathWeaver maintains robust recall even with a discarding ratio of 0.7, in contrast to the significant recall decline observed with the random method.

Next, we analyze the recall rate by adjusting cool-down ratios with a fixed neighbor discarding ratio at 0.5, as shown

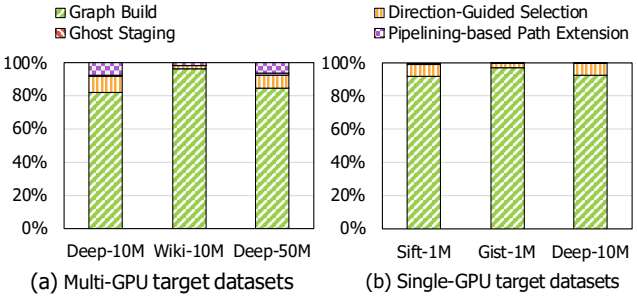


Figure 17: Graph build time overhead analysis.

in Fig. 16. Similar to the prior analysis, PathWeaver shows robust recall even when the cool-down ratio decreases, while the random method fails to maintain high recall.

For example, at a cool-down ratio of 0.3, neighbor discarding in direction-guided selection demonstrates only a minor recall drop of 0.002, compared to a significant degradation of 0.032 with the random discarding approach on the Deep-10M dataset. Both analyses demonstrate the effectiveness of using the direction of vectors when filtering out neighbors.

5.7 Graph Build Time Analysis

The preprocessing steps of PathWeaver, such as generating inter-shard connection edges for pipelining-based path extension, ghost nodes connections for ghost staging, and direction bit vectors for direction-guided selection, necessitate an overhead analysis to evaluate their impact. Therefore, we perform a breakdown analysis of the proximity graph build time, as shown in Fig. 17. Note that PathWeaver uses the graph build algorithm of CAGRA [44], represented as 'graph build'.

The results indicate that the overall overhead is less than 10% for datasets targeting a single GPU. The overhead of constructing ghost node connections was lower than generating direction bit vectors. This is because the latter requires creating direction bit vectors for all edges in the graph, while the former requires connecting a relatively smaller number of data points. For multi-GPU target datasets, the overhead was less than 4% for Wiki-10M, while it reached 15% for Deep-50M. This is because building the graph also involves many L2 distance calculations, whose overhead is proportional to the vector dimension, similar to the search kernel. Additionally, the overhead of constructing ghost shards remains negligible, with the overheads of generating inter-shard connections and direction bit vectors being comparable. The minimal overhead of creating inter-shard connections arises from leveraging the existing connections in the proximity graph. These connections provide information about which node in the next shard is closest to the target node in the current shard. Overall, for both cases of single- and multi-GPU settings, the additional graph build overhead from PathWeaver's designs is small.

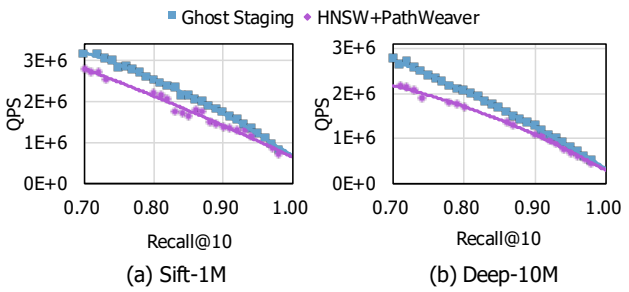


Figure 18: Comparing ghost staging and GPU-based HNSW.

6 Discussion

6.1 Comparing Ghost Staging and Existing Hierarchical Graph Approaches

Although ghost staging is a kind of hierarchical method, it differs from existing methods such as HNSW [40] and GGNN [25] in its objective and implementation. HNSW sequentially inserts each node into level n with exponentially decreasing probability as n increases, constructing a proximity graph at each layer. On the other hand, GGNN partitions the nodes, builds a graph within each partition, and finally merges them by selecting nodes for the next layer.

In contrast, ghost staging starts from an already-built proximity graph, and the extra layer is constructed by sampling vertices and connecting them. While existing hierarchical solutions aim to improve overall reachability and convexity by introducing hierarchy during graph construction, our approach uses the additional ghost stage solely to efficiently identify an entry node on the base proximity graph.

To investigate the difference, we conduct searching on the HNSW graph with our GPU-based search kernel and compare the results with those of ghost staging (direction-guided selection and pipelining-based path extension are disabled for fair comparison). As depicted in Fig. 18, ghost staging consistently achieves faster search speed compared to the HNSW graph even when it's searched with GPUs.

6.2 Managing Dynamic Updates

While PathWeaver currently targets static graphs, it is worthwhile to discuss dynamic updates to the proximity graphs [60]. Because PathWeaver is based on shards, any update would only involve modifying the affected shard. When rebuilding is needed, the cost mostly comes from the affected local graph, because the auxiliary data introduced by PathWeaver only accounts for a small portion as reported in Fig. 17.

For a small number of insertions, nodes can be added to one of the existing shards, followed by a rebuilding for the associated graph and the auxiliary data. If the number of insertions is small enough, the growth of the shard will not cause load imbalance. In such a scenario, the inter-shard edges can be incrementally updated because the small change in the local graph does not affect the similarity between existing vertices. For a large number of insertions, an additional shard can be created without modifying existing ones. For a small number of deletions, a deleted node may still act as a bridge between its neighbors. To preserve connectivity, a deletion flag can be used to logically remove the node. However, when a substantial portion of a shard is deleted, rebuilding the shard and its associated structures becomes beneficial.

6.3 Comparing Direction-Guided Selection with Static Graph Pruning Strategy

In ANNS, several approaches perform graph pruning [44, 50] in a static manner to reduce memory usage and improve query efficiency. This strategy reduces graph density by eliminating less important edges during index construction. While static pruning offers a consistent structure across all queries, it may not be optimal for query-specific behaviors.

On the contrary, direction-guided selection can be considered a dynamic pruning strategy. Instead of permanently discarding edges, it dynamically chooses a subset of neighbors during the search process, based on the direction to the query. One potential issue of its fixed top-n selection is that it might discard important candidates. Even though we did not observe a significant drop in recall in our experiments, one could instead prune based on the similarity criteria, as done in [66] for neural ranking. Such a method could preserve good candidates, at the potential cost of warp imbalances due to non-uniform pruning.

6.4 Overhead Analysis of Pipelining-based Path Extension

In this subsection, we quantify the communication overhead of pipelining-based path extension. First, the volume of inter-GPU communication is only $Q \times b_{idx}$, where Q is the number of queries and b_{idx} is the number of bytes per transmitted index. In contrast, the amount of GPU memory accesses scales with $I \times J \times Q \times v \times b_{elem}$, where I is the number of search iterations, J is the out-degree of each node, v is the vector dimension, and b_{elem} is the number of bytes per vector element.

Although inter-GPU channels (e.g., NVLink) are typically over 10× slower than GPU's memory bandwidth, the dominant cost still lies in the GPU's memory term, as the product $J \times v$ often exceeds 10^4. As a result, the reduction in I achieved by pipelining-based path extension has a much greater impact on total latency than the relatively small inter-GPU communication cost.

7 Related Works

7.1 Graph-based ANNS Solutions

CPU-based solutions. A range of CPU-based solutions has been proposed to enhance the construction and traversal of proximity graphs for approximate nearest neighbor (ANN) search. Key studies, such as [20, 21, 40, 41, 54], focus on designing efficient graph structures that are leveraged by the beam search algorithm to achieve scalable and effective ANN retrieval. To improve search efficiency, several methods incorporate further optimizations [9, 12, 16, 22, 42, 43, 61, 64]. For example, subgraph sampling and edge pruning techniques are utilized in [64] to reduce traversal overhead, while [16] explores reordering graph structures to enhance cache efficiency. Additionally, [12] introduces a novel distance function approximation that accelerates the search process, demonstrating significant performance gains in graph-based ANNS.

GPU-based solutions. SONG [65] represents the first graph-based ANN search solution for GPUs, leveraging GPU parallelism to address the bottleneck of computationally expensive distance calculations while optimizing data structures through various approaches. GANNS [62] and GGNN [25] build upon this by focusing on GPU-friendly implementations that efficiently utilize shared memory for maintaining data structures and parallelizing their operations. CAGRA [44] enhances graph construction and search processes on GPUs, employing techniques such as warp splitting and forgettable hash to achieve high throughput. On the other hand, PathWeaver moves beyond the sole reliance on proximity graphs during the search process. By integrating information from adjacent shards and considering query directionality, it achieves enhancements in both speed and accuracy.

Other platforms. Graph-based ANNS solutions have also been explored on alternative platforms to enhance performance [30, 36, 51, 57, 59, 63, 67]. Pyramid [67] introduces a hierarchical near-memory-computing (NMC) architecture specifically designed for efficient graph-based ANNS. Computational storage devices (CSD) are leveraged in works such as [36, 51], where software-hardware co-design accelerates search algorithms by integrating computation closer to data storage. DF-GAS [63] proposes a distributed FPGA-as-a-Service architecture, enabling parallel search operations across both full graphs and subgraphs. CXL-ANNS [30] employs memory disaggregation from the host via CXL to optimize graph-based ANNS performance.

7.2 Scaling ANNS to Large Datasets

For ANNS on large datasets, various approaches focus on compressing data vectors. Quantization-based methods [23, 26, 32, 33] achieve this by clustering nodes within a graph and replacing them with centroid-based representations. Similarly, SONG [65] utilizes 1-bit random projection to enable large datasets to fit within GPU memory.

Another prominent approach to handling large datasets involves hierarchical or hybrid architectures. Methods such as [24, 29, 45, 47, 49, 50] utilize storage devices to store large datasets, whereas BANG [53] combines CPU and GPU resources by maintaining the large graph index in CPU memory and leveraging GPU acceleration. FusionANNS [52] employs a combination of CPU-GPU collaboration and SSDs to optimize performance. These approaches enable cost-efficient processing of large datasets, though the achievable speedup remains inherently limited by the constraints of single-machine architectures.

Sharding datasets to fit into device memory and utilizing multiple devices has been widely investigated. Methods such as [14, 25, 31, 36] employ a multi-device strategy by partitioning datasets across devices, processing queries independently on each, and aggregating the results on the host. While these approaches enable distributed processing, achieving substantial and consistent speed-up as the number of devices increases often remains challenging. In contrast, our pipelining-based path extension achieves a proportional speedup with the number of machines used, effectively addressing the scalability limitations observed in previous methods.

8 Conclusion

We propose PathWeaver, a multi-GPU framework for graph-based ANNS that provides scalable and efficient performance on large datasets. PathWeaver addresses the main bottlenecks of graph-based ANNS through three key innovations: GPU-to-GPU communication to reduce unnecessary search iterations, a representative dataset search to refine starting data points, and data point filtering to minimize unnecessary memory access and distance calculation. Our evaluation results demonstrate that PathWeaver outperforms both CPU- and GPU-based solutions, achieving higher performance while maintaining accuracy.

Availability

We have open-sourced the code of PathWeaver and made it publicly available. The GitHub repository for PathWeaver can be found at `https://github.com/AIS-SNU/PathWeaver`.

Acknowledgement

This work was partially supported by National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (2022R1C1C1011307), and Institute of Information & communications Technology Planning & Evaluation (IITP) (RS-2024-00395134, RS-2024-00347394, RS-2023-00256081, RS-2021II211343). Jinho Lee is the corresponding author.

References

[1] Billion-Scale Approximate Nearest Neighbor Search Challenge: NeurIPS'21 competition track. https://big-ann-benchmarks.com/neurips21.html.

[2] Chroma. https://trychroma.com/.

[3] MongoDB. https://www.mongodb.com/.

[4] NVIDIA NVLink Bridge. https://www.nvidia.com/en-us/products/workstations/nvlink-bridges/.

[5] Pinecone. https://www.pinecone.io/.

[6] Weaviate. https://weaviate.io/.

[7] Wiki-all Dataset. https://docs.rapids.ai/api/cuvs/stable/cuvs_bench/wiki_all_dataset/.

[8] L. Amsaleg and H. Jégou. Datasets for approximate nearest neighbor search. http://corpus-texmex.irisa.fr/.

[9] Fabien André, Anne-Marie Kermarrec, and Nicolas Le Scouarnec. Accelerated Nearest Neighbor Search with Quick ADC. In *International Conference on Multimedia Retrieval (ICMR)*, 2017.

[10] Artem Babenko and Victor Lempitsky. Efficient indexing of billion-scale datasets of deep descriptors. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[11] Nandan Banerjee, Ryan C. Connolly, Dimitri Lisin, Jimmy Briggs, and Mario E. Munich. View management for lifelong visual maps. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2019.

[12] Patrick Chen, Wei-Cheng Chang, Jyun-Yu Jiang, Hsiang-Fu Yu, Inderjit Dhillon, and Cho-Jui Hsieh. Finger: Fast inference for graph-based approximate nearest neighbor search. In *ACM Web Conference (WWW)*, 2023.

[13] Rihan Chen, Bin Liu, Han Zhu, Yaoxuan Wang, Qi Li, Buting Ma, Qingbo Hua, Jun Jiang, Yunlong Xu, Hongbo Deng, and Bo Zheng. Approximate Nearest Neighbor Search under Neural Similarity Metric for Large-Scale Recommendation. In *ACM International Conference on Information & Knowledge Management (CIKM)*, 2022.

[14] Wei Chen, Jincai Chen, Fuhao Zou, Yuan-Fang Li, Ping Lu, Qiang Wang, and Wei Zhao. Vector and line quantization for billion-scale similarity search on GPUs. *Future Generation Computer Systems*, 2019.

[15] Kenneth L. Clarkson. An algorithm for approximate closest-point queries. In *Symposium on Computational Geometry (SCG)*, 1994.

[16] Benjamin Coleman, Santiago Segarra, Alex Smola, and Anshumali Shrivastava. Graph reordering for cache-efficient near neighbor search. In *International Conference on Neural Information Processing Systems (NIPS)*, 2022.

[17] Scott Deerwester, Susan T Dumais, George W Furnas, Thomas K Landauer, and Richard Harshman. Indexing by latent semantic analysis. *Journal of the American society for information science*, 1990.

[18] Matthijs Douze, Alexandr Guzhva, Chengqi Deng, Jeff Johnson, Gergely Szilvasy, Pierre-Emmanuel Mazaré, Maria Lomeli, Lucas Hosseini, and Hervé Jégou. The Faiss library. *arXiv:2401.08281*, 2024.

[19] P. G. Emma. Understanding Some Simple Processor-performance Limits. *IBM Journal of Research and Development*, 1997.

[20] Cong Fu and Deng Cai. Efanna : An extremely fast approximate nearest neighbor search algorithm based on knn graph. *arXiv:1609.07228*, 2016.

[21] Cong Fu, Chao Xiang, Changxu Wang, and Deng Cai. Fast approximate nearest neighbor search with the navigating spreading-out graph. *Proceedings of the VLDB Endowment (VLDB)*, 2019.

[22] Jianyang Gao and Cheng Long. High-Dimensional Approximate Nearest Neighbor Search: with Reliable and Efficient Distance Comparison Operations. *Proceedings of the ACM on Management of Data (PACMMOD)*, 2023.

[23] Tiezheng Ge, Kaiming He, Qifa Ke, and Jian Sun. Optimized Product Quantization for Approximate Nearest Neighbor Search. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2013.

[24] Siddharth Gollapudi, Neel Karia, Varun Sivashankar, Ravishankar Krishnaswamy, Nikit Begwani, Swapnil Raz, Yiyong Lin, Yin Zhang, Neelam Mahapatro, Premkumar Srinivasan, Amit Singh, and Harsha Vardhan Simhadri. Filtered-DiskANN: Graph Algorithms for Approximate Nearest Neighbor Search with Filters. In *ACM Web Conference (WWW)*, 2023.

[25] Fabian Groh, Lukas Ruppert, Patrick Wieschollek, and Hendrik P. A. Lensch. GGNN: Graph-Based GPU Nearest Neighbor Search. *IEEE Transactions on Big Data*, 2023.

[26] Ruiqi Guo, Philip Sun, Erik Lindgren, Quan Geng, David Simcha, Felix Chern, and Sanjiv Kumar. Accelerating large-scale inference with anisotropic vector quantization. In *International Conference on Machine Learning (ICML)*, 2020.

[27] Qiang Huang, Jianlin Feng, Yikai Zhang, Qiong Fang, and Wilfred Ng. Query-aware locality-sensitive hashing for approximate nearest neighbor search. *Proceedings of the VLDB Endowment (VLDB)*, 2015.

[28] Piotr Indyk and Rajeev Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. In *ACM Symposium on Theory of Computing (STOC)*, 1998.

[29] Shikhar Jaiswal, Ravishankar Krishnaswamy, Ankit Garg, Harsha Vardhan Simhadri, and Sheshansh Agrawal. OOD-DiskANN: Efficient and Scalable Graph ANNS for Out-of-Distribution Queries. *arXiv:2211.12850*, 2022.

[30] Junhyeok Jang, Hanjin Choi, Hanyeoreum Bae, Seungjun Lee, Miryeong Kwon, and Myoungsoo Jung. CXL-ANNS: Software-Hardware Collaborative Memory Disaggregation and Computation for Billion-Scale Approximate Nearest Neighbor Search. In *USENIX Annual Technical Conference (ATC)*, 2023.

[31] Jeff Johnson, Matthijs Douze, and Hervé Jégou. Billion-Scale Similarity Search with GPUs. *IEEE Transactions on Big Data*, 2021.

[32] Hervé Jégou, Matthijs Douze, and Cordelia Schmid. Product quantization for nearest neighbor search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2011.

[33] Yannis Kalantidis and Yannis Avrithis. Locally Optimized Product Quantization for Approximate Nearest Neighbor Search. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2014.

[34] Tanaka Kanji, Chokushi Yuuto, and Ando Masatoshi. Mining visual phrases for long-term visual SLAM. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2014.

[35] Urvashi Khandelwal, Omer Levy, Dan Jurafsky, Luke Zettlemoyer, and Mike Lewis. Generalization through Memorization: Nearest Neighbor Language Models. In *International Conference on Learning Representations (ICLR)*, 2020.

[36] Ji-Hoon Kim, Yeo-Reum Park, Jaeyoung Do, Soo-Young Ji, and Joo-Young Kim. Accelerating large-scale graph-based nearest neighbor search on a computational storage platform. *IEEE Transactions on Computers*, 2023.

[37] Conglong Li, Minjia Zhang, David G. Andersen, and Yuxiong He. Improving Approximate Nearest Neighbor Search through Learned Adaptive Early Termination. In *International Conference on Management of Data (SIGMOD)*, 2020.

[38] Wen Li, Ying Zhang, Yifang Sun, Wei Wang, Mingjie Li, Wenjie Zhang, and Xuemin Lin. Approximate Nearest Neighbor Search on High Dimensional Data — Experiments, Analyses, and Improvement . *IEEE Transactions on Knowledge & Data Engineering*, 2020.

[39] Ting Liu, Charles Rosenberg, and Henry A. Rowley. Clustering Billions of Images with Large Scale Nearest Neighbor Search. In *IEEE Workshop on Applications of Computer Vision (WACV)*, 2007.

[40] Yu A. Malkov and D. A. Yashunin. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2020.

[41] Yury Malkov, Alexander Ponomarenko, Andrey Logvinov, and Vladimir Krylov. Approximate nearest neighbor algorithm based on navigable small world graphs. *Information Systems*, 2014.

[42] Magdalen Dobson Manohar, Zheqi Shen, Guy Blelloch, Laxman Dhulipala, Yan Gu, Harsha Vardhan Simhadri, and Yihan Sun. ParlayANN: Scalable and Deterministic Parallel Graph-Based Approximate Nearest Neighbor Search Algorithms. In *ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2024.

[43] Sameer A. Nene and Shree K. Nayar. A Simple Algorithm for Nearest Neighbor Search in High Dimensions. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 1997.

[44] Hiroyuki Ootomo, Akira Naruse, Corey Nolet, Ray Wang, Tamas Feher, and Yong Wang. CAGRA: Highly Parallel Graph Construction and Approximate Nearest Neighbor Search for GPUs. In *IEEE International Conference on Data Engineering (ICDE)*, 2024.

[45] Yu Pan, Jianxin Sun, and Hongfeng Yu. LM-DiskANN: Low Memory Footprint in Disk-Native Dynamic Graph-Based ANN Indexing. In *IEEE International Conference on Big Data (BigData)*, 2023.

[46] Hadi Pouransari, Zhucheng Tu, and Oncel Tuzel. Least squares binary quantization of neural networks. In *IEEE/CVF conference on computer vision and pattern recognition workshops (CVPRW)*, 2020.

[47] Jie Ren, Minjia Zhang, and Dong Li. HM-ANN: efficient billion-point nearest neighbor search on heterogeneous memory. In *International Conference on Neural Information Processing Systems (NeurIPS)*, 2020.

[48] Chanop Silpa-Anan and Richard Hartley. Optimised KD-trees for fast image descriptor matching. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2008.

[49] Aditi Singh, Suhas Jayaram Subramanya, Ravishankar Krishnaswamy, and Harsha Vardhan Simhadri. FreshDiskANN: A Fast and Accurate Graph-Based ANN Index for Streaming Similarity Search. *arXiv:2105.09613*, 2021.

[50] Suhas Jayaram Subramanya, Devvrit, Rohan Kadekodi, Ravishankar Krishaswamy, and Harsha Vardhan Simhadri. DiskANN: fast accurate billion-point nearest neighbor search on a single node. In *International Conference on Neural Information Processing Systems (NeurIPS)*, 2019.

[51] Bing Tian, Haikun Liu, Zhuohui Duan, Xiaofei Liao, Hai Jin, and Yu Zhang. Scalable billion-point approximate nearest neighbor search using SmartSSDs. In *USENIX Annual Technical Conference (ATC)*, 2024.

[52] Bing Tian, Haikun Liu, Yuhang Tang, Shihai Xiao, Zhuohui Duan, Xiaofei Liao, Xuecang Zhang, Junhua Zhu, and Yu Zhang. FusionANNS: An Efficient CPU/GPU Cooperative Processing Architecture for Billion-scale Approximate Nearest Neighbor Search. *arXiv:2409.16576*, 2024.

[53] Karthik V., Saim Khan, Somesh Singh, Harsha Vardhan Simhadri, and Jyothi Vedurada. BANG: Billion-Scale Approximate Nearest Neighbor Search using a Single GPU. *arXiv:2401.11324*, 2024.

[54] Javier Vargas Muñoz, Marcos A. Gonçalves, Zanoni Dias, and Ricardo da S. Torres. Hierarchical clustering-based graphs for large scale approximate nearest neighbor search. *Pattern Recognition*, 2019.

[55] Jianguo Wang, Xiaomeng Yi, Rentong Guo, Hai Jin, Peng Xu, Shengjun Li, Xiangyu Wang, Xiangzhou Guo, Chengming Li, Xiaohai Xu, et al. Milvus: A Purpose-Built Vector Data Management System. In *International Conference on Management of Data (SIGMOD)*, 2021.

[56] Mengzhao Wang, Xiaoliang Xu, Qiang Yue, and Yuxiang Wang. A comprehensive survey and experimental comparison of graph-based approximate nearest neighbor search. *Proceedings of the VLDB Endowment (VLDB)*, 2021.

[57] Yitu Wang, Shiyu Li, Qilin Zheng, Linghao Song, Zongwang Li, Andrew Chang, Hai "Helen" Li, and Yiran Chen. NDSEARCH: Accelerating Graph-Traversal-Based Approximate Nearest Neighbor Search through Near Data Processing. In *ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*, 2024.

[58] Frank F. Xu, Uri Alon, and Graham Neubig. Why do nearest neighbor language models work? In *International Conference on Machine Learning (ICML)*, 2023.

[59] Weihong Xu, Junwei Chen, Po-Kai Hsu, Jaeyoung Kang, Minxuan Zhou, Sumukh Pinge, Shimeng Yu, and Tajana Rosing. Proxima: Near-storage Acceleration for Graph-based Approximate Nearest Neighbor Search in 3D NAND. *arXiv:2312.04257*, 2023.

[60] Zhaozhuo Xu, Weijie Zhao, Shulong Tan, Zhixin Zhou, and Ping Li. Proximity graph maintenance for fast online nearest neighbor search. *arXiv:2206.10839*, 2022.

[61] Shangdi Yu, Joshua Engels, Yihao Huang, and Julian Shun. PECANN: Parallel Efficient Clustering with Graph-Based Approximate Nearest Neighbor Search. *arXiv:2312.03940*, 2023.

[62] Yuanhang Yu, Dong Wen, Ying Zhang, Lu Qin, Wenjie Zhang, and Xuemin Lin. Gpu-accelerated proximity graph approximate nearest neighbor search and construction. In *IEEE International Conference on Data Engineering (ICDE)*, 2022.

[63] Shulin Zeng, Zhenhua Zhu, Jun Liu, Haoyu Zhang, Guohao Dai, Zixuan Zhou, Shuangchen Li, Xuefei Ning, Yuan Xie, Huazhong Yang, and Yu Wang. DF-GAS: a Distributed FPGA-as-a-Service Architecture towards Billion-Scale Graph-based Approximate Nearest Neighbor Search. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2023.

[64] Minjia Zhang, Wenhan Wang, and Yuxiong He. Grasp: Optimizing graph-based nearest neighbor search with subgraph sampling and pruning. In *ACM International Conference on Web Search and Data Mining (WSDM)*, 2022.

[65] Weijie Zhao, Shulong Tan, and Ping Li. SONG: Approximate Nearest Neighbor Search on GPU. In *IEEE International Conference on Data Engineering (ICDE)*, 2020.

[66] Weijie Zhao, Shulong Tan, and Ping Li. Guitar: Gradient pruning toward fast neural ranking. In *International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR)*, 2024.

[67] Zhenhua Zhu, Jun Liu, Guohao Dai, Shulin Zeng, Bing Li, Huazhong Yang, and Yu Wang. Processing-in-hierarchical-memory architecture for billion-scale approximate nearest neighbor search. In *ACM/IEEE Design Automation Conference (DAC)*, 2023.

Universal Checkpointing: A Flexible and Efficient Distributed Checkpointing System for Large-Scale DNN Training with Reconfigurable Parallelism

Xinyu Lian
UIUC

Sam Ade Jacobs
Microsoft

Lev Kurilenko
Microsoft

Masahiro Tanaka
Microsoft

Stas Bekman
Snowflake

Olatunji Ruwase
Microsoft

Minjia Zhang
UIUC

Abstract

Deep neural network (DNN) training continues to scale rapidly in terms of model size, data volume, and sequence length, to the point where multiple machines are required to fit large models for training. Different distributed and parallel training strategies have been developed to support large-scale DNN training by partitioning the training state across GPUs. However, existing DNN training systems provide very limited support for reconfiguring parallelism strategies in the middle of the training via checkpointing. This limitation arises because distributed checkpoints are tightly coupled to specific model parallelism and hardware configurations, preventing large-scale training jobs from efficiently adapting to hardware failures or resource elasticity.

This paper presents *Universal Checkpointing* (UCP), a novel checkpointing system that enables flexible and efficient DNN training with reconfigurable parallelism. UCP overcomes challenges in existing systems by decoupling checkpoint structure from parallel training strategies and hardware configurations. In addition, we present a pattern-based reconfiguration pipeline that enables automatic, flexible, and efficient mapping of checkpoint state to various parallelism strategies. Evaluation on a range of DNN models, including state-of-the-art dense and sparse LLMs, shows that UCP enables reconfiguration for a broader set of widely used parallelism strategies than existing solutions while adding negligible reconfiguration cost. UCP has been successfully employed in real LLM training workloads, greatly enhancing their flexibility and resilience to dynamic hardware environments.

1 Introduction

The emergence of Large Language Models (LLMs) [6] has opened up new opportunities across various fields. In comparison to traditional methods, LLMs such as ChatGPT [42] and GPT-4 [1] have exhibited unique capabilities such as instruction following, commonsense reasoning, and few-shot generalization [26]. These abilities have been achieved primarily through unsupervised learning on massive model sizes, ranging from billions to even trillion-scale parameters [3, 9, 22, 54, 62]). However, as DNN continues to scale in model size, data size, and sequence length, multiple machines are required to fit the model for training. This has been made possible primarily through advanced distributed systems and parallelism technologies, where a model is partitioned across multiple devices and leverage the aggregated memory and compute capability from many GPU devices to train large-scale models [18, 21, 28, 37, 38, 46, 48, 51].

While demonstrating excellent training efficiency and scalability improvement, training modern DNNs/LLMs can still span several days or even months on highly optimized GPU clusters [2, 53]. For example, GPT-4 is reportedly trained on ~25,000 NVidia A100 GPUs over 90-100 days [19]. Given that training happens across many GPUs over an extended period, there is a high likelihood that of interruptions because of hardware failures, software bugs, or capacity management issues. For instance, the 54-day training of LLaMA 3.1 [54] on 16,000 GPUs encountered 419 failures, with training failing on average every three hours. In such cases, the model either faces prolonged and unpredictable failure recovery time or must adapt to a new training environment, such as continuing training with a different number of GPUs or migrating from one training cluster to another. However, existing parallelism strategies are hard to adapt, which calls for innovations that enable *reconfigurable parallelism*, which allows a training job to flexibly reconfigure its parallelism strategies while maintaining the same training convergence and accuracy.

One promising solution to enable reconfigurable parallelism is through the checkpointing system. Intuitively, a model checkpoint contains a snapshot of the model state, including weights, optimizer states, and additional metadata that captures the training progress (e.g., training iterations). If a checkpoint can be saved from one type of parallelism strategy (e.g., ZeRO-style data parallelism [46]) and reloaded to resume training with a different strategy (e.g., 3D parallelism [38]), it becomes possible to reconfigure parallelism across different hardware configurations. This results in more resilient DNN training in the face of hardware failures and allows for more flexible parallelism choices to adapt to resource

elasticity. Popular DL frameworks, such as PyTorch [20], Megatron [41], and DeepSpeed [47], already have their checkpointing systems and some research work has also proposed efficient methods for checkpointing [14, 36, 60]. However, are they sufficient to provide such reconfigurable parallelism?

Challenges of reconfigurable parallelism. Unfortunately, existing checkpointing systems are limited in supporting reconfigurable parallelism for three main reasons. First, existing checkpointing mechanisms are highly coupled to specific parallelism strategies [20, 33, 36, 60]. As a result, different parallelism strategies create distributed checkpoint files in different structures, and a checkpoint for a specific parallelism strategy cannot be loaded for another strategy. For instance, PyTorch Distributed Elastic [11] only allows adjustment of data parallel degrees (i.e., the number of data parallel workers), and there is currently no system that supports reconfiguring a checkpoint from ZeRO-style data parallelism [46] to 3D parallelism [38] or reconfiguring hybrid parallelism for sparse Mixture-of-Experts models [23].

Second, there lacks an automatic reconfiguration pipeline for a wide range of commonly used parallelism strategies. Because existing checkpointing systems are tightly coupled with specific parallelism strategies, only a small subset of these strategies can be reconfigured through checkpoints. This is typically achieved by implementing dedicated converters [20, 33] or ad-hoc conversion scripts [34]. However, this approach requires significant engineering effort and system expertise while still being error-prone, because it requires navigating a large and complex set of parallelism strategies and carefully examining how each strategy maps to model checkpoints. This includes determining whether each tensor operator in a model has its data replicated or partitioned, identifying the axis along which an operator is partitioned, and understanding how these replicated parameters and partitioned parameters map to distributed checkpoint files.

Third, as model sizes continue to increase, the overhead associated with reconfiguring from one parallelism to another becomes non-negligible. Larger models often require more time for reconfiguration, which can lead to sub-optimal training performance if not optimized. Therefore, it is important to ensure that the reconfiguration process is efficient and scalable. While there has been work optimizing the checkpoint saving and loading overhead [14, 36, 59, 60], few studies have focused on optimizing the reconfiguration overhead of distributed checkpoints.

Our approach. To address the aforementioned challenges, we take a different approach by introducing *Universal Checkpointing* (UCP), a novel checkpointing system that enables large-scale DL training with reconfigurable parallelism. UCP allows users to flexibly reconfigure a significantly larger and more complicated set of parallelism strategies compared to prior checkpointing mechanisms, including widely used parallelism strategies such as DP [29], TP [48], PP [18], SP [21, 30], ZeRO [46], and their combinations such as 3D-Parallelism [38, 51] with variable GPU counts. UCP achieves this through several techniques. First, we introduce *atom checkpoint*, which is a new checkpoint structure that decouples the checkpoint file from any specific parallelism strategy and hardware configurations, while being flexible to adapt to a wide range of parallelism strategies. Second, we introduce a *pattern-based reconfiguration pipeline* that enables reconfigurable parallelism on top of atomic checkpoints via a well designed *pattern set* and *pattern-based reconfiguration operations*. We show that this pipeline allows automatically and flexibly reconfiguring complex parallelism strategies such as 3D parallelism and ZeRO-style data parallelism over state-of-the-art LLM architectures, including dense LLMs, and architectures that require hybrid parallelisms, such as sparse Mixture-of-Experts (MoE) [22], and models with irregular attention such as Grouped-Query Attention (GQA) [4]. Finally, we introduce efficient system optimizations to reduce the reconfiguration cost via *nested parallel reconfiguration*, *redundancy-bypassing loading*, and *lazy reconfiguration invocation*, which makes impact of UCP on the training negligible.

We conduct extensive evaluation on large-scale LLM models, including both dense GPT [48] models and sparse MoEs [22]. Our evaluation results show that UCP enables reconfiguration of a significantly larger and more complex set of parallelism strategies from arbitrary distributed checkpoint files, outperforming prior checkpointing mechanisms. Additionally, we show that UCP is efficient and scalable, which adds minimal reconfiguration overhead. To summarize, we make the following contributions:

- We formulate the problem of reconfigurable parallelism for large-scale distributed training and develop Universal Checkpointing (UCP), a novel checkpointing system that enables reconfigurable parallelism for a broad set of commonly used parallelism strategies.
- We introduce atomic checkpoint, a new checkpoint structure that decouples checkpoints from specific parallelism strategies and serves as a common representation for flexible reconfigurable parallelism.
- We design a pattern-based reconfiguration pipeline, which provides systematic and automated parallelism reconfiguration through a carefully designed pattern set and pattern-based reconfiguration operations.
- We introduce nested parallel reconfiguration and lazy reconfiguration invocation, which significantly reduce the reconfiguration cost. Compared to the sequential approach used in ad-hoc conversion scripts, our nested-parallel method achieves a 14-257x reduction in time cost for models ranging from 7B to 1T parameters. The reconfiguration cost for 1T model is within 3 minutes, which is less than 0.001% of the total training time.

- We conduct extensive evaluation on models with multi-billion parameters and demonstrate the effectiveness of UCP in terms of accuracy, coverage, and efficiency.

UCP has been verified through the end-to-end training of several real-world large-scale models, including BigScience BLOOM (176B) [52], Microsoft Phi-3.5-MoE (42B) [3], UCB SmileyLlama (8B) [8] and RUC YuLan-Mini (4.2B) [17]. This has greatly improved these models' resilience to hardware failures during training, reducing their failure waiting time. It also provides these models flexibility to choose a better-suited parallelism strategy when the training environment changes via reconfigurable parallelism. We have implemented and open-sourced UCP in DeepSpeed[1]. UCP also becomes readily usable through popular DL frameworks that integrate the library as a backend, such as HuggingFace [61] and PyTorch lightning [45].

2 Background

We provide a brief introduction to the background of parallelism strategies for distributed training. In particular, training DNNs efficiently at scale typically involves the following parallelism strategies.

Data Parallelism (DP). DP is one of the most commonly used parallelism strategies for scaling DNN training on multi-GPUs. It divides a mini-batch into micro-batches across data parallel workers but the model state is replicated. Each worker independently computes a micro-batch and calculates gradients, which are subsequently synchronized through all-reduce to get consistent model parameters throughout training. To eliminate the redundancy of replicated model states in DP, zero redundancy optimizer (ZeRO) was introduced to progressively eliminate model redundancy in optimizer states (ZeRO-1), gradients (ZeRO-2), and weights (ZeRO-3) [46].

Tensor Parallelism (TP). To train models that exceed device memory, TP partitions compute and memory-intensive tensor operations (e.g., *matmul*) along non-batch axes across multiple devices. It then lets each GPU compute the partitioned operation in parallel and uses communication collectives (e.g., *all-reduce*, *all-gather*) at the split and merge point of these partitioned operators to ensure consistent states [48].

Pipeline Parallelism (PP). Different from TP, PP divides model states into multiple stages on different GPUs [18, 37]. Meanwhile, a pipeline scheduler splits a mini-batch into micro-batches and streams the micro-batches through different pipeline stages. Two consecutive pipeline stages exchange intermediate data via point-to-point communication. Such data dependencies lead to pipeline bubbles, i.e, computation stalls. Multiple pipeline schedules such as 1F1B [37] and interleaved 1F1B [38] have been introduced to reduce pipeline bubbles to improve overall pipeline efficiency.

[1] https://github.com/deepspeedai/DeepSpeed

Sequence Parallelism (SP). Motivated by training long-context LLMs, researchers have also developed SP, which divides the model activations along the sequence input dimension and leverages techniques such as distributed attention mechanism to hold the large activation memory from training long sequence inputs [21, 30, 31].

These parallelism strategies are often combined to scale out modern DNN training, and a training system needs to orchestrate different parallelism strategies with the goal to achieve high training efficiency. For example, the state-of-the-art training systems such as Megatron-LM and DeepSpeed, combine these parallelism strategies for scaling Transformer-based LLMs, known as 3D/4D parallelism [24, 38, 51].

3 Problem Formulation

In this section, we formulate the problem of reconfigurable parallelism for distributed DL training. In particular, as input, we are given (1) a training task that has a DNN model Φ with a model state of $\Phi(t)$ (weights and optimizer states) at training iteration t, (2) a *Source* parallelism strategy P_{src} that decides how $\Phi(t)$ is partitioned and executed in parallel on N_{src} parallel workers (e.g., 64 NVidia A100 GPUs), (3) a *Target* parallelism strategy, on a target hardware configuration with N_{tgt} workers (e.g., 48 NVidia A100 GPUs), to which the model training needs to be reconfigured. In both cases, a parallelism strategy can be DP [29], TP [48], PP [18, 37], SP [30], ZeRO [46], or a composition of those strategies such as 3D/4D parallelism [38, 51]. The objective is to design a reconfiguration mechanism that transforms $\Phi(t)$ from the *Source* parallelism strategy P_{src} to the target strategy P_{tgt}.

4 Challenges and Opportunities

In this section, we introduce the challenges prior work faces to offer reconfigurable parallelism and opportunities for improvement, which guides the design in the subsequent section.

Model parallelism implementation-coupled checkpoint saving/loading. The distributed checkpoints are highly coupled to specific model parallelism strategy. For example, data parallelism (e.g., PyTorch DistributedDataParallel [29]), one of the most commonly used parallelism strategies, replicates model weights and optimizer states across GPUs, and therefore only one rank (e.g., rank 0) saves the entire model states in a checkpoint file. When resuming training, each data parallel worker loads the same checkpoint file before launching the training iterations. Other parallelism techniques, such as ZeRO [46] and 3D parallelism [38], shard model parameters and optimizer states across GPUs. Since the model states are partitioned across GPUs, each GPU separately saves and loads checkpoint files that contain only a fraction of model state it owns. Fig. 1 shows the structure of checkpoint files from ZeRO-3, which is very different from others.

Model parallelism-coupled distributed checkpoints lead to a relatively simple checkpointing saving/loading pipeline, because the checkpointing loading is simply the reverse of

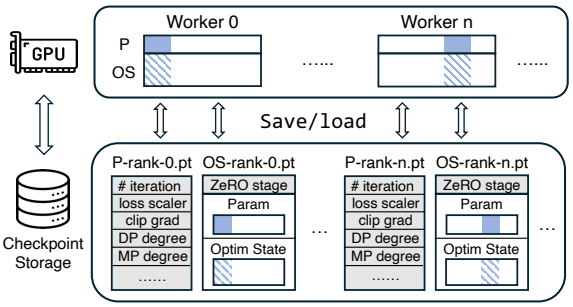


Figure 1: Existing distributed checkpoints are highly coupled to specific model parallelism strategy. Each worker in ZeRO-3 saves its own unique sharded model parameters (P) and optimizer states (OS). Reconfiguring ZeRO-3 requires altering each sharded parameter and optimizer states.

checkpoint saving process for each worker. This design also carries a performance advantage because distributed checkpoint saving requires to block all parallel workers, and having each parallel worker saves the model states it owns incurs no additional synchronization overhead and does not negatively impact the overall training speed. However, this coupled design makes it painful and error-prone to reconfigure the checkpoints to a different parallelism strategy, because developers need to hand-write conversion scripts to convert the checkpoint files from one parallelism to another. As a result, existing frameworks support reconfiguration of only a very limited set of parallelism strategies, e.g., changing the data parallel degree in distributed data parallel [10], or supports weight-only conversion of distributed checkpoints for inference-only evaluation (e.g., the conversion of optimizer states is not supported) [41]. As such, any node failure or resource reallocation in the training process will hobble the entire training process if there lacks support for a training job to reconfigure its parallelism on a new hardware environment.

Limited coverage of automatic reconfiguration pipeline for a large set of widely used parallelism strategies. Table 1 summarizes the state-of-the-art checkpointing systems for DL training. CheckFreq [36] is an efficient checkpointing system that overlaps the computation of training with checkpoint saving and adaptively tunes the checkpointing frequency at runtime based on I/O profiling. Gemini [60] introduces an in-memory checkpointing technique that checkpoints GPU state to local and remote CPUs and interleaves checkpointing IO with training computation to reduce the overhead, which impressively enables checkpointing on every training iteration. Both CheckFreq and Gemini accelerate checkpointing efficiency but do not provide support for reconfiguration of advanced model parallelism such as TP, PP, and ZeRO. Some concurrent work have proposed changing parallelism through checkpointing. For example, PyTorch Distributed Checkpointing (DCP) supports changing the parallelism degrees via loading-time resharding, e.g., after

	DP [29]	TP [38]	PP [37]	SP [30]	ZeRO [46]
CheckFreq [36]	✗	✗	✗	✗	✗
Gemini [60]	✓	✗	✗	✗	✗
DCP[2]	✓	✗	✗	✗	✓
MCP[3]	✓	▲	✓	✗	✗
UCP	✓	✓	✓	✓	✓

Table 1: Comparison of support for reconfiguring distributed checkpoints across various parallelism strategies. Triangle(▲) represents partial support with limitations.

all distributed checkpoint files are loaded to the CPU, each sharded parameter with a shape of $[X/n, Y]$ is concatenated to obtain the complete $[X, Y]$ parameter. The tensor is then resharded based on the current DP degree. Similarly, Megatron Dist Checkpointing (MCP) saves checkpoints with a local shape of $[PP*TP, X/PP, Y/TP]$. During loading, these checkpoints are concatenated to a global shape of $[1, X, Y]$ and then sharded based on the current PP and TP degrees. Both of these approaches are limited in that they only support a subset of parallelism strategies. Given that different parallelism strategies have different performance implications on various hardware, it is desirable to have reconfigurable parallelism that covers widely used parallelism strategies that are also easily extensible to different model architectures.

High reconfiguration overhead. Modern DNN sizes are often massive due to their large number of parameters, e.g., Transformers [26, 57] have increased by over 1000× in the past few years. Notable examples include PaLM [9] with 540 billion parameters and MT-NLG [51] with 530 billion parameters. The checkpoint size is approximately 12× the parameter count (4× for model parameters and 8× for optimizer states), thus checkpoint for these models reaches tens of TB in size. Given the huge checkpoint sizes, the reconfiguration time can be constrained by the bandwidth of the persistent storage [14, 60]. For example, it takes 42 minutes to checkpoint the model states of MT-NLG [51] to the remote persistent storage when the bandwidth is 20Gbps. Given the large and ever increasing model sizes, reconfiguration needs to have relatively low overhead in order to avoid becoming a major performance bottleneck.

5 Universal Checkpointing

5.1 UCP Design Overview

UCP is a distributed checkpointing system, specifically designed for handling large-scale DL training with reconfigurable parallelism. It allows flexible and efficient reconfiguration from any *Source* parallelism strategy to any *Target* parallelism strategy. Since training accuracy is the main re-

[2] https://pytorch.org/tutorials/recipes/distributed_checkpoint_recipe.html
[3] https://docs.nvidia.com/megatron-core/developer-guide/latest/api-guide/dist_checkpointing.html

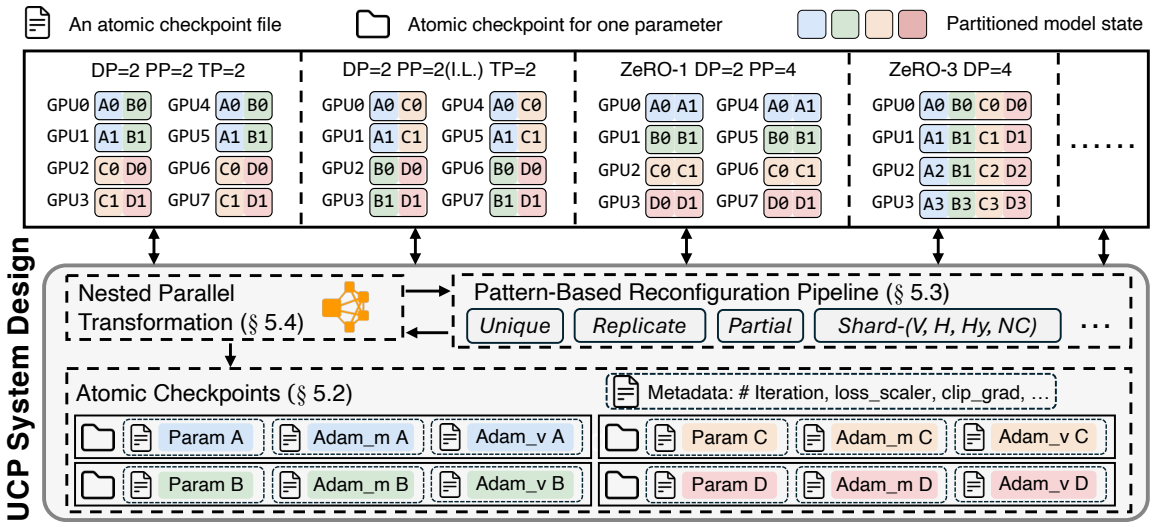


Figure 2: Overview of UCP system design. UCP enables flexible and efficient reconfiguration from any *Source* parallelism strategy, e.g., P_{src} (ZeRO-1 + PP) = (DP=2, PP=4) to any *Target* parallelism strategy, e.g., P_{tgt} (3D-Parallel) = (DP=2, TP=2, PP=2), via atomic checkpoints, pattern-based reconfiguration pipeline, and efficient optimizations such as nested parallel reconfiguration.

quirement, we focus on design choices that do not lead to training accuracy loss or related metric degradation. We first provide an overview of UCP and discuss each component in more details in subsequent sections.

Fig. 2 provides an overview of UCP. At a high level, UCP consists of three major components: atomic checkpoints, pattern-based reconfiguration pipeline, and nested parallel transformation. In § 4, we describe the challenge of parallelism-coupled checkpoints, which are cumbersome to reconfigure. We address this challenge by raising the level of abstraction, and decoupling the checkpoint file format from its parallelism strategy and hardware configurations through *atomic checkpoints* (§ 5.2). Atomic checkpoints enable radically simpler mechanism to achieve flexible reconfiguration, spanning the space of parallelism choices. UCP also provides a pattern-based reconfiguration pipeline to systematically and automatically reconfigure distributed checkpoints from a *Source* parallelism to a *Target* parallelism through atomic checkpoints (§ 5.3). UCP achieves this by carefully designing a parallelism pattern set and developing pattern-aware reconfiguration operations to automatically convert *Source* distributed checkpoints into atomic checkpoints and map atomic checkpoints to any *Target* parallelism strategy. UCP leverages *nested parallel reconfiguration*, *redundancy-bypassing loading* and *lazy reconfiguration invocation* to reduce the reconfiguration overhead such that reconfiguration only adds minimal overhead to the training process (§ 5.4).

5.2 Atomic Checkpoint

In existing DL distributed training systems, different parallelism strategy handles model states differently. For example, DP replicates and maintains a consistent view of model states during training across data parallel workers, so it is enough to have a single GPU for checkpointing. Other parallelism strategies, such as tensor-slicing parallelism or pipeline parallelism, partition model states along different axes and layers across GPUs. Therefore, each GPU creates a checkpoint file based on a snapshot of the partitioned model states that are in its local memory. Because of this highly parallelism-coupled checkpointing mechanism, practitioners need to hand-write conversion scripts to convert distributed checkpoints for different parallelism strategies, which incurs significant engineering cost and hard to maintain.

In UCP, we use *atomic checkpoints* to represent model states. An *atomic checkpoint* contains a consolidated view of the model states corresponding to a tensor operation, e.g., a merged state of fragmented states, if the tensor operation is partitioned over multiple parallel workers. Different model states (e.g., weights and optimizer states) have separate atomic checkpoint files, and all atomic checkpoint files corresponding to one parameter is considered as a conjunction of atomic checkpoints (e.g., a directory) to that parameter. In line with standard practices in mixed-precision training, UCP stores optimizer states in FP32 to maintain numerical stability and avoid loss of precision. While this increases checkpoint size, it ensures correctness during training. Without loss of generality, assuming the training uses the Adam optimizer [27], for each tensor operation, a conjunction of atomic checkpoints to its parameter has three atomic checkpoint files:

model.pt: A fp32 tensor represents parameter's weight.

adam_m.pt: A fp32 tensor represents first order moment.

adam_v.pt: A fp32 tensor represents second order moment.

The atomic checkpoint is much more fine-grained than most existing distributed checkpoints that have the entire local snapshot of individual parallel worker. An atomic checkpoint is also no longer coupled with any parallelism strategy or hardware configurations, e.g., it does not include any rank id, partitioning information, or additional data from specific sharding strategies such as padding for alignment. It simply maps each parameter in the model to a consolidated view of its weights and corresponding optimizer states.

The atomic checkpoint is sufficient to map to a wide range of parallelism algorithms, and it also enables flexible reconfiguration of parallelism through checkpoints. Critically, this representation is naturally data parallel when each data parallel worker needs to load a replica of the model states in a parameter-by-parameter basis. Also, as we identify partitioning patterns in many parallelism (§ 5.3.1), it becomes possible to define a set of primitives to automatically transform atomic checkpoints into specific parallelism strategies (§ 5.3.2). As such, UCP acts as a common interchange format between different distributed training techniques, and one does not need to implement individual converters from each *Source* to *Target* parallel config, significantly increasing the flexibility and coverage of reconfigurable parallelism.

Special consideration for data types. In practice, developers often adopt hardware-friendly training techniques such as mixed-precision training (MPT) [35], where both IEEE float16 weights and float32 weights and optimizer states are maintained to leverage the high computation throughput of hardware features such as TensorCore [32]. MPT affects the design of reconfigurable parallelism, as different states may have different data formats. Furthermore, dynamic mixed precision training saves checkpoints in one data format (e.g., float16) and switches to another one (e.g., bfloat16 or float32) at different training stages, which have different bit allocation policies between the mantissa and exponent [25]. Therefore, it is crucial to consider the data formats for checkpoints to accommodate these training techniques. In UCP, we keep all atomic checkpoints (weight/optimizer values) in float32, and add support to resume training in other data types, including float32, float16, and bfloat16, making it flexible to reconfigure parallelism with different data types.

5.3 Pattern-Based Reconfiguration Pipeline

Atomic checkpoints are decoupled from specific parallelism strategies. However, another important design choice remains unspecified: How to automatically convert distributed checkpoints *from* various P_{src} to atomic checkpoints and map atomic checkpoints to different P_{tgt} parallelism strategies? Most importantly, how to make sure that this automatic conversion process covers a wide range of widely used parallelism strategies as well as new parallelism strategies from state-of-the-art model architectures? The design choice here does not change the meaning of atomic checkpoints, but they are essential to ensure the utility of the resulting reconfiguration system. To provide answers, in this section, we develop a *pattern-based reconfiguration pipeline*.

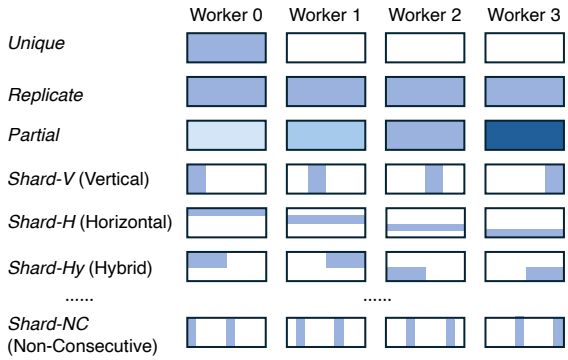


Figure 3: Illustration of patterns defined in UCP.

5.3.1 Designing the Pattern Set

To support flexible reconfiguration of parallelism based on atomic checkpoints, we seek an approach that can automatically map distributed checkpoints into atomic checkpoints. To achieve that, we look into patterns inside distributed checkpoint files so we can perform pattern-based transformation. The key consideration here is how to design and select the patterns. Good patterns should be able to have two key properties: coverage and flexibility. The coverage is desirable because it enables mapping between distributed checkpoints from a wide range of parallelism strategies and model architectures to atomic checkpoints. The flexibility facilitates subsequent parallelism reconfiguration operations. Given these two key properties, we propose the following pattern set.

Fig. 3 illustrates the pre-defined patterns in UCP. *Unique* means that a tensor is uniquely associated with a distributed checkpoint, which is commonly seen from checkpoints generated through inter-op parallelism such as pipeline parallelism. *Replicate* means one parameter will be replicated across multiple distributed checkpoints. For example, even though *matmuls* are sharded across GPUs in TP, other parameters such as LayerNorm and the Bias terms are replicated across GPUs because they are not compute and memory intensive. As a result, those replicated parameters will cause redundancy if we converted all those replicated parameters to atomic checkpoints. *Partial* indicates a parameter that is updated independently across GPUs, which corresponds to models with asynchronous training. More complicated patterns exist in intra-op parallelism strategies, which partition parameters along certain dimensions, e.g., *Shard-V* for column-wise sharding and *Shard-H* for row-wise sharding in tensor parallelism, *Shard-Hy*, which shards along multiple dimensions of the model states, and *Shard-NC*, which shards model states non-consecutively. We show later that the set of patterns we choose enables conversion of complicated parallelism strate-

gies such as ZeRO-3 [46] and 3D parallelism [38] while being flexible enough to also support new model architectures such as MoE [23] and GQA [5]. Note that although the set of patterns covers many existing parallelism strategies, this approach is general and extensible to support new parallelism patterns, not just the ones considered in the paper.

5.3.2 Pattern-Aware Reconfiguration Operations

Given the pattern set, UCP extracts the pattern information from distributed checkpoints and launches pattern-aware reconfiguration operations. In particular, Table 2 lists the main reconfiguration operations in UCP, including *Extract*, *Union*, *StripPad*, *UcpInfo*, and *Load*. For different patterns, each operation performs pattern-aware transformation, and Algorithm 1 demonstrates how the *Extract* and *Union* consolidate different parameter fragments based on their patterns. We use several concrete examples to illustrate how pattern-aware reconfiguration operations in UCP support complex parallelism strategies and different model architectures.

Table 2: Pattern-aware reconfiguration operations in UCP.

Operator	Definition
Extract	Takes distributed checkpoint files from P_{src} as input and returns a set of parameter fragments contained in that checkpoint file.
Union	A pattern-specific union is called on parameter fragments to obtain consolidated parameters.
StripPad	Strips pattern-specific padding from a consolidated parameter.
UcpInfo	Generates the metadata (e.g., shape and location information to a given rank) associated with P_{tgt} for each atomic checkpoint.
Save	Saves the consolidated model states for each parameter as atomic checkpoint files.
Load	Loads atomic checkpoints to each rank based on the *UcpInfo* of P_{tgt}. UCP loads atomic checkpoints in a layer-by-layer fashion to prevent model states from exceeding memory limits.

ZeRO Stage 3. The process of applying UCP's reconfiguration pipeline to ZeRO-3 checkpoints is illustrated in Fig. 4. If ZeRO-3 is the P_{src}, each DP rank persists the sharded parameters and optimizer states it owns to a checkpoint file, leading to N distributed checkpoint files in total when the DP degree is N. UCP then identifies parallelism patterns of ZeRO-3: ZeRO-3 flattens each tensor into a 1D tensor and shards the 1D tensor equally across different data parallel ranks. UCP identifies those tensors in ZeRO-3 distributed checkpoint files as the *Shard-V* pattern. Based on the pattern, UCP runs *Extract* and *Union* on fragmented parameters to create atomic checkpoints, which contain a consolidated view of parameters and optimizer states. One complexity of ZeRO-3 is that it adds padding to make sure the 1D tensor can be evenly divided across N DP workers for alignments. For example, for a parameter with a shape [1024], if the original DP degree is 3, then ZeRO-3 pads the parameter to [1026], such that each worker owns a fragment of [342]. UCP uses *StripPad* to remove the padding and uses *Save* to save the resulting atomic checkpoints to persistent stage.

On the other hand, if ZeRO-3 is the P_{tgt}, each DP worker calculates its new partition metadata via *UcpInfo* and then loads atomic checkpoints to each rank sequentially, following the layer order and with alignment padding added for high performance. Using the previous example, assuming $N_{tgt} = 2$, UCP will resume training with each rank having the correct parameter shape [512] instead of [513]. Once all the partitioned states are loaded into a GPU, e.g., in the flatten memory attribute *fp32_partitioned_groups_flat* of ZeRO-3, the updated attribute is then broadcast to other necessary attributes, such as *fp16_partitioned_groups_flat* for MPT. Through this process, UCP flexibly reconfigures parallelism with ZeRO-3 either as P_{src} or P_{tgt} under various DP degrees.

Algorithm 1 Pattern-Aware Reconfiguration

```
▷ Extract
for distributed checkpoint ckpt in storage do in parallel
    for param_name p, tensor t in ckpt do
        ToShuffler(p, t)
▷ Union
param_list = FromShuffler()
for param p in param_list do in parallel
    {T_1, T_2, ..., T_n} ← all tensors name matches p
    Switch p
        case PatternMatch(Replicate, p) then
            ucp_p = T_1
        case PatternMatch (Partial, p) then
            ucp_p = Sum(T_1, T_2, ..., T_n) / n
        case PatternMatch(Shard, p) then
            ucp_p = Concat(T_1, T_2, ..., T_n)
        case PatternMatch(Unique, p) then
            ucp_p = T
    if hasPadding(p) then
        ucp_p = StripPad(ucp_p)
    Save(ucp_p)
```

3D parallelism. The reconfiguration process with 3D parallelism either as P_{src} or P_{tgt} is similar to ZeRO-3. However, since 3D parallelism consists of various parallelism strategies, it has its own set of complexities. Similar to ZeRO-3, when saving distributed checkpoints for 3D parallelism, each GPU only saves a slice of the model state it owns. However, the parameter pattern of 3D parallelism is much more complicated: parameters can have *Replicate* (e.g., *LayerNorm*), *Shard-V* and *Shard-H* (e.g., *matmul*), *Partial* (e.g., Alibi Embedding [44]) with TP degree > 1, and *Replicate* (e.g., Tied-Embedding) pattern and *Unique* (majority) with PP degree > 1. UCP significantly lifts the burden of manually converting distributed checkpoints from 3D parallelism through pattern

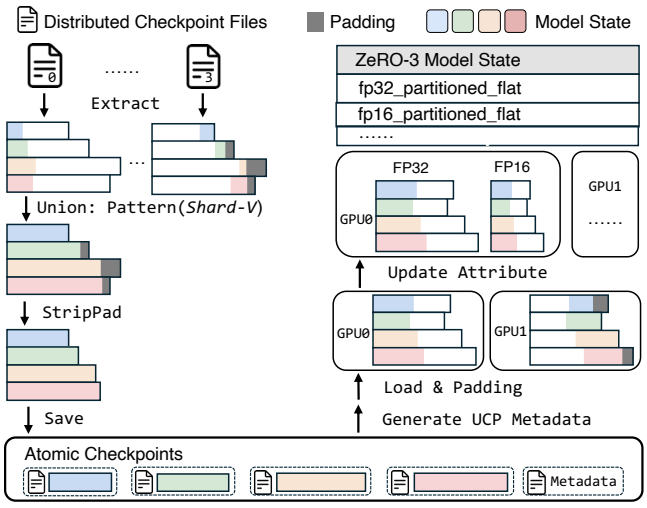


Figure 4: Workflow of UCP to flexibly reconfigure parallelism with ZeRO-3 either as P_{src} or P_{tgt}. The left side shows the process of converting checkpoints from P_{src} (ZeRO-3, DP=4) to atomic checkpoints. The right side shows how atomic checkpoints are converted to P_{tgt} (ZeRO-3, DP=2).

matching and using *Extract*, *Union*, *StripPad* to create consolidated atomic checkpoints without any padding. For example, depending on *Shard-V* or *Shard-H*, UCP would concatenate fragmented parameters from TP into a single consolidated tensor either with the row dimension or column dimension. Similar to ZeRO-3, if 3D parallelism is chosen as the P_{tgt}, a new mapping between atomic checkpoints and GPU ranks is generated first through *UcpInfo*, and each rank loads from atomic checkpoints based on the new mapping policy.

Additional complexities from 3D parallelism also come from the pipeline schedules. For instance, Interleaved 1F1B [38] is a pipeline schedule that reduces the pipeline bubbles by assigning non-contiguous layers of a model to the same rank, resulting in distributed checkpoints highly coupled with the specific order of layer assignment. UCP's atomic checkpoints are decoupled from such scheduling, as the parameters and optimizer states are only associated with their respective parameter names, allowing to reconfigure the mapping between each rank and the parameters.

Sparse Mixture-of-Experts and Irregular Attention GQA. The patterns described in Fig. 3 not only covers model sharding strategies for dense LLMs, as illustrated in the ZeRO-3 and 3D parallelism example, it also covers model architectures such as sparse MoE models [23] and irregular attention mechanisms such as GQA [5], both of which have received intensive interests in LLM training but also require more complex parallelism strategies. Fig. 5 illustrates the complexity. The MoE model in this example defines the weight tensor of an MoE's FFN layer as *[n_experts × hidden_out, hidden_in]*, which is a *fused* weight matrix different from standard MoEs that use separate matrices to represent different experts. With the fused matrix, one can apply TP to this layer. However, different from standard TP, the partition happens along the *hidden_out* dimension, which does not fall under the common *Shard-V* and *Shard-H* pattern. In UCP, this is handled through the *Shard-NC* pattern, which allows UCP to identify it as a 3-dim tensor and apply pattern-aware reconfiguration operations such as *Extract* and *Union* to still obtain a consolidated view of the *fused* matrix from distributed checkpoints.

In the GQA [5] example, the QKV matrices in the multi-head attention (MHA) are also fused together as one tensor. However, different from traditional MHA, the QKV in GQA have different sizes, causing the fused matrix to have an irregular shape of *[q_size + k_size + v_size, hidden]*, where *q_size != k_size == v_size*. If Tensor Parallelism (TP) is applied to this tensor, it needs to partition the tensor along the first dimension for each Q, K, and V but with different sizes. The *Shard-NC* together with shape info allows UCP to identify these variable-size fragments and apply pattern-aware reconfiguration operations accordingly.

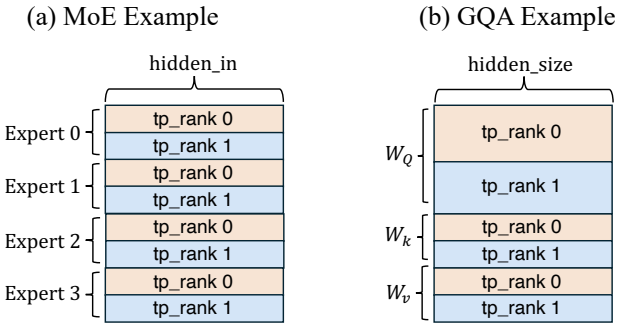


Figure 5: Illustration of the reconfiguration complexities from sparse Mixture-of-Expert models and models with irregular attention mechanisms, such as Grouped Query Attention (GQA). This figure shows examples of MoE with 4 experts and GQA, both with TP=2. The *Shard-NC* pattern allows UCP to identify partitions along different dimensions and variable-size fragments, and UCP supports reconfiguration operations based on these identified patterns.

These examples show that the pattern-based reconfiguration pipeline in UCP is quite flexible, supporting various parallelism strategies as either P_{src} or P_{tgt}. Meanwhile, its carefully designed pattern set allows it to support not only a wide range of complex parallelism strategies but also very extensible to both sparse Mixture-of-Expert models and models with irregular attention in addition to dense models. Introducing a new parallelism strategy involving patterns beyond UCP's current scope would require a similar amount of implementation effort as the examples presented above. We actively work with the community to identify these new patterns to extend UCP to have a wider coverage.

5.4 Efficient Reconfiguration

UCP is intended for high-performance training, therefore the operators described in § 5.3 need to scale well with larger models and more nodes. To achieve low reconfiguration cost, UCP leverages three optimizations: (1) Nested parallel reconfiguration, (2) Redundancy-bypassing loading, and (3) Lazy reconfiguration invocation.

Nested Parallel Reconfiguration. Motivated by leveraging additional nodes to increase the reconfiguration efficiency, we introduce a nested parallel reconfiguration scheme by formulating the reconfiguration as a MapReduce problem [12]. MapReduce was originally introduced as a programming model for processing large datasets in a distributed environment, and has been widely used in big data processing frameworks such as Hadoop [50]. MapReduce decomposes a problem into three phases: Mapper, Shuffler, and Reducer. In our case, as shown in Fig. 6, each parallel mapper reads a distributed checkpoint and parses it according to the parameter fragments of the tensors it contains. The shufflers send the output from mappers, i.e., the tensors each representing a fragment of parameter to the corresponding reducers assigned to process those parameters. The reducers process these parameter fragments based on specific pattern, such as *Shard* and *Partial*, which we discussed in detail in § 5.3.

It is important to note that parameters vary significantly in size. For example, in a 176B model, the embedding parameter has a size of [50257, 12288], while the bias of LayerNorm has a size of [12288]. Randomly assigning these parameters to workers would lead to imbalance, as both computation time and the loading/saving time are proportional to parameter size. This imbalance results in most workers idly waiting for the one handling the heaviest task, which dominates the overall conversion time. To address this issue, we introduce a pre-calculation based balancing method. The master worker divides parameters into N groups based on *numels*. For instance, with 4 workers, the master might assign the embedding parameter to worker 1, while distributing multiple smaller parameters like LayerNorm biases and weights across workers 2, 3, and 4 to achieve a similar total numel count per worker.

Building upon the balanced distribution of parameters across workers, we have implemented a second level of parallelism utilizing multi-core processing within each worker. This approach leverages the full computational capacity within a worker node by engaging all available CPU cores to process parameters concurrently. The nested parallelism strategy – balancing across workers and then across cores within each worker – maximizes resource utilization and minimizes idle time at both the node and core levels, leading to affordable conversion times for large language models.

Redundancy-bypassing loading. Each worker within the same DP group shares or partially shares (except ZeRO3-DP) the same model states, thus needs to load the same atomic checkpoint files. UCP eliminates redundant checkpointing loading by evenly distributing loading workloads among workers in the same DP group. Each rank reads its assigned atomic checkpoint files into CPU memory, transfers them to GPU memory, and then uses all-gather operations to distribute the data to other workers. This optimization significantly alleviates the IO bandwidth pressure by eliminating redundant data transfers from storage to CPU memory and leverages the high-bandwidth GPU-to-GPU interconnects (such as NVLINK at 900 GB/s) to redistribute data once it is loaded into GPUs. Moreover, UCP employs a memory-efficient loading design. UCP loads atomic checkpoints layer by layer and assigns tensors to GPUs based on the selected parallelism strategy. Once a layer is transferred to GPU memory, its tensors are released from CPU memory, reducing peak CPU memory usage from the size of full model checkpoints to that of a single layer.

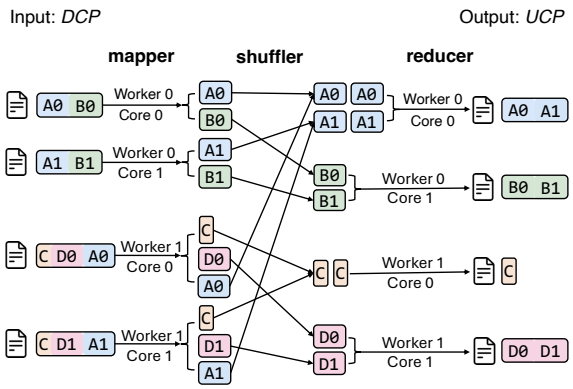


Figure 6: Illustration of the nested parallel reconfiguration process of UCP. UCP leverages a MapReduce-based approach to utilize the aggregated compute and bandwidth of multi-node multi-processors to convert distributed checkpoints into atomic checkpoints in parallel. Meanwhile, it performs careful load balancing to avoid the straggle problem.

Lazy reconfiguration invocation. The careful reader may think "when should the reconfiguration of parallelism be triggered?" or "Is it possible to directly consolidate distributed model states into a single checkpoint file when saving checkpoints?" Such a method has two drawbacks: Consolidating distributed model state into a single checkpoint unacceptably slows down training, and is impractical at extreme scales because there may not be sufficient memory to host the consolidated model states. To avoid slowing down normal training, we adopt a lazy invocation design for UCP. UCP reconfiguration is invoked *only* when a P_{src} and P_{tgt} are different or the hardware changes. This way, the distributed checkpoint saving logic does not need any change, and UCP only incurs $<0.001\%$ of the training time overhead when there is a need for reconfiguration of parallelism since reconfiguration does not add cost to the critical path length, i.e., the normal distributed training process.

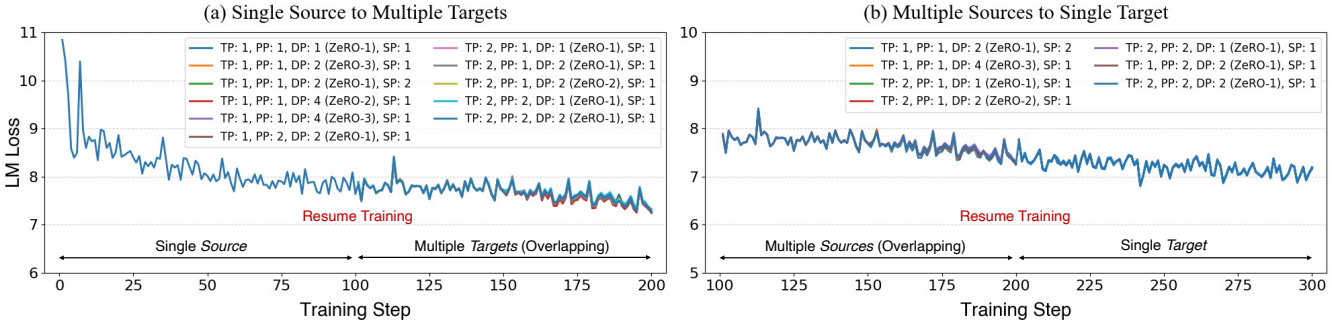


Figure 7: (a) Training curves of reconfiguring training from one *Source* of parallelism into different *Target* parallelisms. (b) Training curves of reconfiguring training from multiple *Source* parallelism strategies to the same *Target*.

6 Evaluation

We implemented UCP in DeepSpeed and evaluate UCP through a series of experiments on training LLMs. Overall, our evaluation aims to answer the following questions:

- Can UCP enable reconfigurable parallelism without compromising model training accuracy?
- How is UCP compared to existing checkpointing systems in terms of supporting flexible parallelism reconfiguration?
- Can UCP generalize to different model architectures?
- How does UCP affect the checkpointing saving and reconfiguration overhead?

6.1 Evaluation Methodology

Workloads. For the accuracy evaluation, we focus on evaluating GPT-style Transformer based models. We select several architectures from prior work: GPT-3 medium [7] ($L = 24, H = 1024, A = 16, 350M$ params), GPT-3 7B ($L = 32, H = 4096, A = 32, 7B$ params), a 176B GPT-3 style model ($L = 70, H = 14336, A = 112, 176B$ params), and a Mixtral-7x8B [23] style MoE model ($L = 32, H = 3072, A = 32, E = 16, 42B$ params), to cover different model configurations and model sizes. We use a subset of the Pile dataset [15] for training to evaluate the impact to the training loss with and without reconfigured parallelism from UCP.

Hardware. We conducted our experiments on: 64xA100 40GB GPUs (256GB DRAM, 10TB storage, 200Gbps interconnect). The evaluation of the 176B GPT style model was conducted on 384xA100 80GB GPUs. The efficiency evaluation of the 1TB model was conducted on 1024xMI250X 64GB GPUs. We measured the I/O bandwidth of the clusters: the NVIDIA GPU clusters achieved 5 GB/s, while the AMD GPU cluster achieved 3 GB/s. These bandwidth levels are comparable to those of consumer-grade SSDs or typical cloud provider offerings.

6.2 Evaluation of Reconfigurable Parallelism

UCP provides flexible reconfiguration from a *Source* parallelism strategy to a different *Target* varying hardware configurations. Since reconfigurable parallelism should not alter the training accuracy (model states preserved after the reconfiguration), we focus first to determine the accuracy impact of UCP with reconfigurable parallelism. After all, regardless how much reconfigurable parallelism a system offers, if it does not preserve training accuracy, it severely limits its applicability in practice. We conduct the evaluation through two categories of reconfiguration:

Single *Source* to multiple *Target*. To test if UCP allows resuming training with different *Target* parallelism strategies and hardware configurations, we first train the GPT-3 model using a configuration of TP=2, PP=2, DP=2 (ZeRO-1), and SP=1. Due to constraints in time and resources, we limited the experiment to the first 200 iterations. We enable UCP on the checkpoints saved at the 100th iteration and resume training using different parallelism strategies and GPUs. We record the LM loss for each iteration. Fig. 7(a) illustrates that the training can be seamlessly resumed using different *Target* parallelism strategies, while achieving consistent convergence if the training were to continue with the *Source* strategy. These results confirm that UCP enables resuming training to different hardware and parallelism configurations.

Multiple *Source* to single *Target*. Fig. 7(b) shows the training curves from multiple *Source* configurations to a single *Target*. We fix all random seeds in the system so that all training job starts with the same model initialization states. We train the GPT-3 model using different *Source* configurations. We then reconfigure their distributed checkpoints saved at the 100th iteration and resume training with a configuration of TP=2, PP=2, DP=1, and SP=1. The results show that regardless different *Source* configurations, all training jobs can resume their training with the same *Target* configuration. Most importantly, the resumed training curves match the curves from the *Source* at iterations 101–200. These results validate the effectiveness of UCP of reconfiguring parallelism while maintaining the training accuracy.

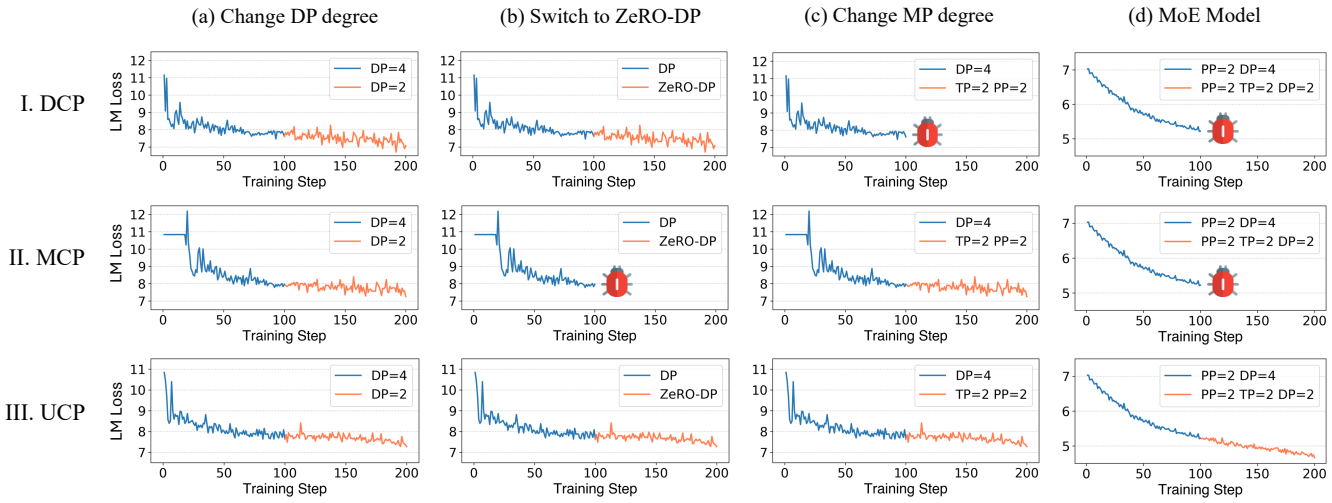


Figure 8: Training curves of (a) changing DP degree, (b) switching to ZeRO-DP, (c) changing MP degree for GPT-3 medium, and (d) for a MoE model when resuming the training from I.DCP [20], II. MCP [33], III.UCP in the middle. "Bug" indicates errors encountered when changing parallelism strategy with a given checkpointing system.

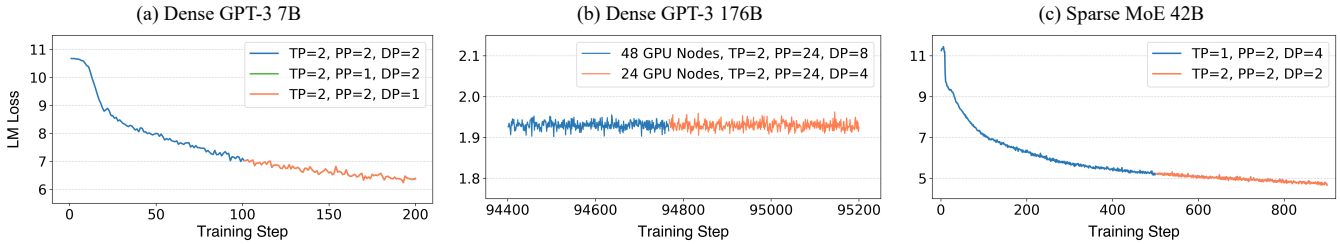


Figure 9: Training curves of (a) dense GPT-3 7B (b) dense GPT-3 176B (c) sparse MoE 42B when resuming the training from UCP in the middle.

Overall, UCP is able to flexibly reconfigure parallelism strategies from any *Source* and any *Target* because its introduced atomic checkpoints decouple distributed checkpoints from specific parallelism and hardware configurations, which serves as a common representation for reconfigurable parallelism. Furthermore, the reconfiguration pipeline provides unified and automatic parallelism reconfiguration through tensor pattern matching and pattern-based transformation primitives for a wide range of parallelism strategies.

6.3 Comparison with Alternative Methods

As shown in Table 1, some checkpointing systems have expanded their capabilities to accommodate reconfigurable parallelism, such as DCP and MCP. To demonstrate the flexibility of state-of-the-art checkpointing systems and validate the results presented in Table 1, we compare the flexibility of DCP, MCP, and UCP under four scenarios: (a) training with DP degree 4, then resume the training with DP degree 2; (b) switch to ZeRO-DP; (c) change the MP degree (including both PP and TP degree); (d) assessing flexibility when applied to a MoE model. For each scenario, we conduct training for 100 iterations, save checkpoints, then resume training with modified parallelism strategies for an additional 100 iterations.

As shown in Fig. 8, DCP supports changes in DP degree and switch to ZeRO-DP but fails to accommodate MP degree change. MCP, while supporting changing both DP and MP degree, shows limitations with MoE models and ZeRO-DP switch. It's important to note that "fail" and "limitation" in this context do not refer to an increase in training loss, but rather to system-level errors. In such cases, developers are required to either restart training from scratch or invest effort in crafting custom conversion scripts. UCP, on the other hand, successfully resumes training under all four scenarios. These results demonstrate UCP's ability to reconfigure a larger and more complicated set of parallelism strategies for DL model training than baseline methods.

6.4 Generalizability to Different Model Architectures

UCP is model architecture agnostic. As such, it is compatible with GPT models varying sizes and both dense and sparse model architectures. Fig. 9 show the training convergence for GPT-3 7B [56], GPT-3 176B [52], and a Mixtral-7x8B

style MoE [23], when resuming from UCP in the middle of training with new parallelism strategies. These figures show that training is seamlessly resumed with UCP, achieving consistent convergence that aligns with the initial training phase across these diverse models. UCP is able to achieve this because it considers a comprehensive set of parallelism strategies and model architectures when designing its tensor pattern matching and pattern-based transformation primitives, as demonstrated through the examples in § 5.3.2. Notably, the GPT-3 176B curve corresponds to a real-world case during the BLOOM 176B [52] training. On July 4, 2022 at 9:00 PM, after nearly three month training, the allocation of 48 nodes expired, and the replacement cluster offered only 24 nodes. Using UCP, the training job was seamlessly resumed on the smaller cluster without interruption. Without UCP, training would have either halted prematurely—leaving tokens incompletely consumed and potentially degrading model quality—or required restarting the entire three-month training process from scratch.

6.5 Reconfiguration Efficiency

Saving cost. As described in § 5.4, UCP incurs no additional saving costs compared to existing distributed checkpointing mechanism and uses *lazy reconfiguration invocation* to reconfigure parallelisms. The input of UCP is the basic distributed checkpoint that is saved periodically. Therefore, the saving cost of UCP is equivalent to that of the standard training process and does not impede the training speed.

Reconfiguration cost. The reconfiguration cost of UCP involves both (1) the transformation overhead from distributed checkpoints to atomic checkpoints and (2) the loading of atomic checkpoints to all parallel workers. Fig. 10 shows that the conversion time remains consistently bounded by approximately 3 minutes, even as model sizes scale up to 1T parameters. This efficiency is achieved through a combination of hardware scaling and the nested parallel conversion method (§ 5.4). The conversion can be executed directly within the training environment, which eliminates the need to transfer exceptionally large checkpoint files (up to TB in size) to different hardware infrastructure. Compared to the sequential(S.Q.) conversion, UCP nested parallelism(N.P.) approach achieves up to a 257x reduction in conversion time. Notably, when the model size increases, the number of GPU nodes also increases to partition the model to limited GPU memory, even for larger models such as LLaMA 3.1 405B that is trained on up to 2000 GPU nodes [54], the conversion time will expected to be bounded to several minutes and does not increase linearly.

To measure the loading cost, we compare UCP with directly loading from the saved distributed checkpoints. As standard distributed checkpoints cannot be loaded when there are changes in GPU counts or parallelism strategies, we keep the same GPU counts and parallelism strategies for the experiments. As shown in Fig. 11, loading from atomic checkpoints incurs approximately 10s overhead compared to standard

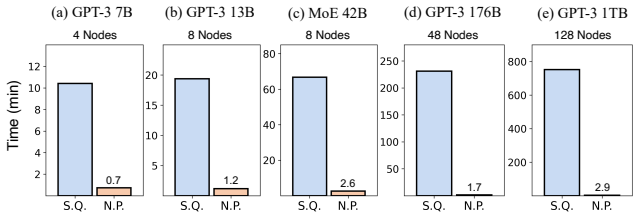


Figure 10: Time cost for converting distributed checkpoints to UCP atomic checkpoints with sequential (S.Q.) and nested parallel (N.P) approach across different model sizes.

distributed checkpoints, consistently across different model size, because the loading time is primarily dominated by the checkpoint data volume, whereas UCP does not change the checkpoint volume significantly. *Redundancy-bypassing loading* further eliminates redundancy when loading the atomic checkpoints within the same DP group. Moreover, the conversion and loading of atomic checkpoints is only triggered lazily (§ 5.4). Therefore, UCP's reconfiguration only accounts for a very small portion of the end-to-end training time.

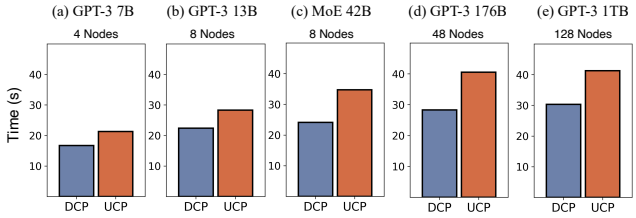


Figure 11: Time cost for loading standard distributed checkpoints (DCP) and loading the UCP checkpoints across models in different sizes.

Table 3 summarizes the end-to-end reconfiguration cost, including checkpoint save, transformation, and load times. In summary, UCP incorporates several system-level optimizations to minimize overhead. The Nested Parallel Reconfiguration is designed to minimize transformation costs, achieving up to a 257× reduction in conversion time. The Redundancy-bypassing loading optimizes loading time by avoiding redundant operations, resulting in a 3× to 20× speedup depending on the degree of data parallelism. The Lazy reconfiguration invocation reduces the frequency of invoking UCP, ensuring that it does not interfere with normal training. UCP is only triggered when a change in parallelism strategy is required. Therefoer, even for models with 1T parameters, total reconfiguration remains under 5 minutes, which is negligible compared to training durations. Although these results are obtained using bandwidth levels comparable to consumer-grade SSDs, UCP maintains functionality even under slower network and storage conditions. However, its performance in such scenarios will be limited by the available bandwidth and I/O throughput.

Model	Hardware Config	Save (min)	Transform (min)	Load (min)	End-to-end (min)
GPT-3 7B	4 nodes A100	0.29	0.73	0.36	**1.38**
GPT-3 13B	8 nodes A100	0.38	1.17	0.47	**2.02**
MoE 42B	8 nodes A100	0.42	2.64	0.58	**3.64**
GPT-3 176B	48 nodes A100	0.48	1.67	0.68	**2.83**
GPT-3 1TB	128 nodes MI250X	0.50	2.93	0.69	**4.12**

Table 3: End-to-end reconfiguration overhead for various models and hardware configurations.

7 Related Work

Checkpointing systems. Checkpointing is widely used as the basic mechanism for fault recovery. While naive checkpointing stalls training, asynchronous checkpointing approaches enable the overlap of disk I/O and model operations [36, 39, 40]. Just-In-Time (JIT) [16] checkpointing employs the state redundancy from DP, creating checkpoints only after failures. To waive the bottleneck from the low bandwidth of remote persistent storage, FastPersist [59] achieves higher bandwidth by extending local NVMe SSDs, and in-memory checkpointing methods [60] utilize remote CPU memories, allowing checkpointing over high-speed training networks. These techniques primarily focus on optimizing checkpoint saving efficiency, which lies on the critical path of the overall training process. UCP does not change the saving logic and is orthogonal to these optimization techniques, allowing for seamless integration.

Spot-instance training. There has been research on using spot instances for DNN training as they are much more cost effective. However, such dynamic environments make overhead of reconfiguration becomes a crucial factor. UCP already introduces several optimizations such as *Nested Parallel Reconfiguration*, *Redundancy-bypassing loading*, *Lazy reconfiguration invocation*, that bring the reconfiguration cost down from hours to a few minutes, making it a potentially viable solution for more elastic training with spot instances. However, prior work, such as Bamboo [55] and Parcae [13], suggest that the dynamism in many cloud spot instances can be extremely high. Therefore, it may require more efficient parallelism reconfigurability to adapt to those environments. We believe that is a very interesting future research study.

Reconfigurable ML systems. VirtualFlow [43] and Singularity [49] decouple DL jobs from physical devices, which enable flexible GPU mapping at runtime but require complex driver-level virtualization and does not support multi-dimensional parallelism changes. Tenplex [58] is a state management library by describing the state as a parallelizable tensor collection (PTC), and generates reconfiguration plans for job change at runtime. While Tenplex supports changing subsets of the parallelism, it is limited to intra-cluster reconfigurations, and storing partitioned parameters in host memory creates memory pressure and impacts training accuracy due to non-optimized state recovery. UCP supports cross-cluster migration while fully preserving training accuracy.

8 Conclusion

This paper presents Universal Checkpointing (UCP), a checkpointing system for training large-scale DNNs with reconfigurable parallelism. The primary goal of UCP is to enable flexible reconfiguration of parallelism strategies through distributed checkpoints. Therefore, UCP provides a new checkpoint structure called atomic checkpoints and builds a pattern-based reconfiguration pipeline that enables automatic and systematic reconfiguration of a broad set of commonly used parallelism strategies while preserving model accuracy. To reduce the reconfiguration cost, UCP introduces nested-parallel reconfiguration and lazy invocation to make the reconfiguration overhead negligible to the end-to-end training cost. Our evaluations on large-scale LLM training show that UCP enables reconfiguration of a larger and more complex set of parallelism strategies with low cost. UCP has been implemented and open-sourced via a PyTorch library for accelerating large-scale DNN training, and its effectiveness has been verified through real-world large-scale LLM pre-training tasks.

Acknowledgments

We sincerely appreciate the anonymous reviewers and our shepherd Young-ri Choi. Their insightful feedback helps significantly improve the quality of the paper. This research was supported by the National Science Foundation (NSF) under Grant No. 2441601. The work utilized the DeltaAI system at the National Center for Supercomputing Applications (NCSA) through allocation CIS240055 from the Advanced Cyberinfrastructure Coordination Ecosystem: Services & Support (ACCESS) program, which is supported by National Science Foundation grants #2138259, #2138286, #2138307, #2137603, and #2138296. The Delta advanced computing resource is a collaborative effort between the University of Illinois Urbana-Champaign and NCSA, supported by the NSF (award OAC 2005572) and the State of Illinois. This work also utilized the Illinois Campus Cluster and NCSA NFI Hydro cluster, both supported by the University of Illinois Urbana-Champaign and the University of Illinois System.

Availability

Universal Checkpointing source code is available at DeepSpeed. For guidance on using and deploying Universal Checkpointing, you can refer to the Huggingface tutorial and Megatron-DeepSpeed examples.

References

[1] GPT-4. https://openai.com/product/gpt-4, 2023.

[2] Claude. https://www.anthropic.com/claude, 2024.

[3] Marah Abdin, Jyoti Aneja, Hany Awadalla, Ahmed Awadallah, Ammar Ahmad Awan, Nguyen Bach, Amit Bahree, and et al. Phi-3 Technical Report: A Highly Capable Language Model Locally on Your Phone. *arXiv preprint arXiv:2404.14219*, 2024.

[4] Meta AI. Introducing Meta LLaMA-3. https://ai.meta.com/blog/meta-llama-3/, 2024.

[5] Joshua Ainslie, James Lee-Thorp, Michiel de Jong, Yury Zemlyanskiy, Federico Lebrón, and Sumit Sanghai. GQA: Training Generalized Multi-Query Transformer Models from Multi-Head Checkpoints. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing (EMNLP'23)*, 2023.

[6] Rishi Bommasani, Drew A Hudson, Ehsan Adeli, Russ Altman, Simran Arora, Sydney von Arx, Michael S Bernstein, and et al. On the Opportunities and Risks of Foundation Models. *arXiv preprint arXiv:2108.07258*, 2021.

[7] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, and et al. Language Models are Few-Shot Learners. In *Proceedings of the 34th International Conference on Neural Information Processing Systems (NIPS'20)*, 2020.

[8] Joseph M. Cavanagh, Kunyang Sun, Andrew Gritsevskiy, Dorian Bagni, Thomas D. Bannister, and Teresa Head-Gordon. SmileyLlama: Modifying Large Language Models for Directed Chemical Space Exploration. *arXiv preprint arXiv:2409.02231*, 2024.

[9] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, and et al. PaLM: Scaling Language Modeling with Pathways. *arXiv preprint arXiv:2204.02311*, 2022.

[10] PyTorch Contributors. PyTorch Distributed Checkpoint - torch.distributed.checkpoint. https://pytorch.org/docs/stable/distributed.checkpoint.html, 2023.

[11] Pytorch Contributors. Pytorch Document: Torch Distributed Elastic. https://pytorch.org/docs/stable/distributed.elastic.html, 2023.

[12] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[13] Jiangfei Duan, Ziang Song, Xupeng Miao, Xiaoli Xi, Dahua Lin, Harry Xu, Minjia Zhang, and Zhihao Jia. Parcae: Proactive, Liveput-Optimized DNN Training on Preemptible Instances. In *Proceedings of the 21st USENIX Symposium on Networked Systems Design and Implementation (NSDI'24)*, 2024.

[14] Assaf Eisenman, Kiran Kumar Matam, Steven Ingram, Dheevatsa Mudigere, Raghuraman Krishnamoorthi, Krishnakumar Nair, Misha Smelyanskiy, and Murali Annavaram. Check-N-Run: a Checkpointing System for Training Deep Learning Recommendation Models. In *Proceedings of the 19th USENIX Symposium on Networked Systems Design and Implementation (NSDI'22)*, 2022.

[15] Leo Gao, Stella Biderman, Sid Black, Laurence Golding, Travis Hoppe, Charles Foster, Jason Phang, Horace He, Anish Thite, Noa Nabeshima, et al. The Pile: An 800GB Dataset of Diverse Text for Language Modeling. *arXiv preprint arXiv:2101.00027*, 2020.

[16] Tanmaey Gupta, Sanjeev Krishnan, Rituraj Kumar, Abhishek Vijeev, Bhargav Gulavani, Nipun Kwatra, Ramachandran Ramjee, and Muthian Sivathanu. Just-In-Time Checkpointing: Low Cost Error Recovery from Deep Learning Training Failures. In *Proceedings of the 19th European Conference on Computer Systems (EuroSys'24)*, 2024.

[17] Yiwen Hu, Huatong Song, Jia Deng, Jiapeng Wang, Jie Chen, Kun Zhou, Yutao Zhu, Jinhao Jiang, Zican Dong, Wayne Xin Zhao, and Ji-Rong Wen. YuLan-Mini: An Open Data-efficient Language Model. *arXiv preprint arXiv:2412.17743*, 2024.

[18] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Xu Chen, HyoukJoong Lee, and et al. GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism. In *Proceedings of the 33rd International Conference on Neural Information Processing Systems (NIPS'19)*, 2019.

[19] Stephen M. Walker II. Everything We Know About GPT-4. https://klu.ai/blog/gpt-4-llm, 2023.

[20] Iris Zhang, Rodrigo Kumpera, Chien-Chin Huang, Lucas Pasqualin. Getting Started with Distributed Checkpoint (DCP). https://pytorch.org/tutorials/recipes/distributed_checkpoint_recipe.html, 2024.

[21] Sam Ade Jacobs, Masahiro Tanaka, Chengming Zhang, Minjia Zhang, Shuaiwen Leon Song, Samyam Rajbhandari, and Yuxiong He. DeepSpeed Ulysses: System Optimizations for Enabling Training of Extreme Long Sequence Transformer Models. *arXiv preprint arXiv:2309.14509*, 2023.

[22] Albert Q. Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, and et al. Mistral 7B. *arXiv preprint arXiv:2310.06825*, 2023.

[23] Albert Q. Jiang, Alexandre Sablayrolles, Antoine Roux, Arthur Mensch, Blanche Savary, Chris Bamford, Devendra Singh Chaplot, and et al. Mixtral of Experts. *arXiv preprint arXiv:2401.04088*, 2024.

[24] Ziheng Jiang, Haibin Lin, Yinmin Zhong, Qi Huang, Yangrui Chen, Zhi Zhang, Yanghua Peng, and et al. MegaScale: Scaling Large Language Model Training to More Than 10,000 GPUs. In *Proceedings of the 21st USENIX Symposium on Networked Systems Design and Implementation (NSDI'24)*, 2024.

[25] Dhiraj Kalamkar, Dheevatsa Mudigere, Naveen Mellempudi, Dipankar Das, Kunal Banerjee, Sasikanth Avancha, Dharma Teja Vooturi, and et al. A Study of BFLOAT16 for Deep Learning Training. *arXiv preprint arXiv:1905.12322*, 2019.

[26] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling Laws for Neural Language Models. *arXiv preprint arXiv:2001.08361*, 2020.

[27] Diederik P. Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. In *Proceedings of the 3rd International Conference on Learning Representations (ICLR'15)*, 2015.

[28] Vijay Korthikanti, Jared Casper, Sangkug Lym, Lawrence McAfee, Michael Andersch, Mohammad Shoeybi, and Bryan Catanzaro. Reducing Activation Recomputation in Large Transformer Models. *arXiv preprint arXiv:2205.05198*, 2022.

[29] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, and et al. PyTorch Distributed: Experiences on Accelerating Data Parallel Training. *arXiv preprint arXiv:2006.15704*, 2020.

[30] Shenggui Li, Fuzhao Xue, Chaitanya Baranwal, Yongbin Li, and Yang You. Sequence Parallelism: Long Sequence Training from System Perspective. *arXiv preprint arXiv:2105.13120*, 2022.

[31] Hao Liu, Matei Zaharia, and Pieter Abbeel. Ring Attention with Blockwise Transformers for Near-Infinite Context. *arXiv preprint arXiv:2310.01889*, 2023.

[32] Stefano Markidis, Steven Wei Der Chien, Erwin Laure, Ivy Bo Peng, and Jeffrey S. Vetter. NVIDIA Tensor Core Programmability, Performance & Precision. In *2018 International Parallel and Distributed Processing Symposium Workshops (IPDPSW'18)*, 2018.

[33] Megatron-Core. Megatron-Core Dist Checkpointing Package. https://docs.nvidia.com/megatron-core/developer-guide/latest/api-guide/dist_checkpointing.html, 2024.

[34] Megatron-LM. Megatron-LM Hybrid Conversion. https://github.com/NVIDIA/Megatron-LM/blob/main/tools/checkpoint/hybrid_conversion.py, 2024.

[35] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory F. Diamos, Erich Elsen, David García, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, and Hao Wu. Mixed Precision Training. In *Proceedings of the 6th International Conference on Learning Representations (ICLR'18)*, 2018.

[36] Jayashree Mohan, Amar Phanishayee, and Vijay Chidambaram. CheckFreq: Frequent, Fine-Grained DNN Checkpointing. In *Proceedings of the 19th USENIX Conference on File and Storage Technologies (FAST'21)*, 2021.

[37] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, Phillip B. Gibbons, and Matei Zaharia. PipeDream: generalized pipeline parallelism for DNN training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP'19)*, 2019.

[38] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Anand Korthikanti, Dmitri Vainbrand, and et al. Efficient Large-Scale Language Model Training on GPU Clusters Using Megatron-LM. *arXiv preprint arXiv:2104.04473*, 2021.

[39] Bogdan Nicolae, Jiali Li, Justin M. Wozniak, George Bosilca, Matthieu Dorier, and Franck Cappello. DeepFreeze: Towards Scalable Asynchronous Checkpointing of Deep Learning Models. In *Proceedings of the 20th International Symposium on Cluster, Cloud and Internet Computing (CCGRID'20)*, 2020.

[40] Bogdan Nicolae, Adam Moody, Elsa Gonsiorowski, Kathryn Mohror, and Franck Cappello. VeloC: Towards High Performance Adaptive Asynchronous Checkpointing at Large Scale. In *Proceedings of the 33rd International Parallel and Distributed Processing Symposium (IPDPS'19)*, 2019.

[41] NVIDIA. Megatron-LM: Evaluation and Tasks. https://github.com/NVIDIA/Megatron-LM?tab=readme-ov-file#evaluation-and-tasks, 2024.

[42] OpenAI. ChatGPT. https://chat.openai.com, 2023.

[43] Andrew Or, Haoyu Zhang, and Michael J. Freedman. VirtualFlow: Decoupling Deep Learning Models from the Underlying Hardware. *arXiv preprint arXiv:2009.09523*, 2021.

[44] Ofir Press, Noah A. Smith, and Mike Lewis. Train Short, Test Long: Attention with Linear Biases Enables Input Length Extrapolation. *arXiv preprint arXiv:2108.12409*, 2022.

[45] PyTorch. PyTorch Lightning. https://github.com/PyTorchLightning/pytorch-lightning, 2019.

[46] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. ZeRO: Memory Optimization Towards Training A Trillion Parameter Models. *arXiv preprint arXiv:1910.02054*, 2019.

[47] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. DeepSpeed: System Optimizations Enable Training Deep Learning Models with Over 100 Billion Parameters. In *Proceedings of the 26th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD'20)*, 2020.

[48] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism. *arXiv preprint arXiv:1909.08053*, 2020.

[49] Dharma Shukla, Muthian Sivathanu, Srinidhi Viswanatha, Bhargav Gulavani, Rimma Nehme, Amey Agrawal, Chen Chen, and et al. Singularity: Planet-Scale, Preemptive and Elastic Scheduling of AI Workloads. *arXiv preprint arXiv:2202.07848*, 2022.

[50] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The Hadoop Distributed File System. In *Proceedings of the 26th Symposium on Mass Storage Systems and Technologies (MSST'10)*, 2010.

[51] Shaden Smith, Mostofa Patwary, Brandon Norick, Patrick LeGresley, Samyam Rajbhandari, Jared Casper, Zhun Liu, and et al. Using DeepSpeed and Megatron to Train Megatron-Turing NLG 530B, A Large-Scale Generative Language Model. *arXiv preprint arXiv:2201.11990*, 2022.

[52] BigScience team. BLOOM: A 176B-Parameter Open-Access Multilingual Language Model. *arXiv preprint arXiv:2211.05100*, 2023.

[53] Gemini Team. Gemini: A Family of Highly Capable Multimodal Models. *arXiv preprint arXiv:2312.11805*, 2024.

[54] Meta AI LLaMA Team. The Llama 3 Herd of Models. https://ai.meta.com/research/publications/the-llama-3-herd-of-models/, 2024.

[55] John Thorpe, Pengzhan Zhao, Jonathan Eyolfson, Yifan Qiao, Zhihao Jia, Minjia Zhang, Ravi Netravali, and Guoqing Harry Xu. Bamboo: Making Preemptible Instances Resilient for Affordable Training of Large DNNs. In *Proceedings of the 20nd USENIX Symposium on Networked Systems Design and Implementation (NSDI'23)*, 2023.

[56] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, and et al. LLaMA: Open and Efficient Foundation Language Models. *arXiv preprint arXiv:2302.13971*, 2023.

[57] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is All you Need. In *Proceedings of the 31st Conference on Neural Information Processing Systems (NIPS'17)*, 2017.

[58] Marcel Wagenländer, Guo Li, Bo Zhao, Luo Mai, and Peter Pietzuch. Tenplex: Dynamic Parallelism for Deep Learning using Parallelizable Tensor Collections. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles (SOSP'24)*, 2024.

[59] Guanhua Wang, Olatunji Ruwase, Bing Xie, and Yuxiong He. FastPersist: Accelerating Model Checkpointing in Deep Learning. *arXiv preprint arXiv:2406.13768*, 2024.

[60] Zhuang Wang, Zhen Jia, Shuai Zheng, Zhen Zhang, Xinwei Fu, T. S. Eugene Ng, and Yida Wang. GEMINI: Fast Failure Recovery in Distributed Training with In-Memory Checkpoints. In *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP'23)*, 2023.

[61] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, and Jamie Brew. HuggingFace's Transformers: State-of-the-art Natural Language Processing. *arXiv preprint arXiv:1910.03771*, 2019.

[62] xAI. Grok-2. https://x.ai/blog/grok-2, 2024.

Towards High-Performance Transactional Stateful Serverless Workflows with Affinity-Aware Leasing

Jianjun Zhao[†], Haikun Liu[†,*], Shuhao Zhang[†], Haodi Lu[†], Yancan Mao[‡], Zhuohui Duan[†]
Xiaofei Liao[†], Hai Jin[†]
[†]National Engineering Research Center for Big Data Technology and System,
Service Computing Technology and System Lab/Cluster and Grid Computing Lab,
School of Computer Science and Technology, Huazhong University of Science and Technology, China
[‡]School of Computing, National University of Singapore, Singapore

Abstract

Function-as-a-Service (FaaS) is the most prevalent serverless computing paradigm, offering significant flexibility to develop, deploy, and operate cloud applications. However, traditional FaaS frameworks face significant challenges in operating transactional stateful workflows, which often involve multiple functions with shared state. Previous solutions rely on external datastores to manage shared state, suffering from high communication overhead to guarantee transactional consistency for stateful workflows.

In this paper, we present RTSFaaS, an RDMA-capable transactional stateful FaaS framework that achieves high performance while guaranteeing transactional consistency. RTSFaaS exploits a lease-based concurrency control protocol to dynamically assign and transfer leases among workers to achieve concurrency control. Specifically, RTSFaaS incorporates two key designs: (1) an affinity-aware lease assignment mechanism that improves the benefit of caching by dynamically assigning data leases to selected workers according to the data function affinity, and (2) an RDMA-capable dynamic lease transferring mechanism to reduce the cost of locking by serializing concurrent data accesses with one-sided RDMA primitives. Experimental results show that RTSFaaS achieves up to 5× and 20× performance speedup compared with state-of-the-art transactional stateful FaaS platforms–Boki and Beldi, and up to 1.7× and 2.1× performance improvement when their concurrency control protocols implemented for RDMA networks are applied to RTSFaaS.

1 Introduction

Serverless computing [12, 22, 27] has gained increasing popularity since it promises to simplify the programming, deploying, and operating scalable cloud applications, particularly through the *function-as-a-service* (FaaS) paradigm [11, 17, 37, 45]. This paradigm is widely adopted by mainstream vendors, including AWS Step Functions [45], AWS Lambda [46], Azure Durable Functions [37], Alibaba Function Compute [11], and Huawei Cloud Functions [17]. FaaS simplifies cloud deployment by reducing the operational complexity of managing servers while providing scalability and elasticity [33]. Developers only need to program cloud applications with *functions* that can be executed elastically by FaaS platforms. To support complex applications, FaaS platforms allow developers to compose functions into workflows [32, 66], typically abstracted as *directed acyclic graphs* [33] (DAGs).

Although the mainstream FaaS paradigm is stateless, we argue that FaaS platforms should support state management for the stateful applications built with functions [25, 60]. For example, in a banking service scenario [23], a transfer business can be implemented with multiple functions and executed in a workflow (i.e., withdraw ⇒ deposit). Because the application state is shared, these functions should be processed within a single transaction to guarantee serializable consistency when concurrent transactions access the shared state. Current stateful FaaS platforms [24, 36, 48] rely on external datastores like Amazon DynamoDB [20, 43] and S3 [44] to manage the application state. While the disaggregation of compute and storage enables shared state management, it often introduces substantial performance overhead from frequent remote state accesses [30, 48], which challenges stateful FaaS platforms in achieving strong consistency without sacrificing performance.

First, the concurrency control usually leads to high communication overhead due to frequent remote accesses to the lock status. To guarantee transactional consistency for stateful workflows, state-of-the-art stateful FaaS platforms like Boki [25], Beldi [60], and T-Statefun [15] rely on a remote datastore to implement concurrency control protocols such as *two-phase locking* (2PL) [21, 49] and *optimistic concurrency control* (OCC) [1, 25]. However, these protocols require frequent remote communications for acquiring and releasing locks, as well as validating conflicts between concurrent function executions. This results in high latency, particularly in high-contention environments, since the communication overhead associated with *remote memory access* (RMA), lock

*Corresponding author: Haikun Liu (hkliu@hust.edu.cn)

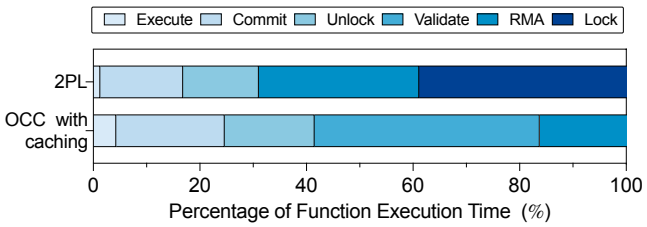


Figure 1: Breakdown of function execution time for the banking service workload. Both 2PL [60] and OCC with caching [25] are optimized using one-sided RDMA verbs.

management, and conflict validation becomes a significant bottleneck. As illustrated in Figure 1, these overheads remain the major performance bottleneck of stateful FaaS workflows even using RDMA networks.

Second, concurrency control protocols often undermine the efficiency of caching in stateful FaaS platforms. To reduce the performance overhead of remote state accesses, many stateful FaaS platforms adopt caching mechanisms that store frequently-accessed data locally [34, 48, 57, 61]. However, the benefit of caching is often limited when applications require strong consistency guarantees. In 2PL, the necessity of acquiring locks from a remote datastore diminishes the effectiveness of local caches since functions must acquire remote locks even when the cache is valid. Moreover, frequent state updates may invalidate the local cache, further reducing its efficiency. In OCC, while functions can read and modify the local cache optimistically, they must validate whether the original data has been modified by other transactions before committing a transaction. If conflicts are detected, the transaction is aborted and retried, offsetting the benefit of caching.

In this paper, we present RTSFaaS, an **R**DMA-capable **t**ransactional **s**tateful **FaaS** framework that achieves high performance while guaranteeing strong consistency. RTSFaaS exploits a lease-based concurrency control protocol to dynamically assign and transfer leases[1] among workers. Unlike traditional leasing mechanisms, our approach allows a worker to cache and control access to specific objects in a distributed cache with exclusive rights. Thus, there is only one replica of an object that is exclusively cached by a single worker from a global perspective. In this way, RTSFaaS allows a large majority of transactions to be executed with a strong affinity for cached objects, and thus reduces remote communication overhead and improves the benefit of caching. Specifically, RTSFaaS incorporates two key designs to address the aforementioned challenges of handling transactional stateful serverless workflows in current FaaS platforms.

First, we propose an **affinity-aware lease assignment mechanism** to enhance the benefit of caching by assigning a group of interdependent objects to a single worker with an exclusive lease. RTSFaaS maintains a statistical table to count the total number of requests and data access frequency for each worker. Upon each request, RTSFaaS exploits a score-based scheduling policy to assign it to a worker who has the strongest data affinity for the request according to the statistical table. After assigning a batch of requests to different workers, RTSFaaS uses the latest statistics to designate workers as leaseholders for each object. If a worker has the highest access frequency to an object, it becomes the leaseholder of the object. In this way, RTSFaaS allows more functions to execute with the local cache, thereby significantly enhancing the benefit of caching and reducing remote memory accesses.

Second, we propose an **RDMA-capable dynamic lease transferring mechanism** to significantly reduce the network communication overhead of locking. For each batch of requests, RTSFaaS decouples the function execution and the consistency guarantee into two non-overlapping phases: planning and execution. During the planning phase, workers identify data dependencies among functions to collaboratively construct a global *task precedence graph* (TPG), which is used to serialize the function execution. During the execution phase, RTSFaaS allows leaseholders to cache and control the access to specific objects with exclusive rights. These leaseholders independently prefetch the corresponding objects into their local caches. Then, they execute functions and dynamically transfer the lease of objects to others according to the global TPG, which ensures sequential data accesses. When a worker has to access objects that are not cached locally, the worker checks the lease of remote objects and directly accesses other workers' caches using one-sided RDMA verbs.

We implement RTSFaaS [42] using a cluster connected with RDMA networks and evaluate it with commonly-used microservice benchmarks [23, 25, 60]. Experimental results show that RTSFaaS improves the throughput of applications by up to 5× and 20× compared with the state-of-the-art stateful FaaS platform–Boki [25] and Beldi [60]. To further evaluate the efficiency of our lease-based concurrency control protocol, we implement their concurrency control protocols in an RDMA-enabled environment for comparison. Experimental results show that RTSFaaS achieves up to 1.7× and 2.1× performance speedup over these RDMA-based implementations.

In summary, we make the following contributions:

- We design RTSFaaS, an RDMA-capable transactional stateful FaaS framework that significantly improves the performance while guaranteeing strong consistency for transactional stateful serverless workflows.
- We propose an affinity-aware lease assignment mechanism that improves the benefit of caching by dynamically assigning data leases to selected workers according to the data-function affinity.
- We propose RDMA-capable dynamic lease transferring to reduce the cost of locking by serializing concurrent data accesses with one-sided RDMA primitives.

[1] The lease mechanism grants temporary access or ownership rights to a resource (such as data or a lock) for a specific period.

- Experimental results show that RTSFaaS outperforms state-of-the-art stateful FaaS platforms, achieving up to 5× and 20× higher throughput than Boki [25] and Beldi [60], respectively. Even when their concurrency control protocols are adapted to RDMA networks and integrated into RTSFaaS, our design still yields up to 1.7× and 2.1× performance improvements.

2 Background and Motivation

In this section, we present the background of transactional stateful FaaS workflows, discuss the limitations of existing solutions, and explore the potential of using *remote direct memory access* (RDMA) in FaaS platforms.

2.1 Transactional Stateful FaaS Workflows

The FaaS paradigm [4, 25, 30, 36, 48, 57, 60] has become increasingly popular for developing, deploying, and operating cloud applications. However, because of the stateless nature of functions [25], existing FaaS platforms can not efficiently support stateful applications that require state sharing and transactional processing, such as payment processing [30], travel reservation [60], inventory management [25], and business services [23].

To support these complex applications, multiple functions can be composed into a workflow, represented as a *directed acyclic graph* (DAG) [33, 45]. Each node in this graph corresponds to an individual function, while edges denote the execution order and the transfer of intermediate results between functions. For instance, in the banking service scenario as shown in Figure 2, several services such as payment (pay), saving (save), and transfer (withdraw ⇒ deposit) are provided. Despite the distinctiveness of these functions, they share the same application state, such as the balance of users' accounts. Additionally, the transactional semantics are crucial to ensure that the transfer service is registered simultaneously on both the recipient and sender accounts. Consequently, these functions in the workflow (withdraw ⇒ deposit) must be executed with a transaction. This requirement mandates FaaS platforms to guarantee transactional consistency, i.e., the results of concurrent transactions are conflict-equivalent to some serial execution of these transactions [25, 60].

2.2 Limitations of Existing Solutions

Current stateful FaaS platforms [24, 36, 48] rely on external datastores like Amazon DynamoDB [20, 43] and S3 [44] to manage the application state. While the disaggregation of computing and storage facilitates the shared state management, it makes the stateful FaaS platform struggles to achieve strong consistency while maintaining high performance. Existing stateful FaaS platforms either support weak consistency [34, 48, 57] or cause high performance overhead

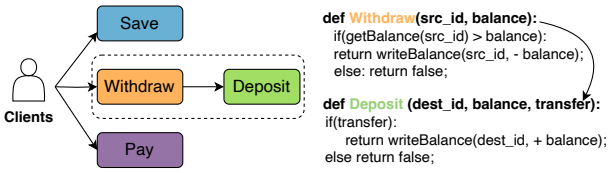


Figure 2: An example of the banking service workflow [23]

to ensure transactional consistency [15, 25, 30, 60]. None of them provides an efficient way to guarantee transactional consistency for transactional stateful serverless workflows.

Weak Consistency. A number of FaaS platforms have been designed on the basis of causal consistency principles, including Cloudbust [48], HydroCache [57], FaaSTCC [34], and CausalMesh [61]. These FaaS platforms exploit local caches on each computing node to achieve high performance. However, these platforms only guarantee weak consistency, such as *transactional causal consistency* (TCC) for workflows. To guarantee TCC, a workflow reads from a causal snapshot [34], and other workflows cannot observe its results until it is entirely completed. However, TCC allows anomalies such as stale reads and write-write conflicts, which do not satisfy the requirements of transactional stateful FaaS applications.

Transactional Consistency. To guarantee transactional consistency, stateful FaaS platforms like Beldi [60], Boki [25] and T-Statefun [15] rely on a remote datastore to implement concurrency control protocols, such as *two-phase locking* (2PL) and *optimistic concurrency control* (OCC). In 2PL, transactions ensure serializable execution by acquiring/releasing locks. In contrast, OCC allows transactions to read and modify the local cache optimistically, but conflicting operations must be checked before committing transactions. Unfortunately, these protocols lead to high communication overhead due to frequent remote accesses for lock management and conflict validation. Additionally, they often undermine caching benefits by invalidating cached data or causing frequent transaction retries.

2.3 Remote Direct Memory Access

RDMA is a networking technology that enables direct access to the main memory of a remote node over networks [6, 28, 67]. Unlike the TCP/IP protocol, RDMA can eliminate memory copying among buffers (zero-copy) via bypassing the OS kernel. Thus, it achieves high-bandwidth and low-latency network transmission with minimal CPU involvement. RDMA supports two types of verbs for remote memory accesses: one-sided and two-sided verbs [28]. Two-sided verbs (send, recv) use channel semantics in which the sender should interrupt the receiver's CPU to process network messages. In contrast, one-sided verbs (read, write, and atomic verbs) use memory semantics. The sender should first register a remote memory region on the receiver to perform RDMA operations. Then, the receiver's *RDMA network interface card* (RNIC) exploits

the *direct memory access* (DMA) mechanism to access data without involving its CPUs.

RDMA in the FaaS platforms. RDMA has been widely used in *high-performance computing* (HPC) and cloud computing scenarios, supporting distributed transactions [7, 19, 29, 53], disaggregated memory systems [31, 47, 52, 62, 63], and other data center environments [7, 13, 16, 54]. RDMA also offers new possibilities in the FaaS domain. Recent RDMA-based FaaS frameworks such as rFaaS [13], MITOSIS [54], and RMMAP [33] have exploited RDMA technologies for various application scenarios, including rapid remote forking [13], fast remote container initialization [54], and serialization-free state transfer [33]. Although these proposals have demonstrated the significant potential of using RDMA technologies for optimizing the performance of FaaS platforms, no effort has been made on transactional stateful serverless workflows using RDMA networks.

3 RTSFaaS Design

We first present the RTSFaaS architecture, and then describe the lease assignment and transfer mechanisms.

3.1 System Overview

Figure 3 illustrates the RTSFaaS architecture, designed for elastic autoscaling by disaggregating storage and compute resources [25, 48, 60]. RTSFaaS consists of three main components: the RTSFaaS driver, workers, and a *key-value* (KV) store using TiKV [39, 51], which supports geo-distributed replication mechanisms for high availability. The RTSFaaS driver and workers run in individual Docker containers [18]. Client requests, received by the API gateway, are routed by the RTSFaaS driver to appropriate workers. Each worker consists of a function scheduler, a set of executors, and a local data cache shared by all executors. The scheduler invokes a set of functions to execute the workflow within these executors according to the global *task precedence graph* (TPG). To eliminate the cost of data synchronization among multiple replicas and to save memory space, each KV object is exclusively cached by only one worker (i.e., a leaseholder), and can be accessed by another worker via RDMA operations when its lease is transferred to the remote executor. All workers' local data caches construct a global shared memory pool which serves as an intermediary between these executors and the KV store. In this architecture, RTSFaaS should address two key problems as follows.

(1) **How to exclusively cache objects in multiple workers to enhance the data-function affinity?** If objects are randomly or on-demand cached by different workers, executors usually have to frequently access objects from remote shared caches, causing significant network communication overhead. Thus, we propose an affinity-aware lease assignment mechanism to enhance the benefit of caching (Section 3.2).

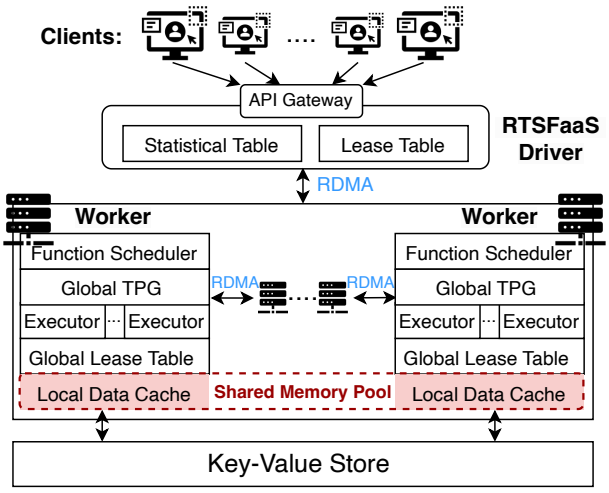


Figure 3: RTSFaaS architecture

(2) **How to access remote shared caches efficiently while maintaining data consistency when objects required by executors are missing in the local data cache?** In traditional concurrency control mechanisms, the data access is tightly coupled with the locking operation. They may suffer from high-latency data accesses when the contention for locks is intensive. Thus, we construct a TPG for functions with data dependencies to serialize the function execution, and dynamically transfer the lease of objects to other workers so that they can access these objects temporarily via one-sided RDMA operations (Section 3.3).

3.2 Affinity-aware Lease Assignment

To assign an incoming request to a worker so that they have the strongest data-function affinity, the RTSFaaS driver maintains a statistical table to record the following information for each worker: 1) the total number of functions N_i assigned to worker i; 2) for each KV object j, the total number of accesses $N_i(j)$ introduced by all functions in worker i.

When a request arrives, the driver identifies its read-write sets, denoted as $K = \{k_1, \ldots, k_n\}$, which includes all data items required by all functions in the current workflow. The affinity score A_i for worker i is calculated as the sum of access counts $N_i(k_j)$ for each key in K:

$$A_i = \sum_{k_j \in K} N_i(k_j)$$

The affinity score reflects how frequently the worker accesses all keys involved in the request. Thus, this request is preferentially assigned to a worker who has a strong data-function affinity. However, load balancing is also critical to prevent worker overloading. To achieve this goal, the total number of functions assigned to each worker (N_i), is used to compute a load balancing score S_i, which favors workers with fewer assigned functions. The load balancing score is

calculated using the following linear formula:

$$S_i = 1 - (N_i - min)/(max - min)$$

where *min* and *max* represent the minimum and maximum number of functions assigned to a worker. Thus, workers with lighter loads have higher scores.

To make a trade-off between data-function affinity and load balancing, the affinity score is also normalized using a linear mapping similar to the load balancing score. Let min' and max' be the minimum and maximum affinity scores among all workers. The normalized affinity score A'_i for worker i is calculated by:

$$A'_i = (A_i - min')/(max' - min')$$

The final score is calculated as the weighted sum of S_i and A'_i. By default, the RTSFaaS driver assigns equal weight to each score. Then, the driver forwards the request to the worker with the highest score and updates the statistical table. After a batch of requests is dispatched, a lease table is instantiated based on the statistics table, designating the worker with the highest affinity score as the leaseholder. If multiple workers get the same score, RTSFaaS assigns the lease to the one with the smallest worker ID.

3.3 RDMA-capable Dynamic Lease Transfer

RTSFaaS decouples the consistency guarantee and the function execution into two non-overlapping phases: planning and execution. In the planning phase, RTSFaaS constructs a *task precedence graph* (TPG) for functions with data dependencies to serialize the function execution. In the execution phase, RTSFaaS schedules the execution of functions based on the TPG, and dynamically transfers the lease of objects to other workers for concurrent state accesses.

3.3.1 Task Precedence Graph Construction

In the planning phase, workers identify fine-grained data dependencies within a batch of requests assigned to them and collaboratively construct a global TPG. Before delving into details, we first formally define the following terms:

Definition 1 (Workflow)**.** A workflow is defined as a set of functions $f_1, f_2, \ldots, f_n$ triggered by a client request. Each function f_i in the workflow accesses exactly one key. To satisfy this constraint, RTSFaaS decomposes any function that accesses multiple keys into a sequence of smaller functions, each operating on a single key, while preserving the original execution dependencies.

Definition 2 (Function Timestamp)**.** The function timestamp represents the time when a request invokes it. The driver is responsible for assigning a global timestamp to each request, ensuring a consistent ordering of requests across all workers.

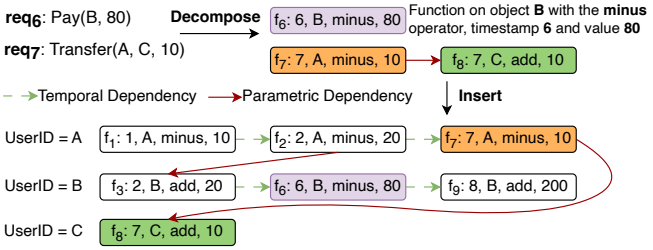


Figure 4: Dependencies and local TPG construction

For each workflow, all functions in the same workflow share the same timestamp. In this context, we define the timestamp of function f_i as $t(i)$.

Definition 3 (Temporal Dependency)**.** f_i temporally depends on f_j if they access the same state but are not invoked by the same workflow request, and f_i has a larger timestamp.

Definition 4 (Parametric Dependency)**.** f_i parametrically depends on f_j if they belong to the same workflow, and f_i needs the output of f_j.

For example, in Figure 4, $(f_6 : 6, B, minus, 80)$ is a function that operates on object B using the *minus* operator and a value 80 when the function timestamp is 6. f_6 temporally depends on f_3 as they access the same object B with different timestamps. Taking f_7 and f_8 as another example, they are triggered by a workflow request req_7 and f_8 needs the outputs of f_7. Thus, f_8 parametrically depends on f_7.

Definition 5 (Task Precedence Graph)**.** Given a batch of requests, the *task precedence graph* (TPG) abstracts functions as vertexes, and edges as temporal and parametric dependencies. Without losing generality, we represent the edge from f_i to f_j as $< f_i \rightarrow f_j >$, which means f_j depends on f_i.

Lemma 1 (Transactional Consistency)**.** If all functions invoked by a batch of requests Q are processed according to temporal and parametric dependencies, the transactional consistency is guaranteed.

Proof Sketch: The task precedence graph is a schedule of functions and we now prove it is conflict-serializable. According to the conflict serializability theorem [5, 38, 56], the schedule is conflict-serializable if and only if its conflict graph is acyclic. For temporal dependencies, if there is a direct edge from f_i to f_j, $t(j)$ is larger than $t(i)$. We assume that there is a cycle in the dependency graph, i.e., there is a set of edges: $< f_1 \rightarrow f_2 >, < f_3 \rightarrow f_4 >, \ldots < f_{n-1} \rightarrow f_n >, < f_n \rightarrow f_1 >$. This implies that the timestamps increase sequentially from $t(1)$ to $t(n)$, and $t(n)$ is less than $t(1)$. Such an assumption leads to contradictions. Thus, there must be no cycles in the dependency graph if we only consider temporal dependencies. Since parametric dependencies among functions only exist within the same workflow, they inherently avoid the cyclic pattern. Therefore, the conflict serializability is guaranteed, if all

functions are scheduled according to temporal and parametric dependencies.

Local TPG Construction. Each worker identifies fine-grained temporal and parametric dependencies within a batch of requests assigned to them and constructs a local TPG. Figure 4 illustrates how a worker constructs a local TPG for the banking service. At first, requests req_6 and req_7 are decomposed into individual functions f_6, f_7, and f_8, which represent vertexes in the local TPG. During this decomposition, parametric dependencies among functions in the same workflow can be identified. For instance, f_8 parametrically depends on f_7. To identify temporal dependencies among functions that may arrive out-of-order, workers insert newly arrived functions into key-partitioned sorted lists. In these lists, functions related to the same KV object are ordered by their timestamps. For example, even though f_9 arrives before f_6, the worker inserts f_6 before f_9 because f_6 has an earlier timestamp. Consequently, for a batch of requests, the worker sorts the functions related to the same object according to their timestamps. Then, temporal dependencies are identified by iterating through all functions in these sorted lists.

Global TPG Construction. When each worker has constructed its local TPG individually, all workers collaboratively build a global TPG based on the lease table provided by the RTSFaaS driver. As illustrated in Figure 5, node 1 has the lease of objects A and B, while node 2 owns objects C and D. RTSFaaS distinguishes *local functions* which access only locally cached data, and *remote functions* which have to access remote data. Using the lease table, each worker dispatches metadata associated with remote functions to the workers who are the leaseholders of the required data. For example, node 1 sends metadata of f_8 to node 2, and node 2 sends metadata of f_{12} to node 1. Workers collect these metadata from other nodes as virtual vertexes and integrate them with their initial local TPG to detect temporal dependencies. For instance, the virtual vertex $(N_1, 7, C)$ contains the metadata of f_8 from node 1 with timestamp 7 and accessing data item C. Thus, node 2 inserts $(N_1, 7, C)$ before f_{10} because f_8 has an earlier timestamp. In this way, each worker constructs a global TPG for a batch of functions that interact with objects stored in its local data cache.

3.3.2 Dynamic Lease Transferring

In the execution phase, each worker maintains a *global lease table* to facilitate dynamic lease transferring, and a *local data cache* accessible to other workers. The *global lease table* is used to determine which worker is the leaseholder of specific objects and the storage order of these objects on each worker. RTSFaaS restricts data prefetching and access control management to leaseholders. These leaseholders fetch the required objects from the KV store in advance to construct the local data cache, ensuring that the data is readily available for subsequent processing.

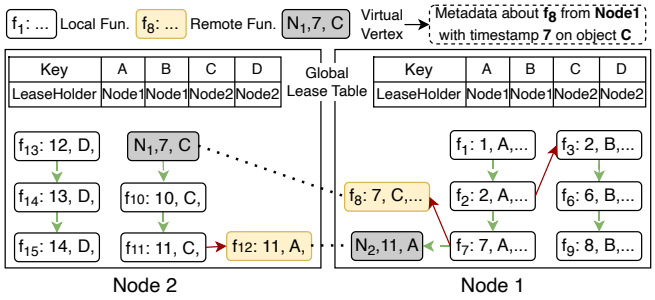


Figure 5: Global TPG construction

Local Cache. Workers share the local data cache by maintaining KV objects in the order of their keys. Each worker sorts KV objects based on the numerical values represented by their keys. When a worker is the leaseholder of a key, it proactively fetches the corresponding values from remote storage and writes them sequentially into pre-allocated memory regions. Additionally, each value is augmented with a 16-bit *lease flag* at the beginning to reflect its lease status. Initially, the lease of each value is assigned according to the lease specified in the global lease table. After processing a batch of requests, each worker writes back updates from its local cache to the remote KV store, ensuring a consistent view for the next batch and enabling fault tolerance (see Section 4.2).

Function Execution. RTSFaaS exploits a dynamic lease transferring mechanism to guarantee data consistency across local data caches. Functions are executed sequentially according to the global TPG generated during the planning phase. When a worker encounters a normal vertex (local or remote function), it first checks the global lease table to determine the data's location. For a local function, the worker accesses data directly using the starting address of its local memory region and calculates the value's offset based on the key index and value size. For a remote function, the worker calculates the memory address based on the starting address of the remote memory region and the key's index, retrieving data using one-sided RDMA verbs without involvement in remote CPUs.

Regardless of whether the data is cached locally or remotely, the worker must verify whether it has the current lease of the object by comparing its worker ID with the lease flag. The worker can continue to access data and execute functions only when the lease of the object has been transferred to it. This ensures that the worker can safely access or update the object only if it holds the data lease. After completing the data access, the remote worker then transfers the lease to the original leaseholder based on the *global lease table*. The leaseholder then continues to transfer data leases according to the global TPG. The leaseholder transfers the lease to other workers based on the virtual vertex, which contains metadata about the source worker and the object required. When a virtual vertex is scheduled, the leaseholder transfers the data lease to the corresponding worker by updating lease flags in its *local data cache*.

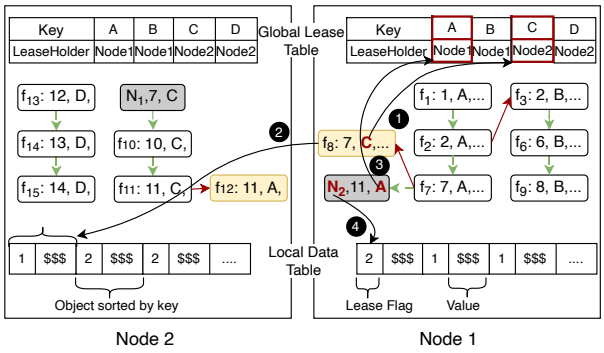


Figure 6: Dynamic lease transferring

As shown in Figure 6, workers adopt a breadth-first exploration mechanism to schedule functions based on the global TPG. Node 1 first picks an executor for f_1, as it has no unresolved dependencies. Once f_1 completes, f_2 becomes eligible for execution. After f_2 has been processed, f_7 and f_3 can be scheduled on different executors in parallel, as their data dependencies are resolved. f_8 presents a more complex case, as it is a remote function that requires access objects cached on other workers. To execute f_8, the executor first checks the global lease table to locate the object in Node 2 and get its index (0) in Node 2's local data cache (❶). It then uses one-sided RDMA verbs to fetch the remote object and checks the lease flag to ensure that Node 1 is the current leaseholder before performing any updates (❷). When scheduling the virtual vertex (N_2, 11, A), Node 1 checks the global lease table to get the index of the object (key = A) (❸). Then, it transfers the lease to Node 2 by updating the lease flag from 1 to 2 (❹). This indicates that once all dependencies of f_{12} are resolved, Node 2 can execute f_{12} by directly accessing object A from Node 1's cache.

Handling Transaction Aborts. To address transaction aborts, we adopt an approach inspired by Yao et al. [58], which effectively eliminates cascading aborts during transaction execution. While RTSFaaS resolves transactional conflicts before execution to ensure that no abort occurs due to such conflicts. However, transaction aborts may still result from constraints of database fields, such as non-negative balance. To handle this, RTSFaaS introduces a condition-variable check as the initial function within each transactional workflow. All subsequent functions in the transaction are parametrically dependent on this initial check. If the condition-variable check fails, it propagates disabled signals to the interdependent functions in the workflow, thereby avoiding cascading aborts and reducing their associated overhead during execution.

4 Implementation

In this section, we describe the RDMA-based data transfer protocol and fault tolerance mechanisms.

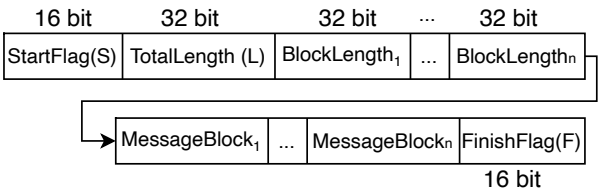


Figure 7: Data layout of RDMA messages

4.1 RDMA-based Data Transfer

4.1.1 RDMA Channel

The RDMA channel in RTSFaaS enables efficient communication between the driver and workers, as well as two workers. RTSFaaS employs a push-based data transfer model with RDMA write and a structured message layout to ensure reliability and parallel processing.

RDMA Verbs. RTSFaaS adopts RDMA write instead of RDMA read to achieve the push-based transfer model due to the following reasons: 1) First, since workers are usually computing intensive, the push-based transfer model empowered by RDMA write is preferable. This approach allows the driver to write the workers' memory directly, enabling workers to poll local memory efficiently. In contrast, using RDMA reads, the driver would continuously read the remote memory till the requested data is available, leading to additional network traffic and higher latency. 2) Second, the RDMA read entails a round-trip message, and results in higher latency and CPU utilization [16]. In contrast, one-sided RDMA write requires only a single trip per message.

Data Layout of RDMA Messages. We carefully design the data layout of RDMA messages to ensure data integrity and to support parallel processing. As illustrated in Figure 7, the message comprises several elements: a short start-flag S indicating the beginning of the message, the total message length L, and the length of each message block. The number of message blocks n corresponds to the number of parallel instances in the receiver, i.e., n is the number of its function executors for each worker. Subsequently, a set of message blocks follows, with each block containing a portion of the messages slated for processing. At last, a short finish-flag F denotes the end of the message.

4.1.2 RDMA Communication Protocol

Connection Establishment. In RTSFaaS, each worker establishes connections with the RTSFaaS driver and other workers, facilitated by a configuration file that specifies the network addresses of all nodes. During the initialization, each worker configures its RDMA resources, including memory regions for data transfer, completion queues, and queue pairs. Simultaneously, the driver sets up a listening endpoint to await connection requests from workers.

When a worker sends a connection request, the RTSFaaS driver responds by performing a handshake to exchange in-

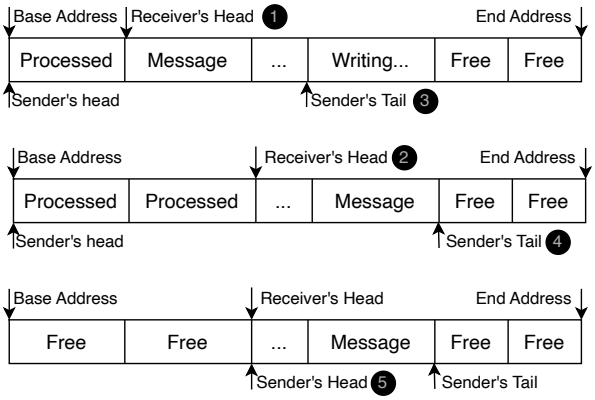


Figure 8: Circular buffer for data transfer

formation about memory regions. Simultaneously, workers exchange connection requests to establish direct connections with one another. During this process, the driver and workers exchange RegionTokens, which encapsulate details of shared memory regions, such as base addresses and sizes. These tokens enable direct memory access during RDMA operations.

Data Transfer. RTSFaaS achieves the RDMA-based data transfer using circular buffer [16, 19], as shown in Figure 8. As RTSFaaS adopts a push-based transfer model, the circular buffer is initialized in the receiver, and there is one buffer for each sender/receiver pair. The receiver periodically polls the buffer at the "Head" position to detect new messages (❶). As shown in the message layout, any start-flag value S in the head indicates a new message. Then, the receiver polls the message trailer based on the total message length L. When it becomes the finish-flag F, the entire message has been received because RDMA writes are performed in a FIFO order. Finally, the receiver advances the head pointer (❷).

The sender utilizes RDMA write to transfer messages to the buffer tail (❸), incrementing the tail pointer with each transmission (❹). It maintains a local copy of the receiver's head pointer and ensures that messages are only written up to that point. The receiver, on the other hand, lazily updates the sender's copy of the head pointer by writing the current head value via RDMA (❺). To minimize overhead, the receiver updates the sender's copy only after processing at least half of the buffer [19]. Consequently, the sender's head pointer always lags behind the receiver's, guaranteeing that unprocessed messages are never overwritten.

RTSFaaS employs a mechanism similar to the Chandy-Lamport algorithm [9] to ensure a consistent transition between the planning and execution phases for workers. In RTSFaaS, the driver maintains the global lease table, which records the current leaseholder for each data item. For each batch of requests, the driver assigns new leases for each data item and updates the lease table accordingly. After the driver has finished sending the requests for a batch, it broadcasts the updated global lease table along with a unique transition-flag message. Each worker then maintains a local copy of the lease table, synchronized with the driver's decisions, ensuring consistent lease ownership across all workers during the execution of the batch.

When workers receive requests, they first build a local TPG during the planning phase. Upon receiving the transition-flag and global lease table, workers collaboratively construct the global TPG. Once the global TPG is completed, workers move to the execution phase. The driver periodically injects the transition-flag to process requests in batches. The interval between injections is determined by the local cache capacity, as workers need to prefetch data items for each batch.

4.2 Fault Tolerance

For the storage layer, we rely on TiKV [39, 51]'s replication scheme for k-fault tolerance. In the computing tier, we adopt the standard fault tolerance approach commonly used in many FaaS platforms: if a machine fails, all machines reload the necessary data and re-execute the batch of requests to ensure that no task is lost. Once compute nodes have completed the transactions, their data caches are written back to the storage layer in batches, as discussed in Section 3.3. To ensure consistency and reliability during this write-back phase, we adopt a snapshot-based recovery mechanism combined with log-assisted tracking for efficient failure recovery.

After a batch of requests is completed, each compute node creates a snapshot of its local cache, which captures the latest state and is used for recovery in case of failures. Once all compute nodes successfully generate snapshots, they proceed to write back their local cache to TiKV [39, 51]. To assist the recovery, RTSFaaS maintains a lightweight log that tracks the progress of snapshot creation and write-back operations, recording the current operation and its status (e.g., "snapshot created," "write-back successfully"). If a failure occurs during snapshot creation, RTSFaaS reloads the original data for the failed nodes from TiKV [39, 51], re-executes the batch of requests, and regenerates the snapshot. For failures during the write-back process, RTSFaaS restores the failed node's local cache from its snapshot and reattempts the write-back.

This recovery mechanism ensures reliable failure handling in the compute layer. For example, when a write-back is completed before failure, RTSFaaS continues processing with zero downtime. If a failure occurs after snapshot creation but before write-back, the snapshot enables a retry within 233 ms. In cases of complete data loss, RTSFaaS can reload the original data, re-execute the batch, and regenerate the snapshot in 729 ms under a snapshot interval of 500 ms with a workload of 5,000 queries per second. In addition, the Docker startup and reconnection times for a cluster of one driver and four workers are 20.31s and 15.66s, which remains the dominant recovery overhead. Further improvements, such as RDMA-based optimizations used in MITOSIS [54], can be applied to reduce cold start latency.

4.3 Limitations

Handling Dependent Reads. Our lease assignment mechanism depends on knowing the full read/write set of each transaction, which becomes challenging in the presence of dependent reads. RTSFaaS follows a strategy similar to deterministic databases [35,50,65], where the driver performs early reads to resolve dependencies and determine the complete access set before assigning leases. While this approach ensures correctness, it introduces additional communication overhead due to the extra round-trip required for these early reads. Moreover, during the construction of the TPG, unresolved dependent reads are conservatively assumed to potentially interact with any prior or future operation. As a result, their metadata must be broadcast to all workers to capture possible global dependencies. This pessimistic strategy ensures correctness but incurs a non-negligible communication cost. Given the inherent difficulty of resolving dependencies for non-deterministic functions [41], developing more efficient solutions remains an important direction for future research.

Hotspot Contention on Remote Keys. Our affinity-aware lease assignment mechanism aims to reduce contention on remote keys by co-locating related functions and data. In large-scale deployments, it is still possible for tons of concurrent functions to acquire the lease on the same remote key. Our current design adopts a single logical leaseholder model per data item in the local data cache, which may become a bottleneck under high contention. Inspired by industrial systems such as PolarDB [8], one promising extension is to adopt a replicated-state model, where multiple read-only replicas are colocated with compute nodes to absorb read traffic, while a single primary replica coordinates updates. Such an approach could further improve scalability by mitigating lease contention hotspots.

5 Evaluation

In this section, we conduct a comprehensive evaluation of the design of RTSFaaS using a set of end-to-end experiments across several representative benchmarks. The source code is available on GitHub at https://github.com/CGCL-codes/RTSFaaS [42]. Our experimental studies aim to answer the following questions:

Q1: How well is the performance of RTSFaaS compared with the state-of-the-art transactional FaaS system using representative workloads? (Section 5.2)

Q2: Does the concurrency control protocol in RTSFaaS outperform those used in Boki [25] and Beldi [60] when all protocols are implemented in an RDMA-enabled environment, and how do varying workload characteristics influence their performance? (Section 5.3)

Q3: What is the performance overhead of RTSFaaS, and how different system configurations affect the performance of RTSFaaS? (Section 5.4)

5.1 Experimental Setup

We deploy RTSFaaS [42] in a cluster with five physical machines, each equipped with Intel Xeon Gold 6230 processors and 128 GB DDR-4 memory. These machines are interconnected via Mellanox ConnectX-3 40/56 GbE network controllers, delivering a round-trip time of 7 µs for an RDMA operation. Within the cluster, one machine is designated as the RTSFaaS driver, while the remaining four serve as compute nodes that execute serverless functions. The driver and each worker operate in individual Docker [18] containers, each configured with 8 CPUs and 32 GB of memory. Each worker reserves 2 GB memory used for the local data cache.

RTSFaaS use TiKV [39,51], a highly scalable, low-latency, and easy-to-use key-value database for database management. The TiKV cluster configuration consists of three placement drivers to manage the metadata and maintain consistent partitions across the database, along with three TiKV nodes to store and manage KV objects.

This setup reflects the two-tier data organization in RTSFaaS. The first tier consists of the local data cache maintained by compute nodes within the same data center to enable low-latency RDMA access, with each data item having a single in-memory replica. The second tier is the TiKV-backed storage, which supports geo-distributed replication mechanisms for high availability.

5.2 Comparing to Transactional FaaS Systems

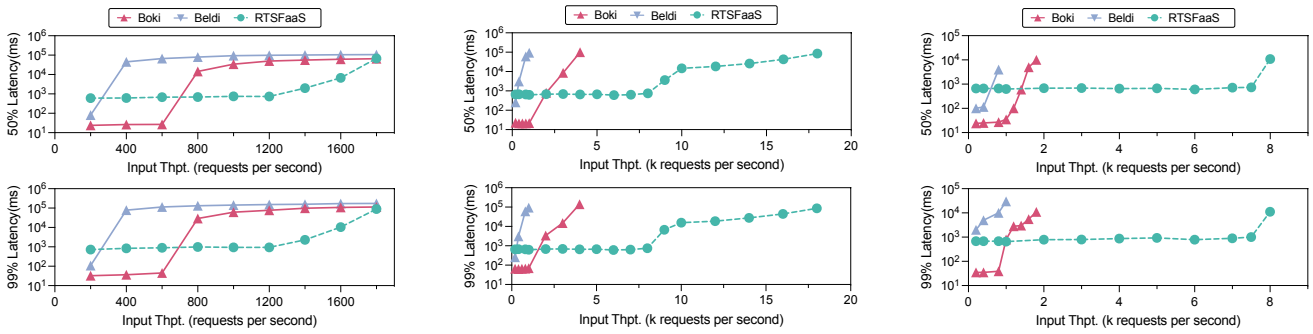
5.2.1 Baselines

In this evaluation, we compare RTSFaaS with the following stateful FaaS systems that guarantee transactional consistency.

Beldi [60] is a recent transactional FaaS system that leverages Nightcore [26] as its function runtime and uses a *two-phase locking* (2PL) protocol to ensure serializable consistency. In this experiment, Beldi [60] uses DynamoDB as its storage backend and operates on 8 function nodes.

Boki [25] adopts the *optimistic concurrency control* (OCC) protocol, leveraging a log-based conflict window for conflict validation. Boki [25] also uses a local cache on compute nodes to enhance read locality. We follow the experimental setup described in the Boki paper [25] to deploy 3 storage nodes, 3 sequence nodes, and 8 worker nodes.

For both Beldi [60] and Boki [25], each node runs on a c5d.2xlarge instance with 8 vCPUs, 16GB of DRAM, and hyper-threading enabled. The round-trip time between VMs ranges from 100 to 120 microseconds. To match these baseline conditions, we configure RTSFaaS with four workers, each of them provisioned with 8 cores and 32 GB of DRAM. In this experiment, we fix the batch interval at 500 ms to assess RTSFaaS's performance under stable conditions. We further explore the effects of varying batch sizes in Section 5.4.



(a) Movie Review (b) Travel Reservation (c) Bank Service

Figure 9: Latency vs. throughput for RTSFaaS, Boki, and Beldi on movie review, travel reservation, and banking service

5.2.2 Workloads

To make a comprehensive evaluation, we adapt and extend the following workloads from prior works [23, 25, 60].
1) The *Movie Review* workload simulates a movie review service where users write reviews for movies.
2) The *Travel Reservation* workload simulates a travel reservation service and consists of two main workflows that we implement: the search workflow, where users read and sort hotel and flight options by criteria such as rating and price, and the reservation workflow, which involves writing reservation details to secure hotel rooms and flights. These workflows are executed independently, maintaining a balanced 1:1 ratio between search and reservation requests.
3) The *Banking Service* workload simulates transfer operations between users. The withdraw function first reads the balance of the source account to verify sufficient funds, then writes the new balance by deducting the specified amount. Upon successful completion of the withdrawal, the deposit function is invoked to read the balance of the target account and update it by adding the specified amount.

5.2.3 Latency vs. Input Throughput

This section evaluates how the input throughput impacts latency for Boki [25], Beldi [60], and RTSFaaS across multiple workloads. Specifically, input throughput is defined as the rate of incoming requests per second to each system under test, while latency represents the total time taken from a user's request submission to the final receipt of execution results. As shown in Figure 9, RTSFaaS demonstrates superior performance compared to Boki [25] and Beldi [60] by maintaining consistently lower latency across increasing input throughput. When the median latency is 700 ms, RTSFaaS improves throughput by 2.0× (6.0×) in movie review workload, 4.0× (20×) in travel reservation workload, and 5.0× (17×) in banking service workload compared to Boki [25] (Beldi [60]).

Beldi [60] experiences higher latency even at lower input throughputs compared to Boki [25] and RTSFaaS because transactions may need to wait for locks. Since Beldi [60] uses remote locks, the overhead of acquiring and releasing locks is inherently higher due to network communication. Benefiting from the local cache and OCC, Boki [25] starts with lower latency at low input throughput because conflicts are infrequent. However, as the input throughput increases, more conflicts are detected during the commit phase, leading to more transaction rollbacks and retries, which could explain the steeper rise in both median and tail latency.

In comparison, RTSFaaS proposes a lease-based concurrency control mechanism that optimizes data locality and executes transactions without any concurrency-related aborts. Thus, RTSFaaS outperforms the other two systems as input throughput increases. Notably, RTSFaaS also exhibits constant latency even when the input throughput is low. This is because RTSFaaS commits a batch every 500ms, which introduces a fixed waiting time regardless of the input throughput.

5.3 RDMA-enable CC Mechanisms

In this section, we evaluate the effectiveness of RTSFaaS by comparing it to concurrency control protocols that rely on remote storage, as used in Boki [25] and Beldi [60]. To ensure a fair comparison and isolate the impact of network performance, we reimplement the concurrency control protocols from these platforms in an RDMA-enabled environment.
Approaches for Comparison. We compare RTSFaaS with
1) **Remote Lock:** Belid [60] ensures transactional consistency based on a variant of 2PL with wait-die lock. We enhance it by using RDMA atomic operations [59], such as *compare-and-swap* (CAS) and *fetch-and-add* (FAA) to implement lock management, including exclusive lock and share lock [10].
2) **Remote OCC + Local Cache**: As mentioned before, Boki [25] supports transactional consistency based on *optimistic concurrency control* (OCC). In addition to enhancing lock management with RDMA [53], we also implement local data caching. Thus, transactions can read and modify the local cache optimistically but must check whether any conflicting operations occur before committing transactions.

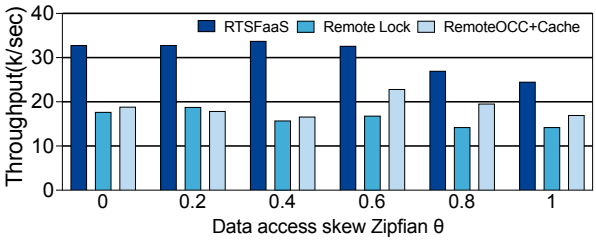


Figure 10: Impact of state access skewness

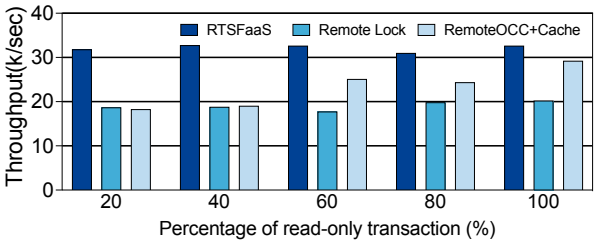


Figure 11: Impact of read-only transactions

Microbenchmark and Testbed Setup. We use a microbenchmark that reads or updates the values associated with a set of keys. The dataset contains 20,000 items, and we vary the data access skew, percentage of read-only transactions, and transactional workflow length to conduct a comprehensive evaluation. This section uses a simple RDMA-based key-value store to implement traditional concurrency control protocols. We evaluate RTSFaaS and other approaches using four workers, each equipped with twenty cores and 64 GB DRAM.

5.3.1 Impact of Data Access Skew

In this evaluation, we compare the throughput performance of RTSFaaS, remote lock, and the remote optimistic concurrency control under varying data access skew (Zipfian θ). To isolate the impact of data access skew, we configure the microbenchmark to consist entirely of write transactions, each with a transaction length of two. As shown in Figure 10, RTSFaaS consistently achieves the highest throughput across all skew levels. This robust performance is due to RTSFaaS's separation of consistency guarantees from execution, which reduces the overhead of remote locking during execution. Additionally, the affinity-aware lease assignment mechanism optimizes data locality, further enhancing throughput.

In contrast, the remote lock protocol consistently delivers the lowest throughput, primarily due to the high overhead associated with lock management. The throughput of using optimistic concurrency control exhibits a two-phase performance trend: initially, as the skew increases, its performance improves due to the effectiveness of caching, reaching around 23k/sec at θ=0.6. However, as the skew continues to increase, the performance declines sharply to about 16k/sec at θ=1. This decline is caused by higher contention, which leads to increased aborts and retries. As a result, the cache becomes less effective, further reducing performance.

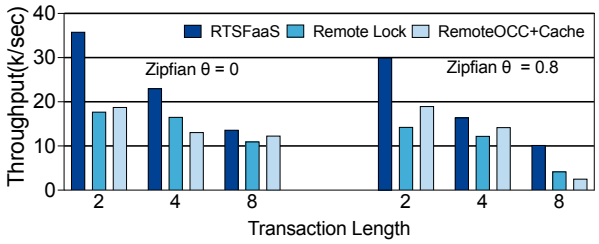


Figure 12: Impact of transactional workflow length

5.3.2 Impact of Read-Only Transactions

Figure 11 illustrates the throughput of RTSFaaS, remote lock, and the remote optimistic concurrency control as the percentage of read-only transactions changes. The remote optimistic concurrency control shows better performance at higher read-only transaction percentages due to effective caching. However, as the read-only percentage decreases, its performance drops because frequent non-local reads are needed to update the cache in the presence of writes, increasing contention and reducing cache effectiveness.

In contrast, RTSFaaS consistently outperforms other mechanisms, especially when the percentage of read-only transactions is low. This is because RTSFaaS improves data locality by assigning groups of interdependent data items to a single worker under the exclusive lease. Additionally, RTSFaaS allows workers to access each other's local caches directly using one-sided RDMA verbs, thereby avoiding the overhead of contacting the storage multiple times to obtain a fresh version. Since RTSFaaS processes read and write operations in the order of their dependencies, its performance advantage remains stable as the percentage of read-only transactions increases.

5.3.3 Impact of Transactional Workflow Length

As shown in Figure 12, we compare throughput across various transactional workflow lengths, which is defined as the number of atomic functions within a single workflow. As the transactional workflow length increases, the performance of all mechanisms declines. This performance decrease is attributed to the increased number of operations each request needs to process, which naturally leads to lower throughput.

For the remote optimistic concurrency control, throughput declines sharply with longer transactional workflows, especially at Zipfian $\theta = 0.8$. The decline is caused by the increased probability of conflicts as the length of each transactional workflow grows, which raises the likelihood of aborts and re-executions during the validation phase. The aborts and re-executions lead to wasted computational resources and a significant reduction in the effectiveness of caching. Similarly, remote lock consistently shows lower performance due to the increased lock contention in longer transactional workflows, which leads to more frequent remote lock acquisitions and increased communication overhead.

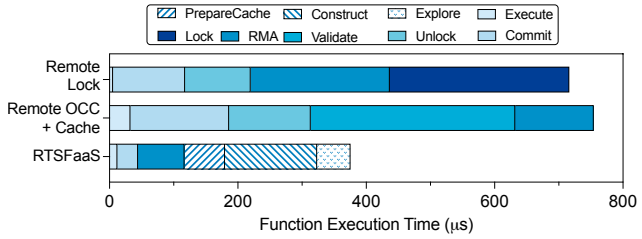


Figure 13: Function execution time breakdown

In contrast, RTSFaaS achieves superior throughput compared to others by serializing function execution through a global TPG, which significantly reduces the network communication overhead associated with lock operations.

5.4 Performance Influencing Factors

5.4.1 Factor Analysis

To further investigate the performance differences between RTSFaaS and directly implement the concurrency control protocols used by Boki [25] and Beldi [60] in the RDMA-enabled environment, this section decomposes the execution time of the function to perform a factor analysis. As depicted in Figure 13, both the remote lock and optimistic concurrency control protocols incur substantial overhead due to frequent interactions with remote storage, including operations such as locking, unlocking, and validation. This result highlights that merely adding RDMA on top of existing protocols designed for transactional consistency does not effectively alleviate the overhead of remote communication. RTSFaaS spends a substantial portion of time on the construction and exploration phases, where it constructs the global TPG and explores this graph to select functions for execution. Despite this additional task, RTSFaaS significantly reduces the communication overhead associated with frequent interactions with remote storage. Thus, RTSFaaS presents an overall improvement in performance compared to the other two protocols.

5.4.2 Impact of Batch Size

In this section, we evaluate the impact of batch size on the performance of RTSFaaS. The batch size, defined as the number of workflows handled between two consecutive punctuations, plays a critical role in RTSFaaS's performance. Figure 14 shows the throughput and latency of RTSFaaS under increasing batch size. The results indicate that throughput initially increases with the batch size, reaching a plateau at $n = 25600$ when the computational resources on each machine are fully utilized. However, this improvement comes at the cost of higher latency, particularly at the 99th percentile. This is because, in RTSFaaS, the function executor must wait for all requests in a batch to arrive before constructing the global TPG, which is done sequentially based on dependencies. As a result, earlier requests in the batch have to wait for later ones

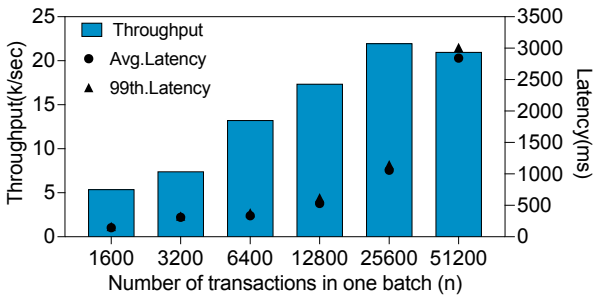


Figure 14: Effects of batch size on performance of RTSFaaS

to arrive, delaying the execution of earlier requests. Additionally, as the batch size increases, the time required to construct the global TPG also increases, further contributing to higher waiting times and higher latency at the 99th percentile.

6 Related Work

Stateful Serverless Computing. As growing data-centric applications are apt to deploy in cloud data centers, both industry and academia have made significant efforts to enhance the performance and functionality of FaaS platforms. Various stateful FaaS platforms have been developed, some offering weak consistency [34, 48, 57], while others incur high-performance overhead [25,30,60] to ensure transactional consistency. Apiary [30] co-locates function logic and data management by wrapping a distributed transactional database management system like VoltDB [14], while Styx [40] co-locates the function execution with application states using a streaming dataflow execution model. Although Apiary [30] and Styx [40] support transactional consistency, they tightly couple storage and computation, making their system architectures fundamentally different from RTSFaaS. Other works, such as Shredder [64], propose cloud datastores that allow functions to execute directly on storage nodes. LambdaObjects [36] co-locates storage and computing by encapsulating data as objects tied to specific functions, inspired by object-oriented principles. However, none of these solutions supports serializable transactions with multiple functions. In contrast, RTSFaaS supports transactional stateful serverless workflows, and further exploits a lease-based concurrency control protocol to achieve high performance.

RDMA-capable Serverless Computing. RDMA technologies offer vast opportunities in the realm of serverless computing. Recent RDMA-based FaaS frameworks have different motivations, including remote fork [54], function scheduling [13], and state transfer [33]. MITOSIS [54] introduces an operating system primitive that aims to facilitate rapid remote fork by integrating the RDMA driver into the OS kernel. Similarly, rFaaS [13] introduces new RDMA abstractions that simplify the network stack for serverless computing, along with a decentralized resource allocation mechanism, which

allows function invocations at the microsecond level. In addition, RMMAP [33] proposes a new OS primitive that maps a remote memory region into a local process using RDMA, and thus can streamline the process of transferring states between any pair of functions, thereby eliminating the need for serialization and deserialization. Unlike these proposes that incorporate RDMA into serverless computing, our work attempts to address a different challenge. RTSFaaS focuses on improving transactional stateful serverless workflows to achieve high performance and strong consistency.

RDMA-enabled KV Stores and Transactional Storage. Existing RDMA-enabled KV stores, such as ROLEX [31] and LOLKV [2], provide strong consistency for individual operations, particularly through RDMA-friendly replication protocols. Thus, these RDMA-based key-value stores cannot be directly applied to FaaS platforms because FaaS requires transactional consistency for workflows, often involving multiple interdependent functions.

RDMA-enabled transactional storage systems exhibit a significant performance improvement for transaction processing [19, 29, 53, 62, 63]. FaRM [19], FaSST [29], DrTM [55], and DrTM-H [53] can support strong consistency for distributed transactions via 2PC/OCC RDMA-based approach. They are effective in monolithic architectures, whereas RTSFaaS is designed for the disaggregated architecture widely adopted in FaaS platforms. More importantly, RTSFaaS reduces remote communication overhead by employing a lease-based concurrency control protocol at the computation layer.

In addition, FORD [63] and Motor [62] further optimize distributed transactions through advanced techniques in disaggregated memory environments. FORD [63] accelerates distributed transactions by utilizing fast one-sided RDMA operations and consolidating operations such as reads and locks into a single RTT. On the other hand, Motor [62] leverages the *consecutive version tuple* (CVT) and one-sided RDMA operations to efficiently support multi-version distributed transactions, and thus reduces network round trips and improves concurrency. While FORD [63] and Motor [62] optimize the performance of individual transactions, they primarily focus on enhancing single RDMA operations. In contrast, transactional workflows involve a sequence of interdependent operations, where the outcome of one function may affect others. These transactional workflows should guarantee strong consistency and coordinate all functions. RTSFaaS addresses this problem by resolving the dependencies among chained functions, and guaranteeing transactional consistency via dynamic lease assignment and transfer, rather than just optimizing individual transactions.

To address communication overhead in RDMA environments, recent efforts such as ALock [3] enable local memory access without relying on RPCs or loopback, significantly improving synchronization efficiency. While ALock [3] targets fine-grained synchronization for low-level memory access, our lease mechanism emphasizes enhancing caching efficiency and managing dependencies among concurrent transactional workflows. Nevertheless, integrating ALock [3] into our dynamic lease transfer protocol could further improve coordination efficiency between local and remote memory accesses in disaggregated FaaS systems.

Deterministic Transactional System. Deterministic systems such as Calvin [50] avoid coordination overhead by predefining a global transaction execution order, thereby eliminating the need for distributed locks or optimistic validation. However, these approaches face limitations when applied to the disaggregated, dynamic nature of FaaS platforms. Specifically, Calvin [50] focuses on read-write conflicts over the same key, whereas transactional workflows in FaaS often involve complex data dependencies, such as parametric dependencies between functions, which may span multiple keys and require fine-grained coordination. Moreover, deterministic systems typically rely on static data partitioning, which constrains their ability to adapt to runtime function placement and workload variations. In contrast, our system introduces an affinity-aware lease assignment mechanism that dynamically adapts data ownership and placement according to function-data affinity, improving both load balancing and execution locality in distributed FaaS environments.

7 Conclusion

In this paper, we present RTSFaaS, an RDMA-capable transactional stateful FaaS framework that achieves high performance while guaranteeing transactional consistency. RTSFaaS exploits a lease-based concurrency control protocol to dynamically assign and transfer leases among workers to achieve concurrency control. Specifically, RTSFaaS incorporates two key designs to address the aforementioned limitations of current systems: an affinity-aware lease assignment mechanism and an RDMA-capable dynamic lease transferring mechanism. Experimental results show that RTSFaaS outperforms the state-of-the-art stateful FaaS platforms, achieving up to 5× and 20× higher throughput than Boki [25] and Beldi [60], respectively. Even when their concurrency control protocols are adapted to RDMA networks and integrated into RTSFaaS, our design still yields up to 1.7× and 2.1× performance improvements.

Acknowledgments

We appreciate our shepherd, Mr. Haonan Lu, and the anonymous reviewers' constructive comments for improving the quality of this paper. This work is supported jointly by the National Key Research and Development Program of China under grant 2022YFB4500303, National Natural Science Foundation of China (NSFC) under grants 62332011, 62302178, NSFC-RGC under grant 62461160333, and Natural Science Foundation of Hubei Province under grant 2021CFA037.

References

[1] Atul Adya, Robert Gruber, Barbara Liskov, and Umesh Maheshwari. Efficient Optimistic Concurrency Control Using Loosely Synchronized Clocks. *ACM SIGMOD Record*, 24(2):23–34, 1995.

[2] Ahmed Alquraan, Sreeharsha Udayashankar, Virendra Marathe, Bernard Wong, and Samer Al-Kiswany. LoLKV: The Logless, Linearizable, RDMA-based Key-Value Storage System. In *Proceedings of the 21st USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 41–54, 2024.

[3] Amanda Baran, Jacob Nelson-Slivon, Lewis Tseng, and Roberto Palmieri. ALock: Asymmetric Lock Primitive for RDMA Systems. In *Proceedings of the 36th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 15–26, 2024.

[4] Daniel Barcelona-Pons, Marc Sánchez-Artigas, Gerard París, Pierre Sutra, and Pedro García-López. On the FaaS Track: Building Stateful Distributed Applications with Serverless Architectures. In *Proceedings of the 20th International Middleware Conference (Middleware)*, pages 41–54, 2019.

[5] Philip A. Bernstein, David W. Shipman, and Wing S. Wong. Formal Aspects of Serializability in Database Concurrency Control. *IEEE Transactions on Software Engineering*, 5(3):203–216, 1979.

[6] Carsten Binnig, Andrew Crotty, Alex Galakatos, Tim Kraska, and Erfan Zamanian. The End of Slow Networks: It's Time for a Redesign. *Proceedings of the VLDB Endowment*, 9(7):528–539, 2016.

[7] Qingchao Cai, Wentian Guo, Hao Zhang, Divyakant Agrawal, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, Yong Meng Teo, and Sheng Wang. Efficient Distributed Memory Management with RDMA and Caching. *Proceedings of the VLDB Endowment*, 11(11):1604–1617, 2018.

[8] Wei Cao, Yingqiang Zhang, Xinjun Yang, Feifei Li, Sheng Wang, Qingda Hu, Xuntao Cheng, Zongzhi Chen, Zhenjun Liu, Jing Fang, Bo Wang, Yuhui Wang, Haiqing Sun, Ze Yang, Zhushi Cheng, Sen Chen, Jian Wu, Wei Hu, Jianwei Zhao, Yusong Gao, Songlu Cai, Yunyang Zhang, and Jiawang Tong. PolarDB Serverless: A Cloud Native Database for Disaggregated Data Centers. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD)*, pages 2477–2489, 2021.

[9] K. Mani Chandy and Leslie Lamport. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Transactions on Computer Systems*, 3(1):63–75, 1985.

[10] Yeounoh Chung and Erfan Zamanian. Using RDMA for Lock Management. *arXiv preprint arXiv:1507.03274*, 2015.

[11] Alibaba Cloud. Function Compute. https://www.alibabacloud.com/en/product/function-compute [Accessed: April 16, 2024].

[12] Knative Community. Knative is an Open-Source Enterprise-level Solution to Build Serverless and Event Driven Applications. https://knative.dev/docs/ [Accessed: April 16, 2024].

[13] Marcin Copik, Konstantin Taranov, Alexandru Calotoiu, and Torsten Hoefler. rFaaS: Enabling High Performance Serverless with RDMA and Leases. In *Proceedings of the 2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 897–907, 2023.

[14] Volt Active Data. VoltDB. https://www.voltdb.com/ [Accessed: April 16, 2024].

[15] Martijn de Heus, Kyriakos Psarakis, Marios Fragkoulis, and Asterios Katsifodimos. Distributed Transactions on Serverless Stateful Functions. In *Proceedings of the 15th ACM International Conference on Distributed and Event-based Systems (DEBS)*, pages 31–42, 2021.

[16] Bonaventura Del Monte, Steffen Zeuch, Tilmann Rabl, and Volker Markl. Rethinking Stateful Stream Processing with RDMA. In *Proceedings of the 2022 International Conference on Management of Data (SIGMOD)*, pages 1078–1092, 2022.

[17] HUAWEI Developers. Cloud Functions Documents. https://developer.huawei.com/consumer/en/agconnect/cloud-function/ [Accessed: April 16, 2024].

[18] Docker. Develop Faster. Run Anywhere. https://www.docker.com [Accessed: April 16, 2024].

[19] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. FaRM: Fast Remote Memory. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 401–414, 2014.

[20] Mostafa Elhemali, Niall Gallagher, Nick Gordon, Joseph Idziorek, Richard Krog, Colin Lazier, Erben Mo, Akhilesh Mritunjai, Somasundaram Perianayagam, Tim Rath, Swami Sivasubramanian, James Christopher Sorenson III, Sroaj Sosothikul, Doug Terry, and Akshat Vig. Amazon DynamoDB: A Scalable, Predictably Performant, and Fully Managed NoSQL Database Service. In *Proceedings of the 2022 USENIX Annual Technical Conference (USENIX ATC)*, pages 1037–1048, 2022.

[21] Kapali P. Eswaran, Jim N. Gray, Raymond A. Lorie, and Irving L. Traiger. The Notions of Consistency and Predicate Locks in a Database System. *Communications of the ACM*, 19(11):624–633, 1976.

[22] Apache Software Foundation. Open Source Serverless Cloud Platform. https://openwhisk.apache.org/ [Accessed: April 16, 2024].

[23] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, and Christina Delimitrou. An Open-source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 3–18, 2019.

[24] Joseph M. Hellerstein, Jose Faleiro, Joseph E. Gonzalez, Johann Schleier-Smith, Vikram Sreekanti, Alexey Tumanov, and Chenggang Wu. Serverless Computing: One Step Forward, Two Steps Back. *arXiv preprint arXiv:1812.03651*, 2018.

[25] Zhipeng Jia and Emmett Witchel. Boki: Stateful Serverless Computing with Shared Logs. In *Proceedings of the 28th Symposium on Operating Systems Principles (SOSP)*, pages 691–707, 2021.

[26] Zhipeng Jia and Emmett Witchel. Nightcore: Efficient and Scalable Serverless Computing for Latency-Sensitive, Interactive Microservices. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 152–166, 2021.

[27] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Yadwadkar, Joseph E. Gonzalez, Raluca Ada Popa, Ion Stoica, and David A. Patterson. Cloud Programming Simplified: A Berkeley View on Serverless Computing. *arXiv preprint arXiv:1902.03383*, 2019.

[28] Anuj Kalia, Michael Kaminsky, and David G. Andersen. Design Guidelines for High Performance RDMA Systems. In *Proceedings of the 2016 USENIX Annual Technical Conference (USENIX ATC)*, pages 437–450, 2016.

[29] Anuj Kalia, Michael Kaminsky, and David G. Andersen. FaSST: Fast, Scalable and Simple Distributed Transactions with Two-Sided RDMA Datagram RPCs. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 185–201, 2016.

[30] Peter Kraft, Qian Li, Kostis Kaffes, Athinagoras Skiadopoulos, Deeptaanshu Kumar, Danny Cho, Jason Li, Robert Redmond, Nathan Weckwerth, Brian Xia, Peter Bailis, Michael Cafarella, Goetz Graefe, Jeremy Kepner, Christos Kozyrakis, Michael Stonebraker, Lalith Suresh, Xiangyao Yu, and Matei Zaharia. Apiary: A DBMS-Backed Transactional Function-as-a-Service Framework. *arXiv preprint arXiv:2208.13068*, 2022.

[31] Pengfei Li, Yu Hua, Pengfei Zuo, Zhangyu Chen, and Jiajie Sheng. ROLEX: A Scalable RDMA-oriented Learned Key-Value Store for Disaggregated Memory Systems. In *Proceedings of the 21st USENIX Conference on File and Storage Technologies (FAST)*, pages 99–114, 2023.

[32] Zijun Li, Chuhao Xu, Quan Chen, Jieru Zhao, Chen Chen, and Minyi Guo. DataFlower: Exploiting the Dataflow Paradigm for Serverless Workflow Orchestration. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 57–72, 2023.

[33] Fangming Lu, Xingda Wei, Zhuobin Huang, Rong Chen, Minyu Wu, and Haibo Chen. Serialization/Deserialization-Free State Transfer in Serverless Workflows. In *Proceedings of the Nineteenth European Conference on Computer Systems (EuroSys)*, pages 132–147, 2024.

[34] Taras Lykhenko, Rafael Soares, and Luis Rodrigues. FaaSTCC: Efficient Transactional Causal Consistency for Serverless Computing. In *Proceedings of the 22nd International Middleware Conference (Middleware)*, pages 159–171, 2021.

[35] Yancan Mao, Jianjun Zhao, Shuhao Zhang, Haikun Liu, and Volker Markl. MorphStream: Adaptive Scheduling for Scalable Transactional Stream Processing on Multicores. In *Proceedings of the 2023 International Conference on Management of Data (SIGMOD)*, pages 1–26, 2023.

[36] Kai Mast, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. LambdaObjects: Re-Aggregating Storage and Execution for Cloud Computing. In *Proceedings of the 14th ACM Workshop on Hot Topics in Storage and File Systems (HotStorage)*, pages 15–22, 2022.

[37] Microsoft. Azure Durable Functions Documentation. https://docs.microsoft.com/en-us/azure/azure-functions/durable/

durable-functions-overview [Accessed: April 16, 2024].

[38] Christos H. Papadimitriou. The Serializability of Concurrent Database Updates. *Journal of the ACM*, 26(4):631–653, 1979.

[39] PingCAP. TiKV. https://tikv.org/ [Accessed: April 16, 2024].

[40] Kyriakos Psarakis, George Christodoulou, George Siachamis, Marios Fragkoulis, and Asterios Katsifodimos. Styx: Transactional Stateful Functions on Streaming Dataflows. *arXiv preprint arXiv:2312.06893*, 2023.

[41] Kun Ren, Alexander Thomson, and Daniel J. Abadi. An Evaluation of the Advantages and Disadvantages of Deterministic Database Systems. *Proceedings of the VLDB Endowment*, 7(10):821–832, June 2014.

[42] RTSFaaS, 2025. https://github.com/CGCL-codes/RTSFaaS.

[43] Amazon Web Services. Amazon DynamoDB. https://aws.amazon.com/dynamodb [Accessed: April 16, 2024].

[44] Amazon Web Services. Amazon Simple Storage Service (Amazon S3). https://aws.amazon.com/s3 [Accessed: April 16, 2024].

[45] Amazon Web Services. AWS Step Functions. https://aws.amazon.com/step-functions/ [Accessed: April 16, 2024].

[46] Amazon Web Services. Serverless Computing - AWS Lambda. https://aws.amazon.com/pm/lambda/ [Accessed: April 16, 2024].

[47] Jiacheng Shen, Pengfei Zuo, Xuchuan Luo, Tianyi Yang, Yuxin Su, Yangfan Zhou, and Michael R. Lyu. FUSEE: A Fully Memory-Disaggregated Key-Value Store. In *Proceedings of the 21st USENIX Conference on File and Storage Technologies (FAST)*, pages 81–98, 2023.

[48] Vikram Sreekanti, Chenggang Wu, Xiayue Charles Lin, Johann Schleier-Smith, Jose M. Faleiro, Joseph E. Gonzalez, Joseph M. Hellerstein, and Alexey Tumanov. Cloudburst: Stateful Functions-as-a-Service. *Proceedings of the VLDB Endowment*, 13(12):2438–2452, 2020.

[49] Alexander Thomasian. Two-Phase Locking Performance and its Thrashing Behavior. *ACM Transactions on Database Systems*, 18(4):579–625, 1993.

[50] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. Calvin: Fast Distributed Transactions for Partitioned Database Systems. In *Proceedings of the 2012 International Conference on Management of Data (SIGMOD)*, pages 1–12, 2012.

[51] Yangyang Wang, Zikai Wang, Yunpeng Chai, and Xin Wang. Rethink the Linearizability Constraints of Raft for Distributed Key-Value Stores. In *Proceedings of the 2021 IEEE 37th International Conference on Data Engineering (ICDE)*, pages 1877–1882, 2021.

[52] Xingda Wei, Rong Chen, and Haibo Chen. Fast RDMA-based Ordered Key-Value Store Using Remote Learned Cache. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 117–135, 2020.

[53] Xingda Wei, Zhiyuan Dong, Rong Chen, and Haibo Chen. Deconstructing RDMA-enabled Distributed Transactions: Hybrid is Better! In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 233–251, 2018.

[54] Xingda Wei, Fangming Lu, Tianxia Wang, Jinyu Gu, Yuhan Yang, Rong Chen, and Haibo Chen. No Provisioned Concurrency: Fast RDMA-Codesigned Remote Fork for Serverless Computing. In *Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 497–517, 2023.

[55] Xingda Wei, Jiaxin Shi, Yanzhe Chen, Rong Chen, and Haibo Chen. Fast In-memory Transaction Processing Using RDMA and HTM. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP)*, pages 87–104, 2015.

[56] Gerhard Weikum and Gottfried Vossen. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.

[57] Chenggang Wu, Vikram Sreekanti, and Joseph M. Hellerstein. Transactional Causal Consistency for Serverless Computing. In *Proceedings of the 2020 International Conference on Management of Data (SIGMOD)*, pages 83–97, 2020.

[58] Chang Yao, Divyakant Agrawal, Gang Chen, Qian Lin, Beng Chin Ooi, Weng-Fai Wong, and Meihui Zhang. Exploiting Single-Threaded Model in Multi-Core In-Memory Systems. *IEEE Transactions on Knowledge and Data Engineering*, 28(10):2635–2650, 2016.

[59] Dong Young Yoon, Mosharaf Chowdhury, and Barzan Mozafari. Distributed Lock Management with RDMA: Decentralization without Starvation. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD)*, pages 1571–1586, 2018.

[60] Haoran Zhang, Adney Cardoza, Peter Baile Chen, Sebastian Angel, and Vincent Liu. Fault-Tolerant and Transactional Stateful Serverless Workflows. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 1187–1204, 2020.

[61] Haoran Zhang, Shuai Mu, Sebastian Angel, and Vincent Liu. CausalMesh: A Causal Cache for Stateful Serverless Computing. *Proceedings of the VLDB Endowment*, 17(13):4599–4613, 2024.

[62] Ming Zhang, Yu Hua, and Zhijun Yang. Motor: Enabling Multi-Versioning for Distributed Transactions on Disaggregated Memory. In *Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 801–819, 2024.

[63] Ming Zhang, Yu Hua, Pengfei Zuo, and Lurong Liu. FORD: Fast One-sided RDMA-based Distributed Transactions for Disaggregated Persistent Memory. In *Proceedings of the 20th USENIX Conference on File and Storage Technologies (FAST)*, pages 51–68, 2022.

[64] Tian Zhang, Dong Xie, Feifei Li, and Ryan Stutsman. Narrowing the Gap Between Serverless and its State with Storage Functions. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*, pages 1–12, 2019.

[65] Jianjun Zhao, Haikun Liu, Shuhao Zhang, Zhuohui Duan, Xiaofei Liao, Hai Jin, and Yu Zhang. Fast Parallel Recovery for Transactional Stream Processing on Multicores. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*, pages 1478–1491. IEEE, 2024.

[66] Zhuangzhuang Zhou, Yanqi Zhang, and Christina Delimitrou. AQUATOPE: QoS-and-Uncertainty-Aware Resource Management for Multi-stage Serverless Workflows. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 1–14, 2023.

[67] Tobias Ziegler, Jacob Nelson-Slivon, Viktor Leis, and Carsten Binnig. Design Guidelines for Correct, Efficient, and Scalable Synchronization Using One-Sided RDMA. In *Proceedings of the 2023 International Conference on Management of Data (SIGMOD)*, pages 1–26, 2023.

Swift: Fast Performance Tuning with GAN-Generated Configurations

Chao Chen[1], Shixin Huang[1], Xuehai Qian[2], and Zhibin Yu[1,3]

[1]Shenzhen Institutes of Advanced Technology, Chinese Academy of Sciences
[2]Tsinghua University
[3]Shuhai Lab, Huawei Cloud

Abstract

This paper proposes *Swift*, a novel Bayesian Optimization (BO) based parameter configuration tuning approach for big data systems. The key idea is to leverage a generative AI approach, generative adversarial network (GAN) , to generate high quality configurations based on the evaluated configuration with the highest performance. Mixing these configurations with randomly generated ones has the effect of skewing search space toward the optimal configuration, leading to faster convergence and less optimization time. Our substantial experimental results on Apache Flink, Spark programs, and an industrial setting show that *Swift* significantly improves the performance of data analytics over state-of-the-art approaches in dramatically shorter time.

1 Introduction

To support a wide range of data analytics applications, frameworks such as Apache Spark [48] and Flink [45] typically provide more than one hundred configuration parameters. It is critical to set these parameters properly to achieve high performance. For instance, the Spark parameter `memory.fraction` [49] specifies the heap memory fraction for task execution and data storage. Its value significantly affects the performance of a program. A recent study [64] showed that tuning *multiple* parameters simultaneously can further improve program performance by up to 89×.

Unfortunately, tuning the configuration parameters to optimize performance is extremely challenging for two reasons. First, *the tuning space is vast* due to the large number of parameters and even larger range of values each parameter can take. For example, prior studies [4, 64] showed that there are 34 and 41 performance-critical parameters for Hadoop and Spark, respectively. Second, these parameters may *interact* with each other in a complex (e.g., non-linear) manner [1, 4, 8, 14, 31, 35, 64].

Due to the two challenges, traditional approaches for parameter tuning are not effective. The search approaches based on rules [60, 61] and analytical models [20] do not work well due to the over-simplified assumptions on parameter interactions. On the other side, approaches based on simulation [33, 38, 55, 56] require many time-consuming simulations to find the optimized configuration, even for a single program. These limitations gave raise to the machine learning based (ML-based) approaches [1, 2, 35, 64] with significantly higher speedups over rule-based alternatives [60, 61].

ML-based approaches directly learn from *a large number of experiments*. However, it is *time-consuming to collect samples by executing a program with a configuration on a real cluster*. Indeed, the state-of-the-art ML-based approaches such as OPPerTune [42], SelfTune [26], PARIS [62, 63] and Quasar [10] do suffer from this drawback, e.g., SelfTune and OPPerTune collect 756 and 252 training samples for a single workload [26, 42].

Bayesian Optimization (BO) has been demonstrated to require the fewest number of iterations to converge [6]. For this reason, BO is adopted in highly efficient ML-based approaches such as CherryPick [2] as the search algorithm. Even though BO requires fewer iterations, evaluating the configuration in each iteration by executing the program on a real cluster would typically take at least minutes [64].

To make BO converge faster and more smoothly, we utilize a **novel approach**: generating the *likely high quality configurations that lead to high performance* for BO algorithm in each iteration, which leads to the **key contribution** of the paper: we propose a method named *Swift* to use Generative Adversarial Network (GAN) [16] to generate configurations from the target configuration, so that they share similar distribution.

To generate a high quality configuration, the goal is to ensure that the configuration itself and its corresponding performance are *"similar but not too similar"* to the target one. The generated configuration and the target configuration can be regarded as two equal-length vectors. Intuitively, the performance of a program should be similar if the distance between the two vectors is short. However, we show that this intuition is not true by the following experiments.

We first explore an intuitive and simple approach which is named *One-Neighbor (ON)*. Essentially, ON uses the Manhattan distance to capture the similarity between configurations (vectors). However, we observed that configurations with similar vector distances to the target one can exhibit either similar or significantly different performance, rendering the approach ineffective for configuration generation. This led to our **key insight**: producing the *"similar but not too similar"* performance requires NOT ONLY a short distance between the two vectors BUT ALSO similar distributions of their element values. To this end, we identified Jensen-Shannon (JS) divergence [13] as the ideal metric, as it is a widely used method for quantifying the similarity between two probability distributions. By mathematic definition, JS divergence ranges from 0.0 to 1.0, and if a distribution is more similar to another one, the smaller JS divergence is. Notably, GANs inherently use JS divergence as a loss function to ensure generated images resemble input images. Although our work focuses on one-dimensional configuration vectors rather than images, we adapted this principle to our domain where training GANs is simple as well as fast.

Although leveraging GAN appears intuitive, its integration into BO optimization is not trivial and needs to be carefully designed. Overall, *Swift* works as follows. First, the GAN needs to be continuously trained by the "current best" evaluated and real sample with a noise that obeys a Gaussian distribution for each parameter. In each iteration, *Swift* uses GAN to generate 150 configurations in addition to 100,000 ones generated randomly. Thus, the pool that BO's acquisition function selects the configurations to evaluate in an iteration is a *mixture* of GAN-generated and randomly generated configurations. After a configuration is chosen, it is evaluated through real execution, and if its performance is better than the "current best", it is stored in the evaluated set (ES). The samples in ES will influence the selection in the future iterations. The details and rationales of the design will be explained in Section 3.2.

We comprehensively evaluate *Swift* using four Flink programs, twenty four Spark programs with three inputs each, and twenty three configuration-sensitive Spark SQL queries from TPC-DS [53]. We compare *Swift* against four state-of-the-art approaches: CherryPick [2], Selecta [27], DAC [64], and GML [18]. Since the last three achieve comparative speedups but require significantly more time to achieve the speedups compared to CherryPick, we take CherryPick as our baseline. It is worth mentioning that OPPerTune [42] and SelfTune [26] require over 10× as many samples as *Swift* and thus we did not compare them with *Swift* in the experimental section.

The experimental results show that *Swift* improves the throughput of Flink programs tuned by CherryPick by up to 1.59× (1.28× on average), and reduces their latency by up to 1.68× (1.31× on average). Most importantly, *Swift* completes the optimization in only 5.8 hours while CherryPick takes at least 12.5 hours. For Spark programs, *Swift* reduces their execution time tuned by CherryPick by up to 2.2× (1.2× on average). Similar to Flink, *Swift* takes up to 156% less time (61% on average) than CherryPick to optimize the performance of the experimented Spark applications.

We also evaluate *Swift* with a Flink program in a production cluster of an internet giant. The program was optimized by manual tuning and the optimization took a performance engineer *four days* to complete. The results show that *Swift* improves the throughput of this manually optimized program by 2.3× and reduces its latency by 2.8× in *only* 6.8 *hours*.

The rest of this paper is organized as follows. Section 2 discusses the background and motivation. Section 3 describes ON and *Swift* in detail. Section 4 discusses the experimental setup. Section 5 analyzes the results. Section 6 describes the related work and Section 7 concludes the paper.

2 Background and Motivation

2.1 Configurations of Big Data Systems

MapReduce is the first programming model designed for big data analytics applications [9] with various implementations such as Apache Hadoop [46] and Spark [65]. Later, many other frameworks have been deployed in industry, such as big table [7], GraphLab [34] for machine learning, Storm [52] for stream analytics, Pregel [36] for graph analytics, and Flink [45] for fused batch and stream analytics.

Although these frameworks serve for different domains, they all need to set a number of configuration parameters. Some parameters specify resource-related factors including CPU, memory, and network resources. For example, `spark.executor.memory` of Spark specifies the memory capacity allocated for an executor. This block of memory is further divided into execution memory, user memory, and reserved memory (e.g., 300 MB). Their sizes are controlled by `spark.memory.fraction` [43]. Other frameworks such as Apache Flink have similar parameters. For instance, parameter `taskmanager.memory.fraction` in Flink specifies the relative amount of memory that the task manager reserves for sorting, hash tables, and caching of intermediate results. The performance of big data systems is significantly influenced by such configuration parameters.

2.2 Limitations of Prior Work

Recent ML-based configuration tuning approaches [4, 15, 20, 21, 64] for Hadoop and Spark have demonstrated significant speedups. But as a general challenge, ML-based approaches need to evaluate a large number of samples in the real cluster, which takes at least several minutes per sample. As a result, it takes a long time (e.g., days or weeks) for the training/optimization to obtain the optimal configurations.

Figure 1 shows the time used by Bayesian Optimization (BO) to tune the configuration parameters of six typical Spark SQL queries ($q14a$,$q14b$,$q23a$, $q23b$, $q64$, and $q72$)

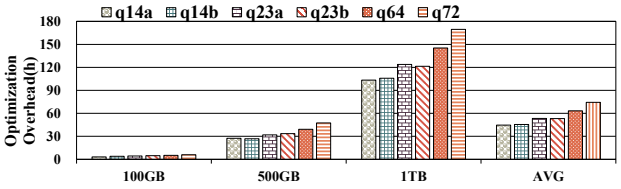


Figure 1: The time taken by using BO to tune the configurations of queries (e.g., $q72$) from TPC-DS [53] for optimal performance.

from TPC-DS [53] for optimal performance with three different input data sizes. It shows that, when the data size is 500 GB, it takes more than 30 hours to optimize each single query's performance. As the input data size increases, the time used to optimize its performance also increases rapidly. For example, it takes at least 100 hours when the input data size is 1 TB. In fact, the common size of data processed in industry ranges from several hundreds of GBs to TBs. As mentioned earlier, other ML algorithms require even longer time than BO to generate training samples to achieve same performance. The long training time significantly hinders the applicability of ML-based approaches in practice.

2.3 Bayesian Optimization

BO incorporates prior belief about objective function f and updates the prior with samples drawn from f to get a posterior that better approximates f. In this paper, f is program's performance ($perf$) which is a function of configuration parameters (c_i). Note that $perf$ can be execution time, latency, or other performance metrics. The goal is to find a vector $\{c_1, c_2, ...c_i, ...c_n\}$ named *optimal configuration* that leads to the highest performance for a program with the minimal number of evaluations[1]. To efficiently search the optimal configuration, surrogate model and acquisition function play the key roles [28]. Different models lead to different efficiency and complexity. We need a model which achieves a good trade-off between the two. Mockus et al. [37] show that the Gaussian Process (GP) is well-suited to serve as a surrogate model. We therefore choose GP in this study.

Acquisition function is used to select the next sample in each BO iteration. The selected sample is expected to achieve the largest improvement. An effective acquisition function is Expected Improvement (EI)—picking the point which can maximize the expected (performance) improvement over the current best [40]. The detailed calculation of EI can be found in [40]. Here, we introduce its typical implementation [12] which is adopted and substantially modified in our system. In the implementation, a function called *acq_max* is used to find the maximum of the acquisition function. It consists of two steps. First, it generates *n_warmup* (e.g., 10^5) random samples and finds the sample ($S1$) with the maximum potential improvement. Second, it runs the L-BFGS-B algorithm [66] from *n_iter* (e.g., 250) random starting points to find another sample ($S2$) with the maximum potential improvement. Finally, *acq_max* chooses the sample with higher improvement from $S1$ and $S2$ as the next sample for evaluation (iteration). Both steps generate samples randomly.

[1]We define sample evaluation as the execution of a program with a configuration.

2.4 Generative Adversarial Networks (GAN)

GANs belong to a new class of machine learning algorithms where generative (G) and discriminative (D) models are simultaneously generated in a competitive setting [17]. The objective of D models is to correctly label samples from either the G models or real data. In contrast, the objective of G models is to deceive the D models. In other words, the G models try to produce samples that are categorized as real data by the D models. As such, G models can be considered as a team of counterfeiters while D models are the law enforcement agents trying to detect the "counterfeit currency". Both teams try to improve their methods until the counterfeits are indistinguishable from the genuine ones [16].

Advantages: GANs can generate data that have the *distribution* similar to the input data [16,17]. As a result, GANs are widely used to generate similar versions of the image, text, video, and audio object. By using GANs and machine learning, one can easily recognize objects such as trees, streets, bicyclists, persons, and parked cars, and can also calculate the distance between different objects. It is worth mentioning that although diffusion models [41] have been widely used, they are notorious for their large data requirements and long-training process [24,39]. GAN, however, works significantly faster and it can even outperform diffusion models for various datasets [25].

Disadvantages: GANs may be hard to train. The reason is two-fold: first, there are too many factors (e.g., features to represent a speech) to learn by the standard GAN algorithm; second, the criterion of being highly similar is too strict. For example, the generated speech is required to be exactly the same as the input one. This may make the training process of a GAN very lengthy. To address this issue, more than 1,000 GAN variants are developed to deal with a special use case [23]. For example, WGAN (Wasserstein GAN) proposes a better way to measure the distance between the generated data distribution and that of the input one, which can generate more similar image to the input image than the standard GAN does [3]. Since we consider at least 27 parameters, it is also challenging to train a standard GAN model. We address this challenge in Section 3.2.2.

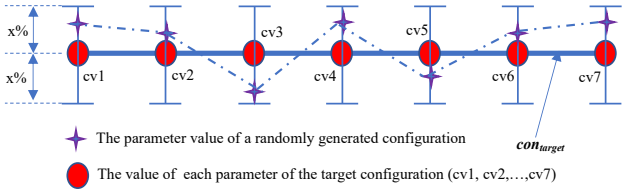


Figure 2: An illustration of how ON generates configurations.

3 Configuration Generation

We propose two approaches to generate configurations: One-Neighbor (ON) and the Swift. ON is a simple approach that generates configurations according to a target configuration based on small random perturbations, whereas Swift leverages GAN to generate configurations that share similar distribution with the target sample.

We define a configuration as a vector *conv* where its elements are the parameter values, as shown in Equation (1).

$$conv = \{c_0, c_1, ..., c_i, ..., c_{n-1}\} \quad (1)$$

where *conv* is a configuration, c_i is the i^{th} parameter value of *conv*. Example configuration parameters (c_i) for Flink and Spark programs are shown in Table 1. n is the number of configuration parameters in *conv*. In this study, n is 27 for Flink and 34 for Spark, as shown in [44].

A *sample* is defined as equation (2) shows.

$$spv = \{pf, conv\} \quad (2)$$

where *spv* is a sample, pf is the performance (e.g., execution time or latency) of a program produced by configuration *conv*.

3.1 One-Neighbor

One-Neighbor (ON) generates a configuration by producing its each parameter value randomly around the corresponding value of the target with a noise threshold. ON assumes that similar configurations produce similar performance for a given big data program. The similarity between two configurations $conv_1$ and $conv_2$ can be measured as their in-between Manhattan Distance, as shown below in equation (7).

$$MD(conv_1, conv_2) = |c_{11} - c_{21}| + ... + |c_{1n} - c_{2n}| \quad (3)$$

where c_{11} and c_{1n} are the first and n^{th} parameter value of $conv_1$, c_{21} and c_{2n} are the first and n^{th} parameter value of $conv_2$. To generate similar configurations according to a target configuration, we make the distance between two corresponding parameter values such as $|c_{11} - c_{21}|$ of two configurations be between 0 and $x\%$ (x is the threshold) randomly. The parameters are all normalized to $[0.0, 1.0]$.

Figure 2 visualizes how ON generates configurations according to a target configuration $conv_{target}$ which is represented by the bold line. The red circles (e.g., $cv1$ and $cv2$) on the bold line denote the parameters values of the $conv_{target}$. To generate a configuration *conv* similar to $conv_{target}$, we need to generate each parameter value around the corresponding value (e.g., $cv1$) of $conv_{target}$. To this end, as shown in Figure 2, we randomly generate a value between $(cvi - x\% \times cvi)$ and $(cvi + x\% \times cvi)$ $(i = 1, 2, ..., 7)$. The star marks show the generated parameter values, and the dashed line in Figure 2 shows a randomly generated *conv*.

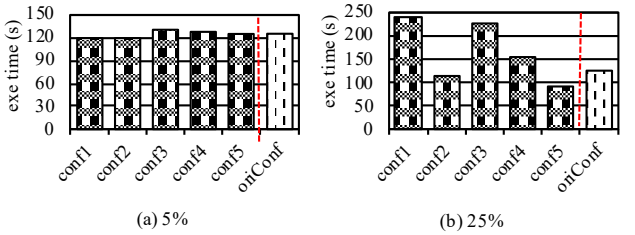


Figure 3: The performance made by ON generated configurations where the element values are within 5% (a) and 25% (a) of those of the original one.

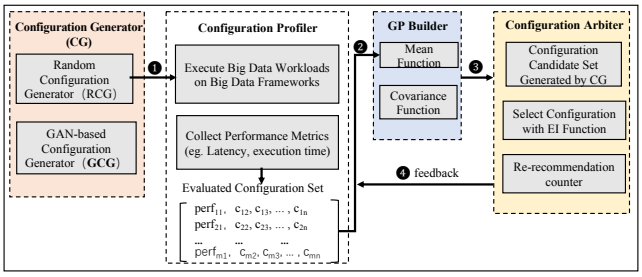


Figure 4: The block diagram of Swift Methodology.

The hope is that, based on ON, the performance of a program configured by *conv* is similar to that of the program configured by $conv_{target}$ when the threshold $x\%$ is small (e.g., 5% or 25%). However, Figure 3 shows that it is not the case. First, when $x = 5$, the performance by the ON generated configurations is almost the same. This does not add valuable samples. Second, when $x\%$ is around 25%, the performance corresponding to *conv* (e.g., $conf1$) can be as much as $2\times$ slower than that of $conv_target$ ($oriConf$ shown in Figure 3(b)).

3.2 Swift

3.2.1 Overview

Swift is a BO-based fast configuration tuning mechanism for optimizing the performance of big data systems. The key idea is to generate some configurations by GAN in each iteration for BO's acquisition function to choose from. These configurations highly likely produce high performance. *Swift* consists of four components shown in Figure 4.

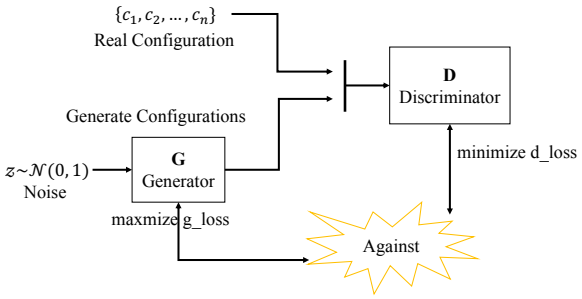


Figure 5: How GAN generates configurations.

- **Configuration generator** generates a number of configurations, either randomly or using GAN.
- **Configuration profiler** executes a program with a configuration and records its performance (e.g., execution time). If a configuration's performance is better than that of all samples in the evaluated set (ES), the configuration and its performance is inserted into ES as a new element.
- **GP (Gaussian process) builder** builds a model based on the samples in ES. The model is updated whenever a new sample is inserted in ES. It is used by BO.
- **Configuration arbiter** implements the new procedure in *Swift* that allow each BO iteration to select from a mixed pool of GAN-generated and random configurations.

3.2.2 Configuration Generator

Swift requires two configuration generators: a Random Configuration Generator (RCG) and a GAN-based Configuration Generator (GCG). The RCG sets each configuration parameter to a uniformly random value within the parameter's value range. In contrast, GCG leverages GAN's nature to generate the configuration that mimics the data distribution of the element values of target configuration.

Figure 5 shows how the GCG generates configurations. As explained before, a GAN consists of a discriminative model (*D*) and a generative model (*G*), which are both implemented by fully connected layers, as shown in `Listing 1`. GCG has two inputs: (1) the current target configuration that achieves high performance so far; and (2) a set of *noise* data which obey Gaussian distribution. The target sample is one of the samples in ES. A noise configuration is generated in two steps. First, we create a random value generator for each parameter. Second, we use a function (e.g., `nextGaussian()`) provided by a statistic software package (e.g., `java.util`) to generate values that obey Gaussian distribution for each parameter. In this way, we obtain a set of noise configurations where the values of each parameter obey Gaussian distribution.

The noise configuration is given to the Generator (G) and the target one (`real_conf0`) is given to the Discriminator (D) of our GAN model, as shown in `Listing 1`. G consists of three fully connected layers. The first layer performs a non-linear transformation by using the active function *ReLu* on the given noise configuration, generating `G_L1` as the output. The second layer performs another *ReLu* transformation on `G_L1`, producing `G_L2`. It is then given to the third layer to change its dimension to a specific number (e.g., the number of configuration parameters) and output `G_out`.

D first uses *ReLu* to perform a non-linear transformation on the target configuration, producing `D_L0`, as shown in lines 12 and 13. Subsequently, `D_L0` is given to the next layer, performing another non-linear transformation by using the active function *sigmoid*, as well as generating a probability `prob_conf0` ranging from 0.0% to 100.0%. Next, D performs a *ReLu* transformation on `G_out` and produces `D_L1` which is given to the next layer of D to perform a *sigmoid* transformation, generating another probability `prob_conf1`. It is used to calculate the *G_loss* to improve G to generate configurations more similar to the target. At the same time, the probabilities `prob_conf0` and `prob_conf1` are used to compute the *D_loss* to improve D to distinguish the target configuration from the generated ones more accurately.

```
1  Listing 1 An implementation of GAN.
2
3  dim=27 # the number of configuration parameters
4
5  with tf.variable_scope('Generator'):
6     G_in=tf.placeholder(tf.float32,[None,dim])
7     G_L1=tf.layers.dense(G_in,128,tf.nn.relu)
8     G_L2=tf.layers.dense(G_L1,64,tf.nn.relu)
9     G_out=tf.layers.dense(G_L2,dim)
10
11 with tf.variable_scope('Discriminator'):
12    D_L0=tf.layers.dense(real_conf0,128,
13                         tf.nn.relu)
14    prob_conf0=tf.layers.dense(D_L0,1,
15                         tf.nn.sigmoid)
16    D_L1=tf.layers.dense(G_out,128,
17                         tf.nn.relu, reuse=True)
18    prob_conf1=tf.layers.dense(D_L1,1,
19                         tf.nn.sigmoid, reuse=True)
20
21 D_loss= -tf.reduce_mean(tf.log(prob_conf0)
22                         +tf.log(1-prob_conf1))
23 G_loss=tf.reduce_mean(tf.log(1-prob_conf1))
```

Note that the goal of the non-linear transformations performed by the active functions *ReLu* and *sigmoid* is to capture the complex features such as data distribution hidden in the values of the target configuration. This makes it possible to use GAN to generate configurations according to only one target configuration, i.e., the one with top-1 performance. A similar approach has been used in computer vision where the authors use GAN to repair a tiger face paint [58]. The inputs of their GAN are one damaged paint and some noise data.

Solution for the hard-training issue. The ultimate goal of GAN in general applications such as image generation is to make `prob_conf0 = prob_conf1`. To achieve this goal, the training of GAN may suffer from the "hard-training" issue—the training process may lead to an infinite loop to satisfy the above condition. However, usage of GAN in *Swift* does not suffer from this problem for two reasons. First, the data type

in *Swift* is one dimension (vector), instead of two (image) or higher dimension in other applications. As a result, the training process is easier.

Second, *Swift* does not and *should not* need the above condition, because the generated configuration is *not* expected to have the same performance. It is not only unnecessary but also *harmful* to the search, i.e., if we always achieve the same performance, BO cannot find the better configuration. Thus, it is in fact *required* to let the performance be within a range (e.g., $\pm 25\%$) of the performance of target sample, which provides the opportunity to generate better configurations.

Guided by these rationales, we train the GAN for a *fixed* number of iterations(i.e., 20 in our experiments). We observe that it typically ensures the performance of generated configurations remains within ±25% of the target sample's performance. Notably, even if 20 iterations fail to bring the performance within ±25%, this does not hinder the optimization process—we can still utilize the generated configurations to complete the search for optimal configurations and achieve the best possible result.

3.2.3 Configuration Profiler

The configuration profiler executes a program with a configuration and records its performance, generating a sample $spv = \{pf, conv\}$ defined in equation (2). After profiling a program a number of times, all the vectors form a matrix that represents ES as shown in Figure 4. A configuration is considered to be evaluated if we know its performance, but it is only inserted into ES if its performance is better than the performance of all rows in ES.

3.2.4 Gaussian Process Builder

To build a GP that outputs the performance pf of a program with a given configuration, we first leverage the *configuration profiler* to generate kk vectors (spv). Then, we use these samples to calculate the initial performance mean and covariance cov between the performance and a configuration as follows.

$$\widetilde{pf} = (\sum_{i=1}^{kk} pf_i)/kk; \widetilde{cf_j} = (\sum_{i=1}^{kk} cf_{ji})/kk \tag{4}$$

$$cov(pf, cf_j) = (\sum_{i}^{kk} (pf_i - \widetilde{pf})(cf_{ji} - \widetilde{cf_j}))/kk \tag{5}$$

with $\widetilde{pf}$ the mean performance of the kk executions, pf_i the performance in the i^{th} sample, cf_j the j^{th} value of a sample, and cf_{ji} the value of the j^{th} parameter of the i^{th} sample.

Thus far, we already have the two essential elements, mean and covariance functions, to define a GP. The last element we need to build a GP is the smooth kernel function which represents how similar two configuration parameters are in terms of performance. If two configuration parameters perform similarly, they have high covariance, and vice versa. There are numerous smooth kernel functions that satisfy this need. We choose $Matern5/2$ [59] as it does not require strong smoothness and is preferred to model practical functions [40]. During BO, the GP model is updated when ES is updated.

3.2.5 Configuration Arbiter

This component is responsible for guiding the steps for each BO iteration. To clarify the mechanisms of *Swift*, let us first explain the iterative process in the conventional BO as shown in Figure 6 (a). All the components in *Swift* can be repurposed for conventional BO as the needed functionality is the same or simpler. In each iteration, a number of configurations (e.g., 100,000 in the example) are randomly generated by the configuration generator. Then, the internal BO algorithm selects a configuration using the acquisition function (EI) for this iteration. If the configuration is not already contained in ES, the configuration profiler executes the program on the real cluster. The execution takes the majority of time of an iteration and the BO optimization. Ideally, the BO algorithm will select the configuration with less and less execution time, so the time spent in each iteration will decrease. A sample is inserted into ES if its performance is higher than the one with the highest performance in the current ES, i.e., "current best"; otherwise (or the selected configuration is already in ES), it is discarded and the BO moves to the next iteration. It means that the effort in this iteration is wasted.

The number of iterations (and the total BO optimization time) is determined by the quality of configurations selected in each iteration: good selections lead to faster execution time decrease and less number of iterations. However, with random configuration generation, the BO algorithm, which needs to consider both exploitation and exploration, may occasionally select a configuration that leads to longer execution time. It will directly increase the BO optimization time.

In our experiments, we observe that the execution time based on a configuration can be $2\times$ slower than the execution in the previous iteration. The iterative BO optimization terminates when the performance does not improve for a number (e.g., five in the example) of consecutive iterations. At this point, the configuration of the sample with the highest performance is given as the outcome. With random configuration generation, the execution time decrease can be slow with fluctuation. We conceptually provide how the execution time is changed over iterations on the left of Figure 6 (c).

Swift reduces the BO optimization time by mixing the randomly generated configurations for each iteration with GAN-generated configurations that have similar *distribution* with the "current best". This design emphasizes both *exploitation*, which takes advantage of the previously obtained information—the target sample—to acquire rewards, and *exploration*, which tries to reach the global optimal with random samples. The workflow of configuration arbiter is shown in Figure 7, and Figure 6 (b) shows a concrete example.

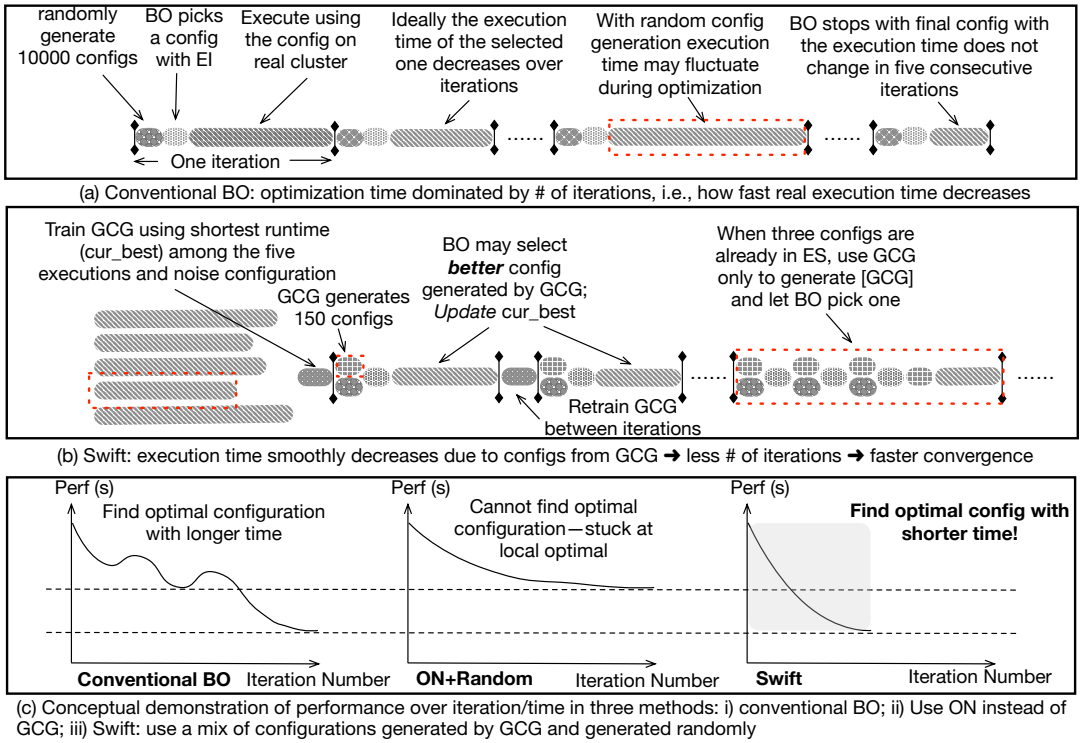


Figure 6: Comparing Conventional BO and *Swift*

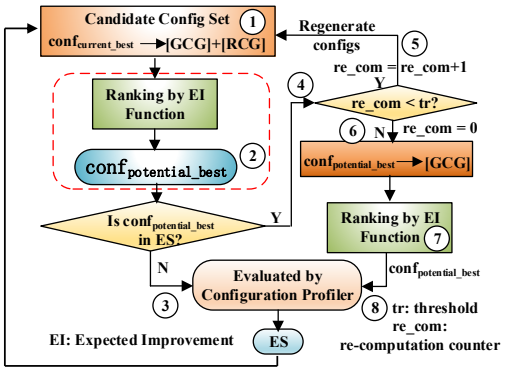


Figure 7: Workflow of the configuration arbiter.

For GAN to generate configurations, it needs to be trained first. To do that, *Swift* first randomly generate five configurations and evaluate them on the real cluster; the configuration with highest performance among the five is used as the target sample to train the GAN with a noise configuration that obey Gaussian distribution for each parameter. The training procedure is described in `Listing 1` and can be done very fast, e.g., several seconds. The training of GAN is triggered whenever the "current best" is updated.

After the first training, BO iterations can start. Different from the conventional BO, the candidate configuration pool is composed of [RCG], a large set (e.g., 100,000) of configurations generated, and [GCG], a small set (e.g., 150) of configurations generated with GAN (step ①). Then, the same to the conventional BO, the algorithm selects the configuration with the highest potential performance (step ②). The high quality configurations in [GCG] can help the BO algorithm to overcome the "bad luck" and make it more likely to select good configurations. As a result, compared to BO, the selected configuration in each iteration in *Swift* tends to be better with *faster and more consistent* execution time decrease, which is shown conceptually on the right of Figure 6 (c). This benefit is clearly observed in our experiments, see Figure 19.

If the selected configuration is not already in ES, it is evaluated by the configuration profiler (step ③). Otherwise, *Swift* gives a small number of chances to re-generate [GCG]+[RCG] and repeat the selection (step ④ and ⑤). The threshold is controlled by *tr* and is set to three in our experiments. The counter *re_com* keeps the number of re-generation attempted. After *re_com* reaches *tr*, if the selection is still contained in ES, *Swift* will generate a new pool of candidates just by [GCG] without the random configurations (step ⑥). BO selects a configuration from the set as the last attempt (step ⑦) before moving to the next iteration. If the selected configuration is not in ES, it is evaluated by the configuration profiler (step ⑧).

For simplicity, we do not show the case when the selected configuration is already in ES in Figure 7. In this case, *Swift* moves to the next iteration. The rationale for this design is to capture the following scenario. BO may try to select a con-

figuration in [RCG] with the highest performance potential, but it is already in ES. However, the second or third best is a configuration in [GCG]. Instead of wasting the iteration, after *tr* attempts to select from [GCG]+[RCG], BO can just focus on [GCG] and still select a high quality configuration.

In principle, we can replace the [GCG] generated by GAN with configurations generated by ON with small perturbation so that the performance is within ±25%. However, as discussed before, such configurations are too similar to the target one and will negatively affect the search. The issue is that, such very similar configurations with good performance tend to make the BO algorithm more likely to stuck at the local optima when terminated. It is conceptually illustrated in the middle of Figure 6 (c). Although the performance increases consistently, but the mix of ON generated and random configurations make search stop with a sub-optimal performance.

3.2.6 Termination Condition

To design the termination condition for Swift, we first define the performance improvement as follows.

$$imp = \frac{|pf_{i+1} - pf_i|}{pf_i} \times 100\% \tag{6}$$

with imp the performance improvement in percentage, and pf_i and pf_{i+1} the performance of the i^{th} and $(i+1)^{th}$ iterations, respectively. During the iterations of Swift, if we observe imp is less than a threshold (e.g., 5%) and N (e.g., 6) iterations have been performed, we terminate Swift searching and take the configuration found in the last iteration as the optimal configuration. This ensures that *Swift* does not terminate the search too soon and only finds the sub-optimal configurations. Moreover, this also prevents *Swift* from struggling to make marginal improvements.

4 Experimental Setup

4.1 Apache Flink Experimental Setup

Our experimental cluster consists of four Linux servers. One server serves as the master node which runs the Job Manager and the other three serve as slave nodes which execute the Task Managers. Each server is equipped with an Intel(R) Xeon(R) CPU E5-2407 2.20GHz 4-core processor and 32GB DRAM. The OS of each node is SUSE Linux Enterprise Server 11. We use Flink 1.4.2 in this study.

We evaluate four programs from the Flink version of HiBench: *FixWindow* (FW) is a window-based aggregation which evaluates window operations in the stream frameworks. *WordCount* (WC) tests the performance of the stateful operators and the cost of checkpoints/Ackers. *Repartition* (RP) evaluates the shuffle efficiency. *Identity* (ID) tests the read-/write efficiency of an external input source represented by a Kafka cluster. Data streams for all the benchmarks are from a Kafka cluster [47], and the input data speed is 1000MB/s.

We also evaluate Swift on a real production cluster of an internet giant. In this experiment, we use two docker containers as Job Managers and ten docker containers as Task Managers. The docker containers are managed by a customized version of Kubernetes [50]. Each container is equipped with 4 CPU cores (Intel(R) Xeon(R) CPU E5-2698 v3 @ 2.30GHz), 8GB memory, and 10GB/s Ethernet network. All the containers share a 100GB network disk. The OS is CentOS 7 with Linux kernel 3.10.0-327.28.3.el7.x86_64 and the Apache Flink version is 1.4. We use a real-time log analysis program written in Flink, which has been manually optimized by the engineers of the company. This optimization took the engineers four days and we use the optimized performance as our baseline.

We choose a wide range of Flink configuration parameters that significantly influence performance, including *memory management*, *execution behavior*, *networking*, *parallelism*, etc. Table 1 shows 5 example parameters and all the 27 parameters experimented in this study are shown in [44]. The last column of Table 1 provides the default values of the parameters and can be found at Apache's website [11]. The third column shows the value range of each parameter.

4.2 Spark Experimental Setup

Our Spark cluster consists of eight servers. One server serves as the master and the others serve as slaves. Each server is equipped with Intel (R) Xeon (R) CPU E5-2630 v3 @2.40GHz and 64GB of memory. The OS of each server is Ubuntu 16.04 LTS and Spark version is 2.2. The resource manager is Yarn v1.22.19.

We employ 24 Spark benchmarks from HiBench [51], each with 3 input data sets. These benchmarks cover a wide range of Spark applications including machine learning, web searching, graph processing, SQL, and micro-benchmarks.

As discussed earlier, we choose a wide range of Spark configuration parameters that significantly influence performance, including *shuffle behavior*, *data serialization*, *memory management*, *execution behavior*, *networking*, etc. Table 1 shows five example configuration parameters and [44] lists all the 34 experimented parameters in detail. The last column of Table 1 provides the default values of parameters which are recommended by the Spark team [43]. The third column shows the value range of each parameter.

It is worth mentioning that *Swift* does not distinguish hardware specific configurations, i.e., configurations generated by *Swift* for a Spark or Flink program on one hardware platform are optimal on this specific platform and are essentially not portable to a different platform.

Table 1: Description of five example Flink and five example Spark configuration parameters.

Framework	Configuration Parameters-Description	Range	Default
Flink	**parallelism.default** - The default parallelism for programs that have no parallelism specified.	1-24	1
	taskmanager.numberOfTaskSlots - # of parallel operator or function instances a TaskManager runs.	1-8	1
	jobmanager.heap.mb - JVM heap size (in megabytes) for the JobManager.	1024-4096	1024
	taskmanager.heap.mb - JVM heap size (in megabytes) for the TaskManager.	1024-10240	1024
	taskmanager.memory.fraction - The relative amount of memory that the task manager reserves.	0.5-0.8	0.7
Spark	**spark.shuffle.file.buffer** - Size of the in-memory buffer for each shuffle file output stream, in KB.	2-128	32
	spark.executor.cores-The number of cores to use on each executor.	4-30	core #
	spark.driver.memory-Amount of memory to use for the driver process, in MB.	1024-36864	1024
	spark.executor.memory-Amount of memory to use per executor process, in MB.	7168-36864	1024
	spark.memory.fraction-Fraction of (heap space - 300MB) used for execution and storage.	0.5-1	0.75

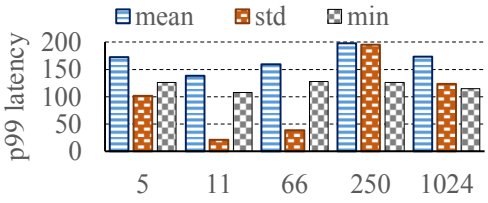


Figure 8: The variation of 99^{th} percentile latency of *wordcount*. The X axis represents the 5 random seeds.

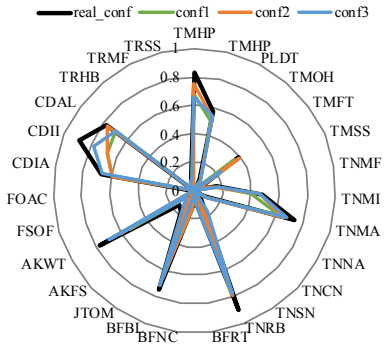


Figure 9: The quality of GCG generated configurations.

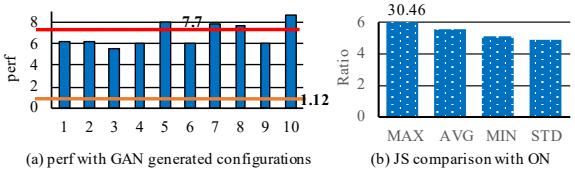


Figure 10: The high quality of GAN-generated configurations.

5 Results and Analysis

5.1 Study of GAN Related Issues

Impact of Random Seed In *Swift*, the randomness in the 5 initial executions may lead to great uncertainty. To study this, we employ 5 random seeds to generate 5 configurations for the Flink version of *wordcount*. We run it with the generated configurations and observe the mean, standard deviation and minimum of the 99^{th} percentile latency. Figure 8 shows the results with random seeds (5, 11, 66, 250, and 1024). We see that the means and standard deviations of the 99^{th} percentile tail latency of *wordcount* are significantly different among each other. However, the standard deviation of the minimum 99^{th} percentile tail latency with different random seeds is 8.7 and the corresponding coefficient of variation (CoV) is only 0.06, which indicates that the latency is highly similar. Thus, the different random seeds do not affect *Swift* very much.

Thus we run one experiment using the default seed and do not use different seeds with multiple runs. We argue that on the one hand, if different seeds are utilized, the optimal performance and configuration may vary, the search processes exhibit similar experimental results, as proved by small CoV values. On the other hand, many benchmarks are evaluated (4 Flink and 24 Spark programs), which eliminate the randomness of one program with a particular seed.

Quality of GAN Generated Configurations In this section, we experimentally show the similarity and high quality of the GAN-generated configurations. For similarity, we first randomly generate a real configuration for Flink programs with 27 parameters. Then we use GCG to generate three artificial configurations using the real one as the target. Figure 9 (left) compares the four configurations in a Kiviat plot. We can clearly see the similarity among the four. Quantitatively, the Manhattan distances between the generated configurations and the real one are 1.56, 0.7, and 1.47, which are small.

For configuration quality, we run the Flink program *Fixwindow* with a real configuration with high performance, ten GAN-generated configurations, and the default configuration. Figure 10 (a) shows the performance comparison. The red and orange dashed lines represent performance with the real and default configurations. All GAN-generated configurations have performance close to that of the real configuration with maximum relative variation 25.7%.

Comparing to One-Neighbor This section quantitatively compare the configurations generated with ON and GAN. We generate 110 configurations by each of the two methods ($\pm$25% noise for ON) based on a real configuration for the Spark version of *PageRank*. We calculate the Manhattan

distance (MD) and Jensen-Shannon (JS) divergence [13] of configuration in the two sets to the real configuration. That is, we calculate 110 Manhattan distances (MD) and 110 JS divergences for the ON generated configurations; we also compute 110 MDs and 110 JS divergences for the GAN generated configurations. Then we compute $ratio_{md}$ and $ratio_{js}$ as equations (7) and (8) show, respectively.

$$ratio_{md} = MD_{on}/MD_{gan} \tag{7}$$

with MD_{on} the MD of an ON generated configuration to the real configuration, and MD_{gan} the MD of a GAN generated configuration to the real one.

$$ratio_{js} = JS_{on}/JS_{gan} \tag{8}$$

with JS_{on} the JS divergence of an ON generated configuration to the real configuration, and JS_{gan} the JS divergence of a GAN generated configuration to the real one.

If $ratio_{md}$ is close to 1.0, it indicates that the ON and GAN generated configurations have similar MDs to the real configuration. If $ratio_{js}$ is significantly larger than 1.0, it implies that the GAN generated configurations' value distributions are much more similar to the real configuration's value distribution than the those of the ON generated configurations.

We then compute (min,max,avg) of $ratio_{md}$ and $ratio_{js}$. For Manhattan distance, the (min,max,avg) of $ratio_{md}$ is (0.46,2.81,1.05). For $ratio_{js}$, the (min,max,avg) is (5.15,30.46,5.58) as shown in Figure 10 (b). Clearly, we see that while the configurations in both sets can be considered "similar" to the real configuration, but the distribution (captured by JS) of GAN-generated configurations is much closer to the real configuration than ON-generated ones. As such, the GAN generated "similar but not too similar" configurations to a real one also guarantee the "the similar but not too similar" performance, whereas ON generated ones do not.

We plot the performance distribution of 110 GAN- and ON-generated configurations as blue solid and red dashed lines in Figure 11. We see that the GAN-generated configurations follow a Gaussian distribution with mean execution time (86.8s) very close to the real configuration (87.4s); while the ON-generated configurations follows an irregular long-tail distribution. Moreover, ON can generate configurations of which performance spreads in a large range. This means that it may generate low quality configurations despite small Manhattan distance. Overall, the performance of GAN-generated configurations is *"similar but not too similar"* while the performance of ON-generated configurations varies significantly.

GAN with Multiple Target Configurations It is interesting to understand whether it is beneficial to use multiple target configurations to train our GAN model. To answer the questions, we choose three configurations for *PageRank* with execution times 87.4s, 87.6s, and 88.5s, respectively. We use the three configurations to train the GAN model and generate 110 configurations. The green solid line with rectangles in

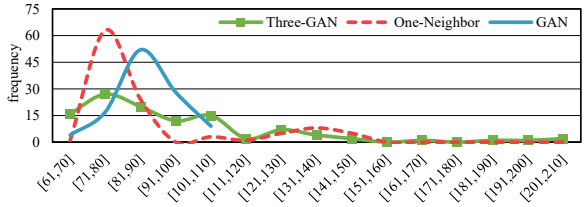


Figure 11: The performance distribution of *PageRank* with GAN, ON, or 3-input-GAN generated configurations.

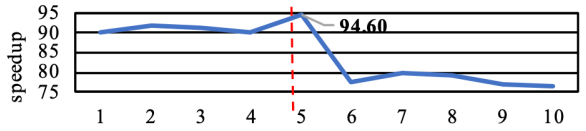


Figure 12: The speedup with different number of initial configurations for *repartition* over default configurations.

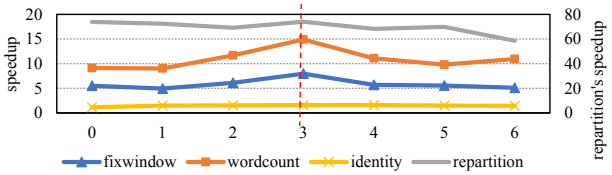


Figure 13: The performance variation with different values of tr. The X axis represents the tr values.

Figure 11 shows the performance distribution with a long tail, which can make the BO optimization even longer. The reason is that the GAN model tries to learn the data distribution of parameter values of all the three input configurations. The results suggest that the GAN model should be only trained with the configuration with the best performance so far.

5.2 Flink Programs on Lab Cluster

Number of initially evaluated configurations. Figure 12 shows the speedups of program *repartition* when using different number of initial evaluated configurations to provide the first target sample to train the GAN model. The performance reaches the highest when five configurations are evaluated and then drops. It is because more random configurations increase the chance of getting the "very bad" configurations. Based on this result, *Swift* initially evaluates five configurations

Tolerance threshold The tolerance threshold controls the times that we allow the acquisition function of BO to select the same sample to evaluate on the real cluster in the next iteration. A large threshold may waste time without much improvement, while a small threshold may make the optimization stuck at a local optima. Figure 13 shows how the threshold (x-axis) affects the performance for four Flink programs. For all of them, the best speedups are obtained when the threshold is three. Thus we use this number in *Swift*.

Figure 14: Latency improvements over CherryPick. FXW-Fixwindow, RPT-Repartition, WCT-Wordcount, IDY-Identity.

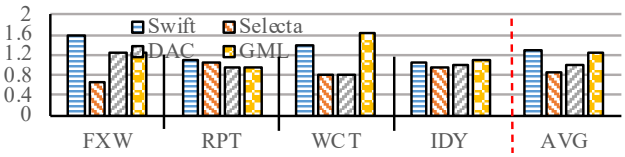


Figure 15: Throughput improvements over CherryPick.

Table 2: The number of evaluated configurations (trial executions) used to find the optimal performance.

Bench	Swift	CherryPick	Selecta	DAC	GML
Fixwindow	25	50+	100	750	500
Repartition	17	50+	100	550	440
Wordcount	23	50+	100	600	460
Identity	24	50+	100	600	420
In Total	89	200+	400	2,500	1820

Optimization time. We compare *Swift* with four parameter tuning approaches, including CherryPick [2], Selecta [27], DAC [64], GML [18]. For a fair comparison, we re-implement CherryPick, Selecta, DAC, and GML and carefully tune their hyper-parameters to achieve the highest performance.

Table 2 shows the number of trial executions used to find the optimal performance for the Flink programs with different approaches. It clearly indicate *Swift* requires much less number of trial executions. Note that the number of trial executions may not accurately represent the final time cost because a Flink program may take different time to execute with different approaches. We therefore also report the time used to find the optimal performance in Table 3. It shows that *Swift* takes only 6.3 hours at most and 5.8 hours on average to find the optimal performance for the 4 Flink programs. In contrast, the average times taken by CherryPick, Selecta, DAC, and GML are 2.2×, 4.3×, 9×, and 7.2× of that of *Swift*.

Speedup. Figure 15 shows the throughput improvements of the four Flink programs tuned by Swift, Selecta, DAC, and GML over by CherryPick. Swift improves the throughput of the four programs by 1.28× on average and up to 1.59× over CherryPick. Figure 14 and Figure 15 show the throughput improvement and latency reduction comparison. In both aspects, *Swift* clearly outperforms other approaches. It is worth noting that the main goal of *Swift* is making the optimization faster. The results are particular encouraging because *Swift* actually *achieves better performance with shorter time*.

Production Cluster. We evaluate a Flink program used to analyze the logs of servers in a real time. The results show that Swift improves the throughput by 2.3× and reduces the latency by 2.8× in only 6.8 hours compared to the four-day manual optimization. Table 4 compares the values of the six important parameters from the manual tuning and *Swift*. We see that *Swift* indeed significantly adjusted the values to achieve better performance.

Table 3: Time (in hours) used to achieve the improvements.

Bench	Swift	CherryPick	Selecta	DAC	GML
Fixwindow	6.3	12.5+	25	62.5	40.1
Repartition	4.3	12.5+	25	45.8	38.9
Wordcount	5.7	12.5+	25	50	38.5
Identity	6	12.5+	25	50	42.3

Table 4: Para values. TM-Taskmanager, JM-Jobmanager.

Parameters	Tuned Values	Opt Values
TM.memory.fraction	0.7	0.3
TM.network.memory.max	1024MB	3945MB
TM.net.client.numThreads	4	2
JM.tdd.offload.minsize	1024B	1434B
Blob.fetch.num.concurrent	50	141
akka.framesize	10MB	16MB

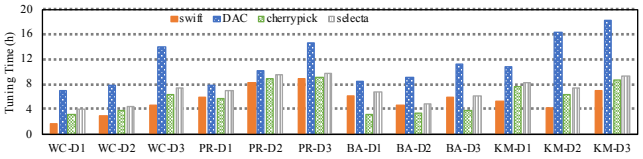


Figure 16: Optimization time comparison.

5.3 Spark Programs Evaluation

Optimization time. Figure 16 compares the optimization time of different approaches for Spark programs. The average time overheads of Swift, DAC, CherryPick, and Selecta are 5.1, 11.3, 5.9, and 7 hours, respectively. We see that *Swift* reduces optimization time more for Flink programs than for Spark programs. It is because of the different program characteristics: Spark programs are batch processing while Flink programs are streaming processing. We also evaluate *Swift* on a heterogeneous cluster consisting of eight x86 servers and four ARM servers with 24 Spark programs from HiBench, each with three input datasets. Figure 17 shows the average speedup and optimization time reduction for each program. *Swift* still significantly outperforms CherryPick. This indicates that *Swift* is robust in significantly different clusters and workloads, which is helpful for practice use.

Speedups. Figure 18 shows the speedups of the four approaches compared to the default configurations. We can see that *Swift* achieves slightly higher speedups but still with much shorter optimization time.

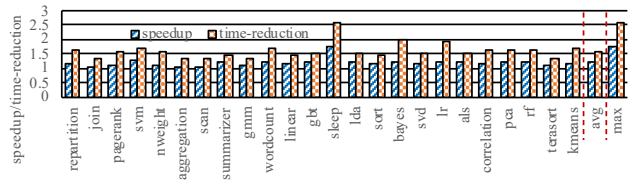


Figure 17: The optimization time reduction comparison with CherryPick for 24 Spark benchmarks.

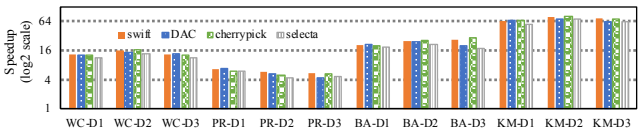


Figure 18: Speedup comparison. The baseline is the execution time of each program with default configuration.

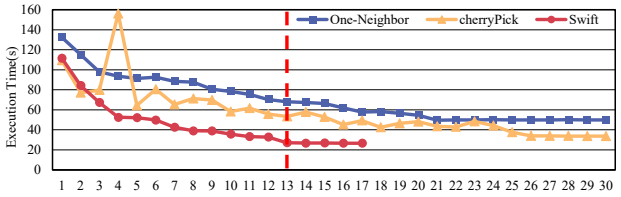


Figure 19: The performance of Spark WordCount tuned by the configurations selected by ON, CherreyPick (BO), and *Swift* during iterations. The X axis denotes the iteration number.

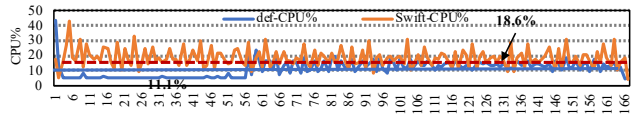


Figure 20: CPU utilization of *Fixwindow* tuned by Swift and with default configuration. The X axis represents samples.

5.4 Insight and Discussion

***Swift* skews the search space**. In *Swift*, mixing randomly and GAN-generated configurations has the effects of skewing the search space toward the optimal configuration, leading to faster convergence. In contrast, with uniformly random configurations, CherryPick searches in a uniform space. Figure 19 shows the performance of configurations selected in each iteration in *Swift* and CherryPick. We can see that, *Swift* converges faster and more smoothly without fluctuation.

Case study. Finally, we deeply analyze on *Fixwindow*. *Swift* accelerates it by 10.7× compared to the default configuration. Figure 20 and Figure 21 show the comparison of CPU and memory utilization comparison. We see that CPU utilization is increased by around 7.5% but the memory utilization is drastically reduced from 46.8% to 20.1%. From such results, we understand that the performance improvement is due to the busier CPU with reduced memory accesses.

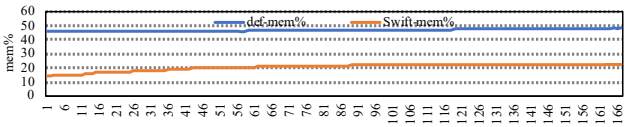


Figure 21: Memory utilization of *Fixwindow* tuned by Swift and with default configuration. The X axis represents samples.

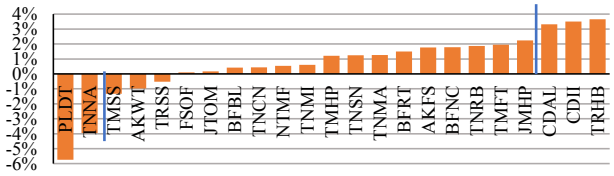


Figure 22: The importance quantification of configuration parameters for *Fixwindow* with respect to performance. The descriptions of the abbreviations along X axis are in [44].

We further investigate how the parameters affect performance, taking *Fixwindow* as an example. Figure 22 shows the 5 parameters affecting the performance most significantly. The increases of *TRHB*, *CDII*, and *CDAL* improve *Fixwindow*'s performance while those of *PLDT* and *TNNA* decrease the performance. The three parameters that positively correlate with performance are all related to memory. *TRHB* specifies whether we should enable bloom filter for the hybrid hash join operations in *Fixwindow*. Bloom filter is a bit vector which is a probabilistic data structure telling us that the element is either definitely not in the set or may be in the set. The default configuration sets it to *false* while Swift set it to *true* because *Swift* is able to detect if a lot of operations' values are in a set. The usage of bloom filter therefore reduces the memory consumption, which increases the CPU utilization.

Limitations. Although *Swift* demonstrates excellent performance, it still has several shortcomings. First, *Swift* was tested exclusively in single workload scenarios and may not fully generalize to complex data processing environments with concurrent workloads (i.e., workloads co-running to maximize hardware resource utilization). Future work is needed to address this limitation. Second, *Swift* requires the setting of several thresholds that are inherently tied to specific cluster environments. For instance, if there is a change in cluster hardware—such as replacing x86-based server CPUs with ARM-based ones—these thresholds may need to be reconfigured accordingly, posing challenges for system adaptability. Third, when dealing with extremely large input data sizes, such as 10 PB, the process of generating even a single trial configuration becomes prohibitively expensive. In such cases, *Swift* may struggle to operate efficiently.

6 Related Work

CherryPick [2] employs BO to optimize cloud configurations. It suffers the drawback of the polarized and unstable quality of configurations. Selecta [27] takes latent factor collaborative filtering and singular value decomposition to recommend near-optimal storage configurations with few samples. Ernest [54] builds a performance model via the job behavior and then predicts its performance on larger dataset sizes.

Another line of efforts use machine learning to build performance models, including DAC [64], RFHOC [4], OtterTune [1],QTune [29], Rafiki [35], Startfish [22], and gray-box based approaches [5], etc. However, the cost of collecting samples is expensive. In contrast, *Swift* reduces the cost significantly. Other search-based approaches take parameter tuning as a black-box optimization issue and uses variant search algorithms to find the optimal configurations [6], including BestConfig [67], Mronline [30], SelfTune [26], OPPerTune [42] and Gunther [32]. They can be applied to general scenarios. However, the key limitation is that they take a long time to find the optimal configuration since they evaluate each randomly generated configuration on a real cluster, e.g., OPPerTune requires 252 samples to tune parameters [42], which is time-consuming. In contrast, *Swift* reduces this time significantly.

Simulation-Based Approaches. Simulation models can capture both the internal metrics of systems and the externally observed workload relationships including [33, 55, 56] and [38]. These approaches need to probe system internals and is hard to cover all factors that can affect performance. In contrast, *Swift* works well on real big data systems.

Rule- and Analytical Model-based Approaches. Rule-[57, 60, 61] and analytical model-based [19, 20] approaches leverage the experiences of human experts (e.g., tuning instructions) to tune parameters for computing systems. They can quickly improve performance. However, deep understanding of the target system is required, which is not easy for emerging systems. In contrast, *Swift* does not need that.

7 Conclusion

This paper proposes *Swift*, a novel Bayesian Optimization (BO) based parameter configuration tuning approach for big data systems. The key idea is to leverage a generative adversarial network (GAN) to generate high quality configurations based on the evaluated configuration with the highest performance. Mixing these configurations with the randomly generated configurations has the effects of skewing the search space toward the optimal configurations, leading to faster convergence and less optimization time. In particular, our results show that *Swift* is able to achieve *better* performance than all existing approaches.

Acknowledgments

We are deeply grateful to our shepherd, Sriram Subramanian, and the anonymous reviewers for their valuable insights and dedicated efforts, which have greatly enhanced our work. This work was supported by Shenzhen Science and Technology Program (No.JCYJ20220818101607015), NSFC (62372442) and State Key Lab of Processors, Institute of Computing Technology, CAS (No.CLQ202308). Zhibin Yu is the corresponding author of this work.

References

[1] Dana Van Aken, Andrew Pavlo, Geoffrey J. Gordon, and Bohan Zhang. Automatic database management system tuning through large-scale machine learning. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD)*, 2017.

[2] Omid Alipourfard, Hongqiang Harry Liu, Jianshu Chen, Shivaram Venkataraman, Minlan Yu, and Ming Zhang. Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics. In *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2017.

[3] Martin Arjovsky, Soumith Chintala, and L'eon Bottou. Wasserstein gan. In *arXiv:1701.07875v3*, 2017.

[4] Zhendong Bei, Zhibin Yu, Huiling Zhang, Wen Xiong, Chengzhong Xu, Lieven Eeckhout, and Shengzhong Feng. Rfhoc: A random-forest approach to auto-tuning hadoop's configuration. *IEEE Transactions on Parallel and Distributed Systems*, 27(5):1470–1483, 2016.

[5] Muhammad Bilal and Marco Canini. Towards automatic parameter tuning of stream processing systems. In *Proceedings of the 2017 Symposium on Cloud Computing (SoCC)*, 2017.

[6] Zhen Cao, Vasily Tarasov, Sachin Tiwari, and Erez Zadok. Towards better understanding of black-box auto-tuning: A comparative analysis for storage systems. In *Proceedings of the 2018 USENIX Annual Technical Conference (ATC)*, 2018.

[7] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. In *Proceedings of the eighth USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[8] Dazhao Cheng, Jia Rao, Yanfei Guo, and Xiaobo Zhou. Improving mapreduce performance in heterogeneous environments with adaptive task tuning. In *Proceedings*

of the 15th International Middleware Conference, pages 97–108. ACM, 2014.

[9] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proceedings of the sixth USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2004.

[10] Christina Delimitrou and Christos Kozyrakis. Quasar: resource-efficient and qos-aware cluster management. In *Proceedings of the 19th International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.

[11] Apache Flink developers. Apache flink documents. https://ci.apache.org/projects/flink/flink-docs-release-1.5/. Accessed Sept 4, 2018.

[12] Fernando Fmfn and Ben Arnao. Bayesian optimization. https://github.com/fmfn/BayesianOptimization/blob/master/bayes_opt/util.py.

[13] Bent Fuglede and Flemming Topsoe. Jensen-shannon divergence and hilbert space embedding. In *Proceedings of the International Symposium on Information Theory (ISIT)*, 2004.

[14] Adem Efe Gencer, David Bindel, Emin Gün Sirer, and Robbert van Renesse. Configuring distributed computations using response surfaces. In *Proceedings of the 16th Annual Middleware Conference*, pages 235–246. ACM, 2015.

[15] Adem Efe Gencer, David Bindel, Emin Gun Sirer, and Robbert van Renesse. Configuring distributed computations using response surfaces. In *Proceedings of the Annual ACM/IFIP/USENIX Middleware Conference*, pages 235–246, December 2015.

[16] Ian J Goodfellow, Jean Pougetabadie, Mehdi Mirza, Bing Xu, David Wardefarley, Sherjil Ozair, Aaron C Courville, and Yoshua Bengio. Generative adversarial nets. pages 2672–2680, 2014.

[17] Roderich Grob, Yue Gu, Wei Li, and Melvin Gauci. Generalizing gans: A turing perspective. In *The 31st Conference on Neural Information Processing Systems (NIPS 2017)*, pages 1–11, 2017.

[18] Yijin Guo, Huasong Shan, Shixin Huang, Kai Hwang, Jianping Fan, and Zhibin Yu. Gml: Efficiently autotuning flink's configurations via guided machine learning. *IEEE Transactions on Parallel and Distributed Systems*, 32(12):2921–2935, 2021.

[19] Herodotos Herodotou. Hadoop performance models. Technical report, Duke University, 2011.

[20] Herodotos Herodotou and Shivnath Babu. Profiling, what-if analysis, and cost-based optimization of MapReduce programs. *Proceedings of the VLDB Endowment*, 4(11):1111–1122, 2011.

[21] Herodotos Herodotou, Harold Lim, Gang Luo, Nedyalko Borisov, Liang Dong, Fatma Bilgen Cetin, and Shivnath Babu. Starfish: A self-tuning system for big data analytics. In *Proceedings of the Biennial International Conference on Innovative Data Systems Research (CIDR)*, pages 261–272, January 2011.

[22] Herodotos Herodotou, Harold Lim, Gang Luo, Nedyalko Borisov, Liang Dong, Fatma Bilgen Cetin, and Shivnath Babu. Starfish: A self-tuning system for big data analytics. In *Cidr*, volume 11, pages 261–272, 2011.

[23] Avinash Hindupur. The gan zoo. https://github.com/hindupuravinash/the-gan-zoo.

[24] Jonathan Ho, Ajay Jain, and Pieter Abbeel. Denoising diffusion probabilistic models. *Advances in neural information processing systems*, 33:6840–6851, 2020.

[25] Nick Huang, Aaron Gokaslan, Volodymyr Kuleshov, and James Tompkin. The gan is dead; long live the gan! a modern gan baseline. In *NeurIPS 2024 Workshop on Mathematics of Modern Machine Learning*.

[26] Ajaykrishna Karthikeyan, Nagarajan Natarajan, Gagan Somashekar, Lei Zhao, Ranjita Bhagwan, Rodrigo Fonseca, Tatiana Racheva, and Yogesh Bansal. SelfTune: Tuning cluster managers. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 1097–1114, Boston, MA, April 2023. USENIX Association.

[27] Ana Klimovic, Heiner Litz, and Christos Kozyrakis. Selecta: Heterogeneous cloud storage configurations for data analytics. In *Proceedings of the USENIX Annual Technical Conference (ATC'18)*, 2018.

[28] Krasserm. bayesian-machine-learning. https://github.com/krasserm/bayesian-machine-learning.

[29] Guoliang Li, Xuanhe Zhou, Shifu Li, and Bo Gao. Qtune: A query-aware database tuning system with deep reinforcement learning. In *Proceedings of Very Large Data Base (VLDB)*, 2019.

[30] Min Li, Liangzhao Zeng, Shicong Meng, Jian Tan, Li Zhang, Ali R Butt, and Nicholas Fuller. Mronline: Mapreduce online performance tuning. In *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing*, pages 165–176. ACM, 2014.

[31] Guangdeng Liao, Kushal Datta, and Theodore L. Willke. Gunther: Search-based auto-tuning of mapreduce. In *Proceedings of the European Conference on Parallel Processing (Euro-Par'13)*, 2013.

[32] Guangdeng Liao, Kushal Datta, and Theodore L Willke. Gunther: Search-based auto-tuning of mapreduce. In *European Conference on Parallel Processing*, pages 406–419. Springer, 2013.

[33] Yang Liu, Maozhen Li, Nasullah Khalid Alham, and Suhel Hammoud. Hsim: a mapreduce simulator in enabling cloud computing. *Future Generation Computer Systems*, 29(1):300–308, 2013.

[34] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. Graphlab: A new framework for parallel machine lerning. In *https://arxiv.org/pdf/1408.2041.pdf*, 2010.

[35] Ashraf Mahgoub, Paul Wood, Sachandhan Ganesh, Subrata Mitra, Wolfgang Gerlach, Travis Harrison, Folker Meyer, Ananth Grama, Saurabh Bagchi, and Somali Chaterji. Rafiki: A middleware for parameter tuning of nosql datastores for dynamic metagenomics workloads. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference (Middleware)*, 2017.

[36] Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A system for large-scale graph processing. In *Proceedings of the Seventeenth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS)*, 2010.

[37] Jonas Mockus. Application of bayesian approach to numerical methods of global and stochastic optimization. *Journal of Global Optimization*, 4(4):347–365, 1994.

[38] Kay Ousterhout, Ryan Rasti, Sylvia Ratnasamy, Scott Shenker, and Byung-Gon Chun. Making sense of performance in data analytics frameworks. In *12th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 15)*, pages 293–307, 2015.

[39] Robin Rombach, Andreas Blattmann, Dominik Lorenz, Patrick Esser, and Björn Ommer. High-resolution image synthesis with latent diffusion models. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 10684–10695, 2022.

[40] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. Practical bayesian optimization of machine learning algorithms. In *Advances in neural information processing systems*, pages 2951–2959, 2012.

[41] Jascha Sohl-Dickstein, Eric Weiss, Niru Maheswaranathan, and Surya Ganguli. Deep unsupervised learning using nonequilibrium thermodynamics. In *International conference on machine learning*, pages 2256–2265. PMLR, 2015.

[42] Gagan Somashekar, Karan Tandon, Anush Kini, Chieh-Chun Chang, Petr Husak, Ranjita Bhagwan, Mayukh Das, Anshul Gandhi, and Nagarajan Natarajan. OPPerTune: Post-Deployment configuration tuning of services made easy. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 1101–1120, Santa Clara, CA, April 2024. USENIX Association.

[43] Apache Spark. Spark configuration. http://spark.apache.org/docs/latest/configuration.html.

[44] Authors Swift. Experimented configuration parameters for apache flink and spark. https://github.com/swift-tunning/Swift-Configuration.

[45] Apache Flink Team. Apache flink. https://flink.apache.org/flink-architecture.html. Accessed December 31, 2018.

[46] Apache Hadoop Team. Apache hadoop. http://hadoop.apache.org/. Accessed December 31, 2018.

[47] Apache Kafka Team. Apache kafka. https://kafka.apache.org/. Accessed January 10, 2019.

[48] Apache Spark Team. Apache spark. https://spark.apache.org/. Accessed December 31, 2018.

[49] Apache Spark Team. Apache spark configuration. https://spark.apache.org/docs/latest/configuration.html. Accessed July 16, 2019.

[50] Google Kubernetes Team. Kubernetes. https://kubernetes.io/. Accessed January 4, 2019.

[51] Intel HiBench Team. Hibench. https://github.com/intel-hadoop/hibench. Accessed July 25, 2022.

[52] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M. Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, Nikunj Bhagat, Sailesh Mittal, and Dmitriy Ryaboy. Storm@twitter. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2014.

[53] TPC. Tpc-ds is a decision support benchmark. https://www.tpc.org/tpcds/.

[54] Shivaram Venkataraman, Zongheng Yang, Michael Franklin, Benjamin Recht, and Ion Stoica. Ernest: Efficient performance prediction for large-scale advanced

analytics. In *Proceedings of the 13th USENIX Conference on Network Systems Design and Implementation (NSDI'16)*, 2016.

[55] Guanying Wang, Ali R Butt, Prashant Pandey, and Karan Gupta. A simulation approach to evaluating design decisions in mapreduce setups. In *2009 IEEE International Symposium on Modeling, Analysis & Simulation of Computer and Telecommunication Systems*, pages 1–11. IEEE, 2009.

[56] Kewen Wang and Mohammad Maifi Hasan Khan. Performance prediction for apache spark platform. In *2015 IEEE 17th International Conference on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cyberspace Safety and Security, and 2015 IEEE 12th International Conference on Embedded Software and Systems*, pages 166–173. IEEE, 2015.

[57] Shu Wang, Chi Li, William Sentosa, Henry Hoffmann, Shan Lu, and Achmad Kistijantoro. Understanding and auto-adjusting performance-sensitive configurations. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 154–168. ACM, 2018.

[58] Xintao Wang, Ke Yu, Shixiang Wu, Jinjin Gu, Yihao Liu, Chao Dong, Yu Qiao, and Chen Change Loy. Esrgan: Enhanced super-resolution generative adversarial networks. In *Proceedings of the European Conference on Computer Vision (ECCV) Workshops*, September 2018.

[59] Christopher KI Williams and Carl Edward Rasmussen. *Gaussian processes for machine learning*, volume 2. MIT press Cambridge, MA, 2006.

[60] Tianyin Xu, Long Jin, Xuepeng Fan, Yuanyuan Zhou, Shankar Pasupathy, and Rukma Talwadker. Hey, you have given me too many knobs!: Understanding and dealing with over-designed configuration in system software. In *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, pages 307–319. ACM, 2015.

[61] Tianyin Xu, Jiaqi Zhang, Peng Huang, Jing Zheng, Tianwei Sheng, Ding Yuan, Yuanyuan Zhou, and Shankar Pasupathy. Do not blame users for misconfigurations. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 244–259. ACM, 2013.

[62] Neeraja J. Yadwadkar, Bharath Hariharan, Joseph E. Gonzalez, and y Katz R. Multi-task learning for straggler avoiding predictive job scheduling. *The Journal of Machine Learning Research (JMLR)*, 17(11):3692–3728, 2016.

[63] Neeraja J. Yadwadkar, Bharath Hariharan, Joseph E. Gonzalez, Burton Smith, and Randy H. Katz. Selecting the best vm across multiple public clouds: A data-driven performance modeling approach. In *Proceedings of the 2017 ACM Symposium on Cloud Computing (SoCC)*, 2017.

[64] Zhibin Yu, Zhendong Bei, and Xuehai Qian. Datasize-aware high dimensional configurations auto-tuning of in-memory cluster computing. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 564–577. ACM, 2018.

[65] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *Proceedings of the second USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2010.

[66] Ciyou Zhu, Richard H. Byrd, Peihuang Lu, and Jorge Nocedal. Algorithm 778: L-bfgs-b: Fortran subroutines for large-scale bound-constrained optimization. *ACM Trans. Math. Softw.*, 23(4):550–560, dec 1997.

[67] Yuqing Zhu, Jianxun Liu, Mengying Guo, Yungang Bao, Wenlong Ma, Zhuoyue Liu, Kunpeng Song, and Yingchun Yang. Bestconfig: Tapping the performance potential of systems via automatic configuration tuning. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC'17)*, pages 338–350. ACM, 2017.

PMR: Fast Application Response via Parallel Memory Reclaim on Mobile Devices *

Wentong Li[1,2], *Li-Pin Chang*[3], *Yu Mao* [4,5], *Liang Shi*[1,2†]
[1]*MoE Engineering Research Center of Software/Hardware Co-Design Technology and Application, East China Normal University, Shanghai, China*
[2]*School of Computer Science and Technology, East China Normal University*
[3]*National Yang Ming Chiao Tung University, Taiwan*
[4]*City University of Hong Kong, Hong Kong,* [5]*MBZUAI, Abu Dhabi*

Abstract

Mobile applications exhibit increasingly high memory demands, making efficient memory management critical for enabling fast and responsive user experiences. However, our analysis of Android systems reveals inefficiencies in the current kernel-level memory reclaim design, which struggles to meet the performance demands of modern apps and fails to exploit upgraded storage devices. To address this challenge, we propose PMR, a parallel memory reclaim scheme. PMR introduces two key techniques: proactive page shrinking (PPS) and storage-friendly page writeback (SPW). PPS enhances the memory reclaim process by decoupling the time-consuming steps of page shrinking and page writeback for parallel execution, while SPW optimizes write I/O operations through batched unmapping of victim pages for bulk, efficient writeback. Experimental results on real-world mobile devices demonstrate that PMR can improve application response times by up to 43.6% compared to the original Android memory reclaim approach.

1 Introduction

The memory requirements of mobile applications are growing significantly, imposing high pressure on modern mobile devices [19, 28, 34]. Many applications now require gigabytes of memory, including popular social networking apps, video streaming apps, and web browsers. By contrast, the physical memory capacity of mobile devices has seen only modest increases, with flagship models such as the latest iPhone 16 series offering just 8 GB of DRAM [15], and the upcoming Samsung Galaxy S25 Ultra series equipped with 16 GB [16]. Running just a few such apps can quickly create significant memory pressure on mobile devices. If this pressure is not properly managed, the applications may experience slowdowns, restarts, or crashes, severely affecting the user experience [10, 20, 32].

To effectively alleviate memory pressure, Android-based mobile systems employ various memory reclaim mechanisms, the most notable being kernel-level memory reclaim (including memory swapping and direct reclaim) and user-space Low Memory Killer Daemon (LMKD) [13]. Specifically, first, when the amount of free memory falls below a pre-defined low watermark, the kernel wakes up a memory swapping thread (i.e., *kswapd*) for asynchronous memory reclaim in background [12]. Second, if *kswapd* fails to provide sufficient free memory for incoming requests, the kernel performs a direct reclaim for synchronous, blocking memory reclaim until the pending memory requests are satisfied [35]. Finally, if neither of these can recover the system from critically low memory, the system resorts to an LMKD process to release memory through selective process killing. However, once an app process is killed, the next time when the app is relaunched, users may notice a long restart latency and loss of all user context [25, 34]. In summary, LMKD termination can quickly relieve memory pressure but has serious side effects. In contrast, kernel-level memory reclaim is transparent to users and more cost-effective for mobile systems that emphasize interactive experience [19, 28, 34].

While LMKD should be avoided whenever possible, applying kernel-level memory reclaim in current systems is challenging. Our experiment reveals that current kernel-level memory reclaim is surprisingly sluggish in practice, failing to keep up with the growing memory demands of applications and being unable to fully utilize advanced mobile device hardware. Specifically, we observe that dozens of LMKD events can occur within just a few minutes (see Figure 2), suggesting the heavy reliance of current systems on frequent LMKD to alleviate memory pressure. As mentioned earlier, one of the primary reasons for this aggressive killing is the inefficiency of kernel-level memory reclaim. Also, while application memory footprints can easily reach several gigabytes, the memory

*This work is supported by the Shanghai Science and Technology Project (22QA1403300). This work is also partially supported by the National Science and Technology Council, Taiwan (Grant No. NSTC 113-2221-E-A49-188-MY3).

†The corresponding author is Liang Shi (shi.liang.hk@gmail.com).

reclaim throughput remains unexpectedly low, less than 150 MB/s at most (see Figure 3). The consistent upgrades for mobile devices have failed to reverse this trend, further confirming the existence of bottlenecks in kernel-level memory reclaim.

To this end, we are motivated to conduct in-depth analysis to illustrate the reason for the inefficient kernel-level memory reclaim. Experiments show that the current reclaim scheme is limited by the rigid reclaim path, and therefore cannot exert the performance of evolving storage devices. Specifically, the memory reclaim procedure follows a sequential execution flow in which two key steps slow down the overall process: (1) *Page Shrinking*, which extracts victim pages from the kernel page list for subsequent reclamation, and (2) *Page Writeback*, which checks whether the victim page is reclaimable and then writes them back to the flash-based storage device. Intuitively, the selection of the victim page should be very fast, and the bottleneck of memory reclaim is more on the I/O operation of page writeback. However, the time breakdowns of memory reclaim suggest that page shrinking and page writeback account for 55% and 45% of the total reclaim latency, respectively. Each round of memory reclaim spends more than one-half of the time waiting for the result of the page shrinking. The current page shrinking step uses multiple conditions to classify reclaimable pages, leading to pessimistic victim selection and repeated iterations to meet memory requirements. For page writeback, the unmapping operations are slow and create fragmented pages, limiting the use of the high write throughput of modern flash-based storage devices. This design can neither meet the demand for fast memory reclaim nor maximize the performance of storage devices.

Motivated by these observations, we propose PMR, a parallel memory reclaim method for fast application response on mobile devices. The core idea of our method is to redesign the kernel's existing memory reclaim path to improve page reclaim efficiency. Specifically, we decouple selected key steps in the reclaim process and parallelize them, thereby accelerating memory reclaim. PMR consists of two parts: proactive page shrinking (PPS) and storage-friendly page writeback (SPW). PPS enables asynchronous execution of page shrinking, preparing an adequate amount of victim pages ahead of memory reclaim bursts. SPW performs unified page unmap on an application-by-application basis and batches I/O operations to exploit the internal parallelism of flash storage devices. Our evaluation results show that PMR prevents LMKD from terminating processes and guarantees quick response time for the running applications compared with the original kernel.

The major contributions of this paper are as follows.

- We firstly benchmark the memory reclaim on mobile devices and reveal the sluggish kernel-level memory reclaim. And then we identify the root cause: (1) sequential execution of page shrinking and page writeback introduces unnecessary delays; (2) page shrinking does not prepare sufficient victim pages, incurring frequent memory reclaim calls; (3) the page unmap mechanism generates fragmented page write I/Os, which under-utilizes the internal parallelism of flash storage.
- We introduce PMR to enhance the original memory reclaim process, involving proactive page shrinking and storage-friendly page writeback. The former decouples page shrinking and page writeback, and the latter performs storage-friendly I/Os to accelerate page writeback.
- We deploy PMR on real mobile devices. The experimental results show that when enabled PMR, the application response time is reduced by 43.6% compared to the original system memory reclaim approach.

2 Background and Motivation

2.1 Memory Reclaim on Mobile Device

Linux-based Android mobile devices have three memory reclaim schemes to manage memory pressure: memory swapping, direct reclaim and LMKD. First, when the system detects that the free memory watermark is lower than a predefined threshold ($watermark_{low}$), memory swapping is triggered to perform asynchronous memory reclaim with a kernel thread *kswapd*. Second, if the free memory level is still critically low, direct reclaim is activated to perform synchronous memory reclaim. Third, if the kernel-level memory reclaim (memory swapping and direct reclaim) fail to relieve the memory pressure, Android provides LMKD to release memory by killing processes selectively.

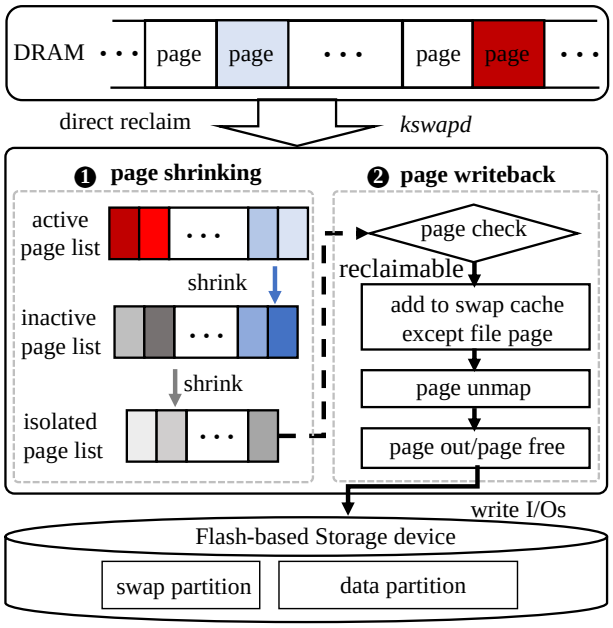


Figure 1: Traditional kernel-level memory reclaim path on mobile device

Although LMKD effectively alleviates memory pressure, killing processes cause loss of the application context and long restart of applications. By contrast, kernel-level memory swapping and direct reclaim are preferred [29, 31, 34]. Recently, mobile device manufacturers, such as Samsung [43], Huawei [21], have included flash-based memory swapping for memory expansion [1,21,43,54]. Figure 1 shows that both memory swapping and direct reclaim follow the two steps for page reclaim: (1) Page Shrinking and (2) Page Writeback.

Page Shrinking. The kernel organizes memory pages into four LRU page lists per memory zone for each NUMA node: inactive anonymous page list, active anonymous page list, inactive file page list, and active file page list. When memory reclaim is triggered, the kernel demotes infrequently used pages from the active page list to the inactive page list, as well as takes out infrequently used pages from the inactive page list for page writeback. Whenever memory reclaim occurs, the kernel calculates the scan ratio for each page list. The `nr` array logs the scanned page count for each page list. Specifically, the kernel executes page shrinking in two steps: First, memory swapping or direct reclaim determines the total number of pages to be reclaimed (`nr_to_reclaim`) and the number of pages to be scanned for each round `nr_to_scan`. Second, the kernel scans all four LRU lists for `nr_to_scan` pages. The scan continues until the array `nr` is empty or the reclaimed pages (`nr_reclaimed`) reach the reclaim target. The victim pages found during the scan are added to the temporary `page_list` to be written back later.

Page Writeback. Once the pages are collected, the kernel performs page-by-page writeback, involving the following steps: First, the kernel performs a page check to confirm whether it can enter the subsequent reclaim process, including page references, page locks, etc. Second, when the page passes the check, the anonymous page is added to the swap cache and allocated a page table entry of swap space. Third, the system performs page unmap and page out for both anonymous and dirty file pages. Specifically, page unmap sets the page table entry mapping the anonymous page to the previously allocated swap type table entry or clears the process page table entry mapping the file page. Page out calls `page->mapping->a_ops->writepage` in the page descriptor to write back the page asynchronously. Finally, the page reclaim is complete, and the memory space occupied by the page is free.

2.2 Preliminary Study of Memory Reclaim

To understand the performance characteristics of the memory reclaim process, we performed extensive experiments on multiple mobile devices equipped with different sizes of memory and storage, including Google Pixel 5, Redmi Note 11 and Google Pixel 6 pro, as shown in Table 1. The experiments involve popular applications to reflect realistic user scenarios. Detailed experimental settings can be found in Section 5. We enabled flash-based swapping with a 2 GB swap partition on all these mobile devices. Our evaluation consists of two parts: First, we measure the frequency of the LMKD process and its impact on application response time. The time refers to the system's duration on transitioning from one application running in the foreground to another, showing the need for efficient memory reclaim. Second, we demonstrate the limitations of the current memory reclaim performance by showing that it fails to keep pace with the growth of app memory footprints.

Table 1: Mobile Devices Under Evaluation.

Model	Google Pixel 5	Redmi Note 11	Google Pixel 6 pro
CPU	Qualcomm Snapdragon 765G	MediaTek Dimensity 810	Google Tensor
Memory	8 GB	6 GB	12 GB
Storage	128 GB/UFS 2.1	128 GB/UFS 2.2	256 GB/UFS 3.1
System	Android 13 (Kernel 5.10)	Android 13 (Kernel 5.10)	Android 13 (Kernel 5.10)
Announced	2020, September	2022, January	2021, October

2.2.1 Frequent, Expensive Process Killing

To quantify the execution behaviors of LMKD, we composed a workload that switches back and forth among 36 applications with various app-use durations and employed the `logcat` command [11] to collect LMKD killing events. To avoid bias, we present the average of ten rounds of application response time.

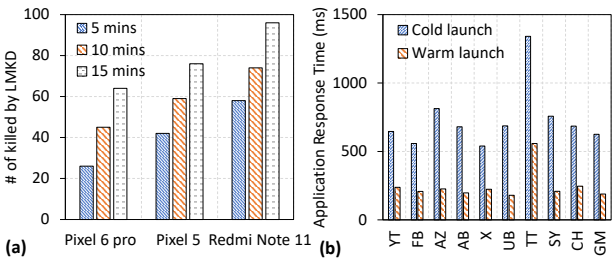


Figure 2: Frequent occurrences of LMKD have a large negative impact on application responsiveness. (a) Many processes are killed among different mobile devices; (b) Applications suffer from long restart latency when the processes are killed on Pixel 6 pro. (YT: YouTube, FB: Facebook, AZ: Amazon, AB: Angry Birds, X: Twitter, UB: Uber, TT: TikTok, SY: Spotify, CH: Chrome, GM: Gmap.)

As shown in Figure 2(a), even though the newest Pixel 6 pro has 12 GB of DRAM, it still experienced 26 process killings after five minutes of workload replay, with more processes being killed by LMKD as the workload ran longer. By contrast, because Pixel 5 and Redmi Note 11 feature small amounts of DRAM, they experienced more LMKD killings under the same workload. Here, switching to an application killed by

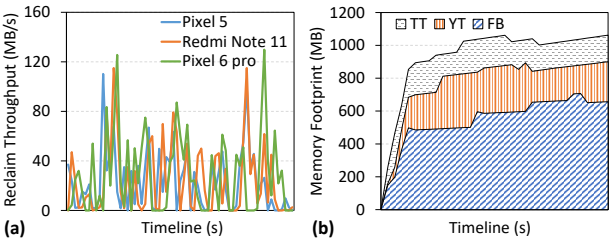


Figure 3: Low memory reclaim throughput accompanying growing memory requirements of applications. (a) Memory reclaim throughput barely improves on fast storage; (b) Extremely high memory demands of applications.

LMKD requires a restart (cold launch). Figure 2(b) shows that the application response time is 3.1 times longer on average than the latency of the warm start of the application, that is, switching to a live cached application. The frequent LMKD killing events degrade application responsiveness and severely impact user experience. Considering the triggering conditions of LMKD mentioned above, frequent LMKD comes from the ineffective kernel-level memory reclaim that fails to alleviate memory pressure in time.

2.2.2 Sluggish Memory Reclaim

To confirm that kernel-level memory reclaim does not match app memory demands, we measured kernel reclaim throughput and app runtime memory footprint. Once the system performs memory reclaim, the kernel writes dirty file pages and anonymous pages back to the storage device. Considering that other processes in the system will also perform write operations for file pages during the memory reclaim, we employ the write throughput of the swap partition as a reliable indicator of the memory reclaim throughput, And there is indeed a high percentage of reclaimed anonymous pages that need to be written back to the swap partition [50].

Figure 3 shows our two findings. First, as shown in Figure 3 (a), memory reclaim throughput fluctuates below 80 MB/s most of the time, and interestingly, reclaim performance remains consistently low across different generations of mobile devices. Specifically, while Pixel 6 pro features faster flash storage than Pixel 5 (UFS 3.1 versus UFS 2.1), the average reclaim throughput of Pixel 6 pro is only 10% higher than that of Pixel 5. On the other hand, Figure 3(b) shows that the memory footprint size of three popular applications expands quickly during the initial startup phase, reaching between 400 MB and 1 GB. It is evident that the memory reclaim process falls far behind the application memory demands.

In summary, **while app memory footprint is increasingly high, the hardware advance does not benefit the efficiency of kernel memory reclaim. This inefficiency results in frequent LMKD killings, which has a negative impact on the user experience. These findings direct our further investigation toward analyzing the memory reclaim procedure.**

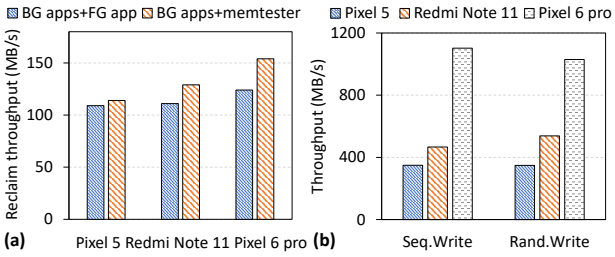


Figure 4: (a) I/O performance w/ or w/o I/O conflict. (b) Random and sequential write throughput with 4 KB I/Os.

3 Performance Bottleneck Analysis

As mentioned in Section 2.1, the kernel-level memory reclaim involves scanning kernel pages and accessing the flash storage. Based on these two key steps, we examine the root causes of memory reclaim stalls.

3.1 I/O Utilization

Intuitively, the stagnant memory reclaim may be first attributed to poor I/O performance, which includes two aspects: I/O conflicts and the storage device itself. According to previous studies, the write I/Os of memory reclaim may conflict with the read I/Os of other processes, and I/O conflicts are considered to be an important factor affecting application performance [18, 30, 39]. However, the storage bandwidth determines the upper limit of the memory reclaim.

3.1.1 Low Impact of I/O Conflicts

To quantify the impact of I/O conflicts on memory reclaim, we conducted two workload scenarios: (1) Saturating memory pressure by running multiple background applications and provoking memory reclaim by switching applications to foreground (`BG-apps+FG-app`); (2) Running background applications and use the `memtester` command [48] to allocate memory and provoke memory reclaim (`BG-apps+memtester`). `BG-apps+FG-app` represents a scenario in which memory reclaim suffers from intensive I/O conflicts because switching applications to the foreground involves reads from the storage. In contrast, memory reclaim requires writing pages to the flash storage. `BG-apps+memtester` represents another scenario in which memory reclaim is almost free from I/O conflicts because `memtester` requests for memory without generating additional read I/O operations.

Figure 4 (a) shows the peak memory reclaim throughput under both workloads on different mobile devices. Taking Pixel 6 pro as an example, the peak reclaim throughput with `memtester` (light conflict) is only 8.3% higher than the throughput with switching applications to the foreground (severe conflicts), and similar results are displayed by Pixel 5 and Redmi Note 11. In other words, I/O conflicts have a low impact on memory reclaim throughput. This is because reads

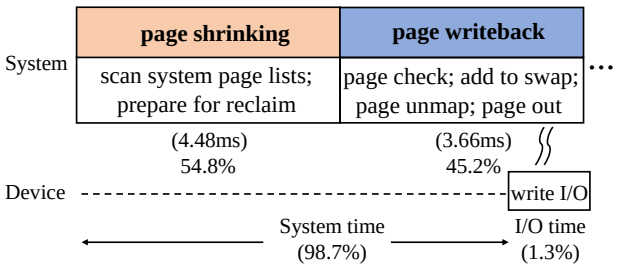


Figure 5: Average time breakdown of memory reclaim path.

are much faster than writes in flash storage, and under I/O conflicts, they only lightly affect the latency of swap writes [47].

3.1.2 Excessive Storage Bandwidth

Storage bandwidth is also a critical factor for overall I/O performance. We measure the write throughput of flash storage using the `fio` tool [2], with the results shown in Figure 4(b). The internal 128 GB UFS 2.1 flash storage in the lowest profile Pixel 5 achieves a random write throughput of 378 MB/s, three times higher than the memory reclaim throughput reported in the previous section. By contrast, the high-profile Pixel 6 pro with UFS 3.1 flash storage demonstrates an even greater capability, achieving throughput at around 1000 MB/s for random and sequential writes. In other words, the memory reclaim process significantly under-utilizes the storage bandwidth, suggesting the inefficiency lies within the system software rather than the hardware.

3.2 Sub-optimal Memory Reclaim Path

To identify bottlenecks in the memory reclaim path, we dissect the time overhead by measuring the delay of each major step using the kernel command *ktime* [3].

3.2.1 Sequential Execution of Key Steps

Figure 5 shows a breakdown of the total memory reclaim latency, contributed by page shrinking and page writeback: First, for each round of memory reclaim, the page shrinking step spends 4.48 ms on average, which accounts for 54.8% of the total memory reclaim time. The page writeback step contributed to the rest 45.2%, using 3.66 ms on average. Interestingly, although the page shrinking step does not involve any I/Os, it uses more than one-half of the total latency just to prepare victim pages for reclamation. Second, each round of page reclaim goes through a serial flow of page shrinking and page writeback, i.e., page writeback must wait until victims have been prepared by page shrinking. In other words, this sequential execution flow significantly limits the potential benefits of enhanced flash storage performance in improving memory reclaim efficiency.

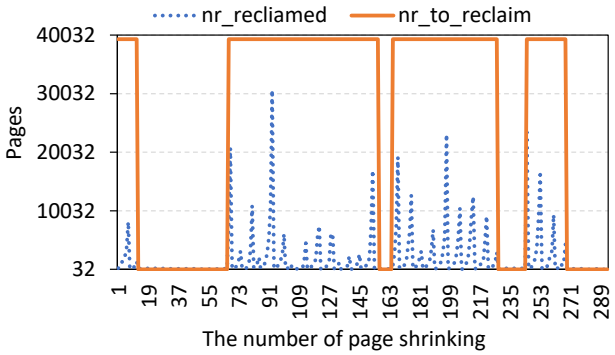


Figure 6: nr_to_reclaim vs nr_reclaimed when system performs memory reclaim.

3.2.2 Inefficient Page Shrinking

As mentioned earlier, the kernel-level memory reclaim consists of memory swapping and direct reclaim. For memory swapping, the expected number of pages reclaimed in each round of invocation is the maximum of `SWAP_CLUSTER_MAX` and `watermark_high` (whose default value is 39340 pages, approximately 154 MB). For direct reclaim, this value is `SWAP_CLUSTER_MAX`, 32 pages by default.

Figure 6 illustrates the expected number of pages to be reclaimed (`nr_to_reclaim`) versus the actual number of pages reclaimed (`nr_reclaimed`) during runtime. When memory swapping executes, although it is expected to reclaim 39,340 pages, the actual number it reclaims is far fewer than the expected value. This is because, during memory reclaim, the page shrinking step examines `nr_to_reclaim` LRU pages in the kernel page lists, but only a small portion of these pages can be reclaimed. Specifically, pages that have been recently referenced or are locked for page writeback cannot be reclaimed. Additionally, page shrinking terminates early if it experiences highly unbalanced scanning between anonymous pages and file pages [14]. As a result, page shrinking is repeatedly invoked to gather sufficient reclaimed pages, making the entire procedure inefficient.

3.2.3 Internally Blocked Page Writeback

Page writeback is responsible for performing the actual page reclaim based on the page type, including page checking, unmapping victim pages, and writing the pages to storage, etc. Intuitively, I/Os for page writing may be the performance bottleneck, but our analysis reveals a different result.

Figure 7 shows a breakdown of the page writeback step into unmapping pages and writing pages (page out). Unexpectedly, the time overhead of page unmap is relatively high and extremely unstable. This is because to unmap a page, page table entries of all processes that share the page must be modified through a reverse mapping mechanism, and this procedure is prone to interruption, so its completion time is extremely unstable [34]. Moreover, not only unmapping but

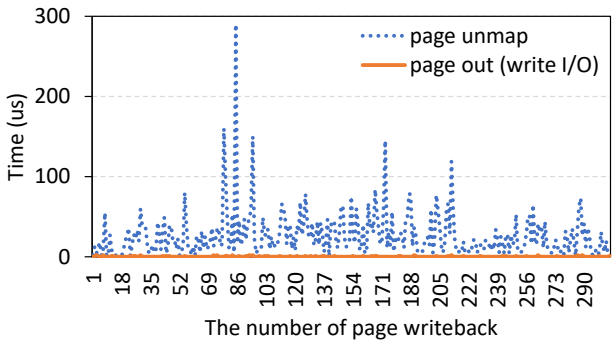


Figure 7: Time breakdown of page writeback (the time of page out is not zero but fluctuates under a few microseconds).

also writing of pages is performed page by page. Writing flash at the page granularity is inefficient and slows down the whole writeback step.

In summary, **sluggish memory reclaim is mainly due to the sub-optimal memory reclaim path, including three aspects: (1) page writeback waits on the results of page shrinking, introducing unnecessary delays; (2) page shrinking suffers from inefficient reclaim and repeated invocation; (3) page unmap latency is highly unstable and produces small writes that cannot unleash the potential of flash storage.**

4 PMR Design

4.1 Overview

The basic idea is of PMR to parallelize key steps of memory reclaim to fulfill application memory demand and thus improve application response. Intuitively, there are two typical ways to perform parallel memory reclaim. First, the kernel can create multiple memory reclaim threads, e.g., Mutiplek-swapd [36] wakes up multiple kswapd threads to perform memory swapping to relieve memory pressure. Second, it is possible to exploit the performance advantage of flash storage through bulk I/Os. For example, SEAL [29] performs memory swapping in units of applications, rather than page by page. However, for the former, multi-threaded memory reclaim is prone to conflict with each other and burden the CPU because they occur simultaneously. For the latter, simply increasing the I/O size still does not resolve the suboptimal execution flow of memory reclaim. Unlike previous work, PMR parallelizes key steps in the memory reclaim path based on observations in Section 3.2.

To realize the basic idea, PMR consists of two components: proactive page shrinking (PPS) and storage-friendly page writeback (SPW). Figure 8 shows an overview of PMR. PPS decouples page shrinking and page writeback to prepare sufficient victim pages in advance, and it includes a kernel thread *kshrinkd* and a victim page list to perform page shrinking independently. Specifically, its operation involves three steps: (❶) Based on pre-defined conditions, *kshrinkd* is activated. (❷) An independent page shrinking procedure moves pages from the system page LRU lists to the victim page list. (❸) The new victim page list is carefully maintained to ensure sufficient pages are provided for page writeback.

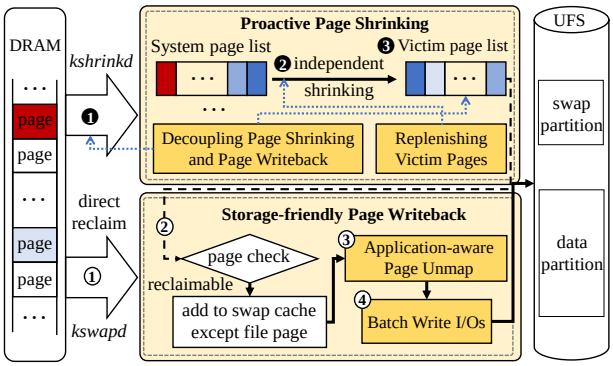


Figure 8: Overview of PMR.

SPW performs application-aware page unmap, and batches write I/Os for efficient page writeback. It redesigns the page unmap process to align with storage-friendly I/O sizes, enabling bulk writes that exploit the internal parallelism of flash storage. Specifically, it works in four steps: (①) When the system runs low on memory, *kswapd* or direct reclaim is triggered for memory reclaim. (②) Unlike the existing design, page writeback is triggered immediately since *kshrinkd* may already have prepared sufficient pages in the victim list. (③) Reclaimable pages undergo application-aware page unmap. (④) A batch of unmapping pages are written to flash storage through efficient bulk I/Os.

There are two key challenges in the implementation of PPS and SPW. First, *kshrinkd* is supposed to prepare sufficient victim pages, subject to the timing of the shrinkage and the granularity of each. The shrinking must be carefully timed to avoid conflicts between page shrinking and other processes accessing pages. The shrinking granularity should meet the need to reclaim memories without creating waste. And the victim page list needs careful maintenance to control overhead. Second, the batch size of the application-aware page unmap must be carefully controlled. The batch size should be sufficiently large to ensure efficient write I/Os, yet not so large as to introduce unnecessary delays in page writeback. The following sections detail the design of PPS and SPW.

4.2 Proactive Page Shrinking

To produce an adequate supply of victim pages for page writeback at a low cost, PPS decouples page shrinking and page writeback and carefully determines the timing and granularity of page shrinking.

4.2.1 Decoupling Page Shrinking and Page Writeback

Existing kernel-level memory reclaim follows the sequential flow of page shrinking first and then page writeback. When the system suffers memory pressure, page shrinking first isolates victim pages to the temporary `page_list`, and page writeback performs subsequent page reclaim.

Two challenges arise regarding the decoupling of page shrinking and page writeback. First, current page shrinking and page writeback are operated by the same thread. Concurrent execution of the two steps is required to achieve the decoupling. Second, the existing page shrinking design uses function `isolate_lru_pages` to temporarily remove eligible pages from the system LRU page lists for subsequent page writeback. With decoupled page shrinking and page writeback, access to the eligible pages between page shrinking and page writeback requires coordination. To address these two challenges, firstly, page shrinking is offloaded to a new kernel thread *kshrinkd*, and page writeback is left to memory swapping and direct reclaim. Secondly, the temporary `page_list` isolated by page shrinking have been redesigned as a new victim page list to coordinate the page production and consumption between *kshrinkd* and page writeback.

Independent *kshrinkd* Thread. *kshrinkd* is designed to perform page shrinking independently and prepare sufficient victim pages in advance. We standardize *kshrinkd* runtime rules to achieve this goal. First, the triggering of *kshrinkd*, no longer depends on memory pressure, but on whether enough victim pages are prepared. When the system faces memory pressure, kernel-level memory reclaim directly accesses the pages produced by *kshrinkd* to perform page writeback. Second, unlike existing page shrinking, which may terminate early without producing the expected number of victim pages, such as when `nr_anon` or `nr_file` are zero, our enhanced *kshrinkd* uses a continuous scanning loop to ensure the target is reached. Finally, to minimize changes to the kernel, *kshrinkd* inherits the page shrinking algorithm from the existing memory reclaim, e.g., calculating the number of pages that need to be scanned for each LRU page list.

Victim Page List Maintenance. When page shrinking and page writeback can be done in parallel with the help of *kshrinkd*, we need to consider how to preserve the victim pages that were produced by *kshrinkd*. The traditional LRU isolation mechanism is limited by factors such as lock contention, and the number of pages that can be isolated each time is limited. Based on this, we maintain a new victim page list and filter eligible pages from the system LRU list by *kshrinkd*, including those pages that are not locked and not frequently referenced. Specifically, we use a reserved bit in the page table entry to record these pages as isolated pages (`PG_ISOLATED`). And *kshrinkd* will move these pages from the system LRU list to the victim page list. Also, to avoid access conflict to the victim page list, lightweight spinlocks are used to ensure normal page accesses for *kshrinkd* and page writeback. In this way, we can reserve sufficient pages in the victim page list to perform efficient page isolation services for subsequent page writeback.

Algorithm 1: Always Ready Page Shrinking

```
Input: nr_victim_page: the actual number of pages in the victim
       page list (The initial value is 0);
nr_ideal_victim_page: the ideal number of pages in the victim page
list (The initial value is δ);
shrink_size: the number of page shrinking each time ;
if System Start then
    while nr_victim_page < nr_ideal_victim_page do
        kshrinkd starts;
        shrink_size = nr_ideal_victim_page ;
    kshrinkd sleep;
Once page writeback is required, provide victim pages;
if Memory Swapping or Direct Reclaim Start then
    while nr_victim_page < nr_ideal_victim_page do
        kshrinkd starts;
        shrink_size = nr_to_reclaim ;
```

4.2.2 Replenishing Victim Pages

The decoupling of page shrinking and page writeback introduces the challenge of ensuring that *kshrinkd* maintains an adequate supply of victim pages for page writeback. To address this challenge, a new control loop is introduced, as detailed in Algorithm 1, which describes our always-ready page shrinking procedure. At system startup, *kshrinkd* is activated, and it continues the collection process until *nr_ideal_victim_page* pages have been inserted into the victim page list. Once this target is met, *kshrinkd* transitions to a sleep state. On memory reclaim requests, the victim pages from the new page list are isolated for page writeback, while in the meantime, the page shrinking procedure is reactivated to replenish victim pages. This ensures a consistent supply of victim pages for writeback, maintaining system efficiency and readiness. Further details are discussed below.

Page Shrinking Watermark. To ensure sufficient victim pages, we set a page shrinking watermark. The parameter *nr_ideal_victim_page* (i.e., δ in Algorithm 1) specifies the target number of victim pages that should be maintained in the new victim page list. To prevent delays in page writeback due to insufficient victim pages, *nr_ideal_victim_page* must be large enough, and its value should be carefully determined through an analysis of application responsiveness. Our sensitivity study in Section 5.5 recommends an empirical setting of 462 MB, ensuring a sufficient supply of victim pages for efficient memory reclaim.

Timely Page Shrinking. The efficiency of our page shrinking design depends on both the timing of *kshrinkd* activation and the batch size of page shrinking. Unlike the on-demand approach of existing memory reclaims designs, which is triggered by low watermarks or allocation rates, our design ac-

tivates page shrinking ahead of memory bursts. At system startup, *kshrinkd* collects victim pages until *nr_victim_page* reaches *nr_ideal_victim_page*, then enters a sleep state. Later, when system memory reclaim is invoked, even if the victim page list is not yet empty, *kshrinkd* is reactivated for parallel execution. In this case, the shrinking batch size is set to `nr_to_reclaim`, one third of *nr_ideal_victim_page*, and iterations continue until *nr_ideal_victim_page* victim pages have been prepared in the list.

In the design of PPS, the system needs to maintain a victim page list as a communication bridge between page shrinkage and page writeback. To circumvent page thrashing (between the system LRU list and the victim page list), a novel page type, the isolated page, is introduced as the victim page selection criterion for page shrinking. This inevitably raises concerns about the performance overhead of maintaining these pages. However, PPS merely screens isolated pages in advance, which can enhance the efficiency of page isolation operations. Furthermore, the parameter *nr_ideal_victim_page* determines whether the system can provide sufficient victim pages for writeback. The default setting of 462 MB is equivalent to three times the fixed number of pages for each memory swapping. Although different applications appear to require different amounts of memory to be reclaimed, it is challenging to set different default settings according to the types of applications. The current fixed default values are derived from sensitivity experiments and are committed to ensuring the performance of most applications. A more detailed maintenance design of the victim page list will be the focus of future work.

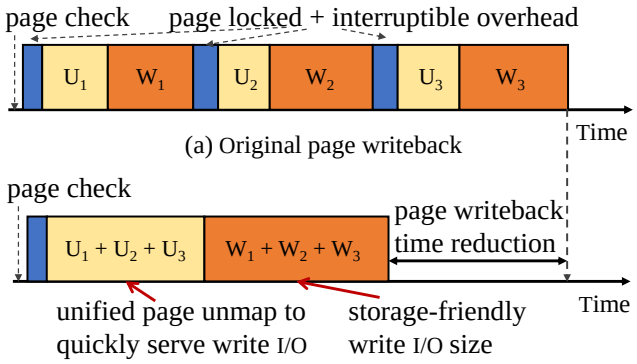


Figure 9: Page writeback scenarios with and without storage-friendly page writeback. U_i is the i^{th} page unmap, and W_i is the corresponding write I/O. SPW expedites paged writeback by exploiting the parallelism of flash storage devices.

4.3 Storage-friendly Page Writeback

A victim page must first be unmapped before being written back to the flash storage. Even though our page shrinking design collects batches of victim pages (i.e., `nr_to_reclaim`), with the existing page writeback procedure, pages are written back to flash storage in a page-by-page manner (4 KB each). In addition, as previously detailed in Section 3.2.3, the page unmap process suffers from highly unpredictable delays, affecting page writeback efficiency. To address these performance issues, SPW introduces application-aware page unmap followed by batched page-out (page writeback).

Figure 9 compares the original page writeback process and our SPW design. Unlike the original approach, SPW separately clusters page unmap operations and page write actions. The clustered page unmap, called an application-aware unmap, mitigates the cost of locking, avoids interference from other contending processes, and accumulates multiple pages for storage-friendly write I/Os. Subsequently, pages can be written back through bulk write I/Os, which better exploits the internal parallelism of flash storage.

4.3.1 Application-aware Page Unmap

There have been prior methods for resizing unmap job sizes to avoid interference from other processes and kernel activities [34]. In contrast, SPW takes a different approach by introducing an application-aware page unmap to enable storage-friendly write I/O operations. The core idea is to organize pages based on their associated processes and determine the unmap job size according to the performance characteristics of the underlying flash storage. Specifically, the application-aware unmap design must carefully handle interrupts to avoid long tail latency and exploit the performance gains of bulk write I/Os on flash storage, effectively balancing throughput and latency.

To minimize interference from other processes and kernel activities and improve the unmapping efficiency, SPW exploits the big-LITTLE processor architecture commonly found in modern mobile devices [46]. Specifically, in addition to increasing the job size for page unmap, SPW assigns a high priority to the page unmap thread and affiliates the thread with a big core, ensuring that the thread is not preempted during execution. With the privileged execution of the application-aware unmap procedure secured, the next focus is on determining the optimal job size for the unmap procedure based on the performance efficiency of flash storage.

4.3.2 Batch Write I/Os

Application-aware page unmap removes the obstacle for implementing bulk write I/Os, which can improve the efficiency of memory reclaim by utilizing the parallelism of storage devices. Next, we analyze the characteristics of flash-based storage devices to find the most cost-effective I/O size.

As shown in Figure 10, write throughput improves with larger I/O sizes, though the improvement is non-linear. The marginal gain saturates when I/Os are large enough to fully utilize parallelism, but increasing the I/O size further introduces high latency. On the Google Pixel 6 pro, a 10 MB I/O

size achieves 1261 MB/s, effectively balancing throughput and latency. Interestingly, cumulative write throughput does not benefit from additional threads due to lock contention. Based on these insights, our design collects unmapped pages from the new victim page list and composes them into bulk write I/Os based on the device-specific size. To address potential performance challenges with larger unmap sizes, a sensitivity analysis of this parameter will be presented in Section 5.5, to ensure an efficient balance between unmap and writeback performance.

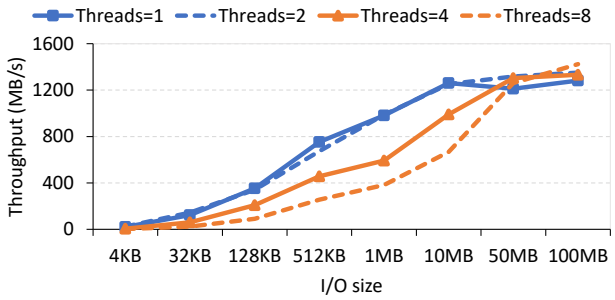


Figure 10: The throughput for random writes varies with the sequential block size and the number of threads in flash memory on Google Pixel 6 pro.

While the existing kernel handles pages individually, our SPW technique batches page unmapping and submits groups of pages to the block layer in bursts. As SPW simply increases the granularity of page unmapping, it inherits the page check and lock mechanisms from the existing kernel with minimal modifications. Our SPW is also orthogonal to the I/O plugging [24] mechanism, which operates at the block layer and is opaque to the page reclaiming process. By contrast, our SPW prepares batched I/O at the system software layer, allows more precise timing control, accelerating I/O submission and reducing merge latency.

4.4 Implementation and Discussions

Implementation Detail. Our implementation of PMR follows the principle of a minimally invasive approach, reusing existing kernel functions whenever possible. This section focuses on PMR's implementation aspects.

To implement *kshrinkd* for page shrinking, the `kshrinkd_init` function is executed as part of the kernel initialization process within `start_kernel`. Within `kshrinkd_init`, `kthread_run` is invoked to create a *kshrinkd* thread for each memory node, similar to *kswapd*. The page shrinking mechanism previously managed by *kswapd* is transferred to *kshrinkd*, with its activation now depending on the insufficiency of pages in the new victim page list, rather than the original *kswapd* activation conditions. Additionally, a new page list type, `LRU_VICTIM`, is introduced as an intermediary between page shrinking and page writeback. The *kshrinkd* thread scans the original LRU page lists and moves eligible victim pages to the new list, enabling page writeback to swap out these pages efficiently. To further optimize performance, the page unmap mechanism performs batch page writes based on the device-optimal I/O size. Using the `/proc` interface, our implementation allows for dynamic adjustment of the unmap unit size (`mem_unmap_unit`) on-the-fly.

Orthogonality. To keep our design simple and focused, several default memory reclaim settings are retained in this paper. First, although studies have shown that LRU-based page replacement strategies may not be optimal [32, 34, 51], our implementation inherits the default LRU-based victim page selection strategy. This decision is based on the fact that PMR is compatible with any page selection strategy and achieves performance improvements regardless of the strategy used. Second, as previous studies have extensively explored optimizing the trigger conditions for memory reclaim [19, 27, 30, 58], we choose not to modify them. Instead, PMR focuses on accelerating the execution of memory reclaim and serves as a complementary approach to these efforts. Finally, despite research on optimizing memory reclaim by adjusting its size [32, 50], we do not modify the original kernel setting, as PMR operates with its own job size for page shrinking and page writeback.

Future Use Cases of PMR. The deployment of large language models (LLMs) [6, 37, 55] on mobile devices imposes significant memory pressure on mobile systems, leading to severe performance degradation for such memory-demanding applications. On the other hand, the performance of mobile storage devices based on NAND flash has seen significant improvements, with the latest UFS 4.0 delivering a 1.8 times higher I/O bandwidth compared with UFS 3.1 [4, 8, 38, 45]. With such a high flash storage bandwidth, there is great potential to reshape the memory reclaim process.

5 Evaluation

5.1 Experiment Setup

Evaluation Platforms and Workloads. We adopt different real mobile devices as our experiment platform, as mentioned in Table 1, and flash-based memory swapping is enabled for memory swapping on these mobile devices, and the swap partition is set to 2 GB.

Experiments are conducted by following the three steps: (1) We installed the pre-selected 36 applications (10 switching applications and 26 background running applications) on the device to begin each experiment. (2) We used *adb* [9] command to collect evaluation results while performing automated tests with UI Automator [53] that emulates UI touches of users. *adb* is a versatile command-line toolkit, and we can use the command "adb shell starts the app's package name (PKN)" to start apps and record the launch latency. UI Automator is adopted to generate pseudo-random streams of user events, such as clicks, touches, or gestures, as well as a

number of system-level events. The app's PKN and the total number of user events we want to generate are provided to run this tool. Specifically, we use UI Automator to run background applications and adopt *adb* to switch the selected ten applications and record the launch latency as response time. We performed the same automated tests in ten rounds and calculated the average to avoid basis. The order of application switching is changed randomly in each round.

Table 2: Applications and automated user interaction.

Category	Foreground Applications	Auto user inputs
Browser	Chrome	Browse/Read posts
Social Network	Facebook, Twitter	Browse/Read posts
Multimedia	YouTube, Tiktok	Watch videos
Business Utility	Amazon, Gmap, Uber, Spotify	Browse and search Listen music
Game	Angry Birds	Play a stage

∗Background applications: Browser (Firefox, Opera), Social Network (WhatsApp, Instagram, Skype, WeChat, LinkedIn), Multimedia (Spotify, MXPlayer, Netflix, Capcut), Online shopping (Taobao, eBay, AliPay, BOA, Paypal), Business Utility (Booking, Gmail, New York Times, BBC News, OfficeMobile, GoogleDrive), and Game (Hill Climb Racing, Boom Beach, ClashRoyale, Call of Duty).

Evaluated Schemes. Five schemes are implemented and measured to show the effectiveness of PMR. **1) Original Memory Reclaim (OriginalMR)** represents the baseline of evaluation, which is inherited from native Linux. Inefficient memory reclaim results in memory pressure and cannot be relieved in time, causing processes to be killed frequently. **2) Acclaim [32]** represents the state-of-the-art work that is closest to this study, which relocates free pages from background applications for foreground applications and adjusts the size of *kswapd* according to the predicted allocation workloads. **3) Fleet [19]** represents the latest memory reclaim optimization idea on mobile devices, which proposes a foreground-aware GC-swap to combine Android Runtime (ART) GC and kernel-based memory swapping. **4) PMR** represents our proposed framework, which combines PPS and SPW.

5.2 Application Response Evaluation

To understand the benefit of PMR on application performance, application response time is evaluated among existing memory reclaim schemes. Application switching may expose memory reclaim when the system is under memory pressure. Different memory reclaim schemes can lead to different application response times due to performance differences. It is a critical performance metric for smartphone users.

Figure 11 shows the application response time of the evaluated schemes. First, OriginalMR suffers the worst response time, several times higher than other schemes. This is because when the system is under memory pressure, original memory reclaim fails to relieve memory pressure in time, and the system tends to kill the application. In this case, the application response time is equivalent to a cold launch. Second, Acclaim and Fleet reduce the response time by an average of 32.4%, 35.7% compared to OriginalMR, respectively. This is because Acclaim reduces direct reclaim and page fault by a foreground-aware page eviction and a dynamical size of the *kswapd*. Fleet designs kernel-based memory swapping and ART GC together to avoid mutual interference between different levels of the system memory reclaim.

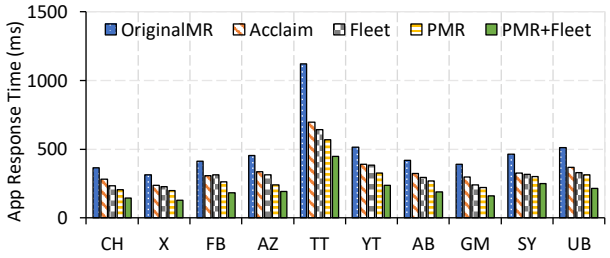


Figure 11: Application response time among different memory reclaim schemes on Google Pixel 6 pro.

Unlike previous works, PMR achieves fast memory reclaim to boost application response by proactive page shrinking and storage device-friendly write I/O. The application response time decreases significantly compared with OriginalMR, with the decreases reaching 43.6%. Also, PMR also has significant performance improvements over Acclaim and Fleet. More importantly, thanks to PMR's focus on improving the common memory reclaim path, it is perfectly complementary to both Acclaim and Fleet. For example, when PMR and Fleet are used together, application response time is reduced by 67.4% compared to OriginalMR and improved by 38.9% compared to Fleet alone. This is because Fleet effectively improves the accuracy of memory reclaim and avoids invalid memory reclaim, while PMR can speed up memory reclaim.

5.3 Memory Reclaim Evaluation

To further understand the performance improvement brought by PMR to application responsiveness, we separately evaluated the memory reclaim throughput and other performance indicators of the system during the following experiments, including the number of LMKD, direct reclaim, and page faults. On the one hand, memory reclaim throughput can intuitively see the benefits of memory reclaim in ensuring system memory availability. On the other hand, the number of LMKD indicates the efficiency with which the system memory pressure is alleviated. In contrast, direct reclaim and page fault indicate the impact of memory reclaim on the application [32]. Note that in this subsection, since Acclaim also focuses on kernel-level memory reclaim optimization, we choose it as the state-of-the-art work for comparison, rather than Fleet, which focuses on the joint design of ART GC and memory swapping.

5.3.1 Memory Reclaim Throughput

Figure 12 shows the memory reclaim throughput under different mechanisms. First, the memory reclaim throughput of OriginalMR and Acclaim shows the same trend. That is, it changes in a very low range. The reason why OriginalMR suffers this performance loss has been revealed previously. Acclaim is committed to reducing the number of direct reclaim and page faults and does not optimize the memory reclaim path, so it also fails to improve memory reclaim throughput. Secondly, compared with previous work, PMR benefits from a parallelized memory reclaim path, which not only avoids passive waiting for page writeback, but also accelerates the traditional page writeback. The results show that the peak memory reclaim throughput of PMR is increased by 82.8% and 75.5% respectively compared with OriginalMR and Acclaim. The huge improvement in memory reclaim throughput shows that PMR can perform memory reclaim at a faster speed, which explains the performance improvement of PMR in terms of application response time. Also, since Fleet focuses more on the accuracy of memory reclaim and does not essentially affect the reclaim throughput, it was not chosen as the comparison object in the following experiments.

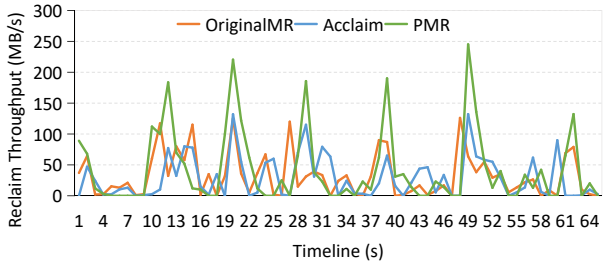


Figure 12: Memory reclaim throughput among different schemes.

5.3.2 Reduction of LMKD/Direct Reclaim/Page Fault

LMKD can cause applications to suffer long restart delays. Direct reclaim, which is a synchronous memory reclaim, has a direct impact on application allocation page requests. Page faults can cause long page reads. To understand the improvement of application response due to PMR, we further analyze the above three performance indicators.

Figure 13 shows the number of LMKD, direct reclaim and page fault under different memory reclaim schemes. First, OriginalMR suffers the most from LMKD, direct reclaim, and page faults. Acclaim benefits from foreground-aware memory reclaim, which can effectively reduce the number of direct reclaim and page faults. However, it essentially does not effectively alleviate memory pressure, and LMKD still occurs frequently. Secondly, unlike Acclaim, PMR can effectively alleviate memory pressure with the help of fast memory reclaim and avoid the frequent occurrence of LMK. Compared

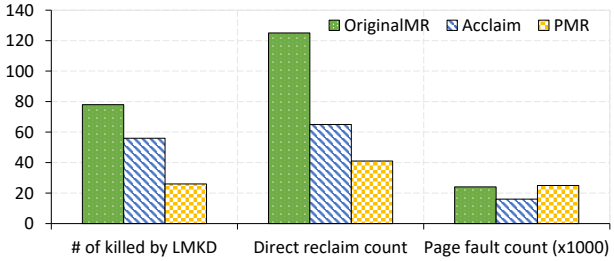


Figure 13: The number of killed applications, direct reclaim and page fault among different schemes.

with OriginalMR and Acclaim, PMR reduces LMKD by 82% and 54% respectively. Moreover, when memory swapping can effectively alleviate memory pressure, direct reclaim will naturally decrease, PMR reduces it by 45% compared to Acclaim. Since PMR does not change the victim selection page, the page fault rate remains the same as OriginalMR.

5.4 Storage Utilization Evaluation

In this subsection, we calculate the memory reclaim throughput on different mobile devices under the same application running condition to demonstrate the efficiency of PMR by taking advantage of evolving storage devices to improve the efficiency of memory reclaim. Specifically, Pixel 5 and Pixel 6 pro were selected as experimental platforms, with the former equipped with a 128 GB UFS 2.1 storage device and the latter equipped with a 256 GB UFS 3.1 storage device. Similar to the previous experiment, 26 applications were running in the background, and six applications were switched back and forth in the foreground, including YouTube, Facebook, Twitter, TikTok, Gmap and Camera. We collect the memory reclaim throughput during the switching of these six applications.

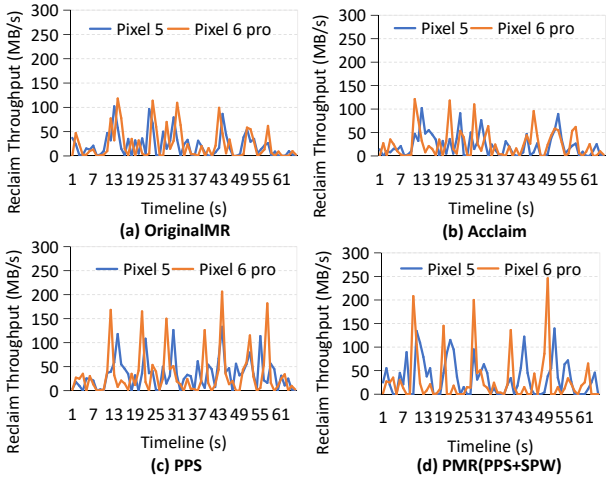


Figure 14: Memory reclaim throughput for different schemes under different mobile devices.

Figure 14 shows the memory reclaim throughput of different memory reclaim mechanisms on different mobile devices. First, OriginalMR and Acclaim pay little attention to the performance improvements of storage devices. This is because Acclaim only focuses on optimizing the victim page selection scheme and dynamically adjusting the size of *kswapd* to reduce direct reclaim and page faults, lacking any design for storage devices. Secondly, compared with the previous mechanisms, the newly proposed PPS significantly improves the memory reclaim throughput, especially the performance of storage devices. For example, for Pixel 6 pro, the maximum throughput of PPS reaches 205 MB/s, which is an 45% increase compared to Pixel 5. Finally, the integration of PPS and SPW can not only provide sufficient victim pages for page writeback in advance, but also enable storage-friendly page writeback to maximize the storage device's performance. The results show that the average throughput on the Pixel 6 pro is 65% of that on the Pixel 5.

5.5 Sensitivity Study

To further understand PMR, several sensitive studies are performed, including the ideal number of pages in the victim page list (*nr_ideal_victim_page*: δ) and the size of the application-aware page unmap.

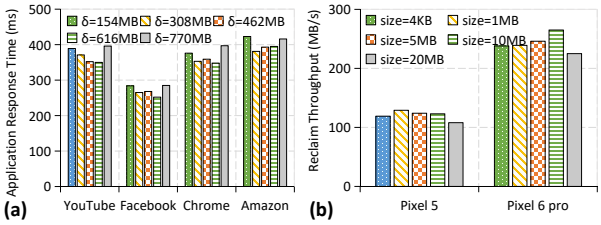


Figure 15: (a) Application responsiveness by varying δ (the ideal number of pages in the victim page list); (b) Memory reclaim throughput by varying the size of the application-aware page unmap.

The ideal pages in the victim page list: δ aims to control the capacity of the victim page list, which directly affects the performance of memory reclaim. Figure 15 (a) shows the effect of changing δ from 154 MB (equivalent to `nr_to_reclaim`) to 770 MB on the application response time. First, as δ increases, the application response time decreases and then increases again. δ is not as large as it should be. This is because a larger δ means that more pages are placed in the victim page list, and pages may face repeated transfers between the victim page list and the system LRU page list, which increases page access overhead. Second, the change in δ has a limited impact on the application's responsiveness because the victim page list is inherited from the system LRU page list, which does not introduce new data structures or memory overhead.

The size of the application-aware page unmap: The size of the application-aware page unmap affects both the performance of the unmap and the efficiency of write I/O. Figure 15 (b) shows the effect of different sizes of the application-aware page unmap on the memory reclaim throughput. First, the optimal value for this size varies from one mobile device to another, for example, the Pixel 5 is 1 MB while the Pixel 6 pro is 10 MB, depending on the underlying storage device.

5.6 Overhead Analysis

In this section, we evaluate PMR's overheads: (1) the performance overhead of PPS and SPW in the current design and (2) the flash write overhead induced by proactive disk writeback.

Performance Overhead. The performance overhead of PMR is broken down into three parts. First, PPS enables an independent *kshrinkd* thread to be responsible for page reduction. However, because shrink can only inherit some of the functions of the original *kswapd*, while the *kshrinkd* brings CPU overhead, the CPU overhead of *kswapd* will also be reduced. Second, PPS maintains a separate victim page list. Although these pages come from the system's original inactive page list, they will not cause new memory overhead. Also, to maintain a sufficient number of victim pages, some additional lock protection needs to be added, which also brings CPU overhead concerns. Finally, SPW adjusts the priority of application-aware page unmap to avoid interference by processes, which may also bring potential performance overhead. As shown in Table 3, PPS increases CPU overhead by 2.0% compared to OriginalMR, while PMR increases by 5.3%.

Table 3: Performance overhead.

	CPU Overhead		
	kswapd	***kshrinkd***	**Total**
OriginalMR	24.31	0	24.31
PPS	13.94	10.97	24.91
PMR (PPS+SPW)	14.51	11.1	25.61

Flash Write Overhead. PMR is more effective in evicting pages, which allows it to reduce application kills. This results in more aggressive flash writes. To evaluate the possibility that excessive flash writes may wear out the flash device, we ran the system with and without PMR for half an hour and collected the write volumes. The experimental results show that PMR increases flash writes by 12.1% compared to OriginMR. In fact, as the storage capacity of mobile devices increases, the lifespan anxiety of storage devices has been greatly alleviated. The impact of PMR on flash device lifespan is negligible and worthwhile compared to the benefits.

6 Related Work

Memory Reclaim Optimization. Many previous works enhance the memory reclaim schemes [7, 20, 41, 44, 49, 52]. First, Fastswap [7] and Hermit [41] optimize how and where

page reclaim is executed (e.g., offloading to a dedicated CPU or spawning threads), while our work improves what is executed, i.e., we redesign the reclaim path itself to make it inherently more efficient. This deeper improvement makes existing mechanisms like Fastswap and Hermit more effective. With our faster page reclaim, Fastswap no longer needs a dedicated CPU, and Hermit requires fewer threads for the same performance. Second, recent studies [27, 58] perform ahead-of-swap to optimize memory swapping. For example, SmartSwap [58] predicts the appropriate swap time based on the design model, and Marin [17] swaps out objects in advance by analyzing the objects working set. Unlike them, PMR does not modify the timing of memory reclaim, but only prepares the victim pages in advance.

Also, another method is to update the LRU-based victim page selection criteria [32, 34, 56] Since PMR is dedicated to accelerating the common memory reclaim path, it is compatible with any effective victim page selection algorithm. Multiplekswapd [36] enables multiple memory reclaim threads to perform proactive memory reclaim. However, blindly using multiple threads will induce mutual interference and will not achieve the expected reclaim effect, especially on mobile devices with limited resources. In contrast, PMR can achieve the purpose of parallel reclaim by enabling an additional page-shrinking thread. Finally, researchers also focus on coordination between ART garbage collection and kernel-level page swapping [19, 27] to boost memory reclaim performance. Unlike these studies, our work targets system-level kernel reclaim logic directly. Combining our kernel-level improvements with Fleet [19] yields even more significant gains, highlighting their orthogonality and the broader applicability of our solution.

Storage-friendly I/O Management. Although the performance of flash storage devices is improving rapidly, current system software is still deficient in leveraging the performance of storage devices. Many researches have been devoted to the design of storage-friendly I/O management [5, 23, 30, 34, 40, 42, 57]. Paralfetch [42] performed a pre-scheduling of launch-related disk read requests for fast I/O reads, and overlaps app execution with disk prefetching for hiding disk access time from the app execution. Its basic idea is close to our work, PMR also pre-schedules page shrinking and page writeback in advance and achieves fast page writeback. SWAM [34] dynamically modifies the size of page unmap to avoid interference with high-priority operations for fast swap-out. Differently, PMR performs bulk page unmap to server storage-friendly page out, which organizes pages by application and determines the size of the unmap based on the performance characteristics of the storage device.

Application Response Optimization. There are a number of strategies to optimize application responsiveness on mobile device [18, 22, 26, 28, 33, 39, 46]. ASAP [46] performed pre-paging by combining application switch footprint estimators and minimizing resource waste for CPU cycles and disk bandwidth during an application switch for fast application switch on mobile devices. Fasttrack [18] proposed a foreground app-aware I/O management scheme to accelerate foreground I/O requests by preempting background I/O requests in the entire I/O stacks, including the storage device, and preventing foreground app's data from being flushed from the page cache. CacheSifter [33] classified cache files into three categories online and greatly reduced the number of writebacks on Flash by dropping cache files that most likely will not be reused to boost application performance. Unlike them, PMR performs parallel memory reclaim path to boost application response.

7 Conclusion

In this paper, we propose PMR, a parallel memory reclaim scheme for fast application response on mobile devices. The basic idea of PMR is to parallelize the memory reclaim path to leverage ever-evolving storage devices and enhance memory reclaim. Specifically, PMR firstly decouples page shrinking and page writeback and prepares sufficient victim pages in advance to quickly serve subsequent page writeback. And then, PMR performs application-aware page unmap and page out in batches to achieve a storage-friendly page writeback. Experimental results demonstrate that PMR effectively improves memory reclaim throughput and boost application response.

References

[1] Oppo introduces new memory expansion technology for its reno5 series, a94 and a74 series smartphones, 2021. https://www.oppo.com/sg/newsroom/press/oppo-introduces-new-memory-expansion-technology/.

[2] fio - flexible i/o tester rev. 3.36, 2025. https://fio.readthedocs.io/en/latest/fio_doc.html.

[3] ktime accessors, 2025. https://docs.kernel.org/core-api/timekeeping.html.

[4] Universal flash storage (ufs), 2025. https://en.wikipedia.org/wiki/Universal_Flash_Storage.

[5] Mohammadamin Ajdari, Pouria Peykani Sani, Amirhossein Moradi, Masoud Khanalizadeh Imani, Amir Hossein Bazkhanei, and Hossein Asadi. Re-architecting i/o caches for emerging fast storage devices. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'23)*, pages 542–555, 2023.

[6] Keivan Alizadeh, Iman Mirzadeh, Dmitry Belenko, Karen Khatamifard, Minsik Cho, Carlo C Del Mundo, Mohammad Rastegari, and Mehrdad Farajtabar. Llm in a flash: Efficient large language model inference with limited memory. *arXiv preprint arXiv:2312.11514*, 2023.

[7] Emmanuel Amaro, Christopher Branner-Augmon, Zhihong Luo, Amy Ousterhout, Marcos K Aguilera, Aurojit Panda, Sylvia Ratnasamy, and Scott Shenker. Can far memory improve job throughput? In *Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys'20)*, pages 1–16, 2020.

[8] Muhammad Arbaz. Ufs 4.0 vs ufs 3.1 vs ufs 3.0 | speed comparison, 2023. https://www.thephonetalks.com/ufs-4-0-vs-ufs-3-1-vs-ufs-3-0comparison/.

[9] Android Developers. Android debug bridge(adb), 2025. https://developer.android.com/studio/command-line/adb.

[10] Android Developers. Keeping your app responsive, 2025. https://developer.android.com/training/articles/perf-anr.

[11] Android Developers. Logcat command-line tool, 2025. https://developer.android.com/tools/logcat.

[12] Android Developers. Low memory management, 2025. https://developer.android.com/topic/performance/memory-management.

[13] Android Developers Docs. Low memory killer daemon, 2025. https://source.android.com/docs/core/perf/lmkd.

[14] Linux Foundation. Vmscan, 2025. https://github.com/torvalds/linux/blob/master/mm/vmscan.c.

[15] GSMArena. Apple iphone 16 pro max, 2025. https://www.gsmarena.com/apple_iphone_16_pro_max-13123.php.

[16] GSMArena. Samsung galaxy s25 ultra, 2025. https://www.gsmarena.com/samsung_galaxy_s25_ultra-13322.php.

[17] Weichao Guo, Kang Chen, Huan Feng, Yongwei Wu, Rui Zhang, and Weimin Zheng. *mars*: Mobile application relaunching speed-up through flash-aware page swapping. *IEEE Transactions on Computers (TC)*, pages 916–128, 2015.

[18] Sangwook Shane Hahn, Sungjin Lee, Inhyuk Yee, Donguk Ryu, and Jihong Kim. FastTrack: Foreground App-Aware I/O management for improving user experience of android smartphones. In *USENIX Annual Technical Conference (ATC'18)*, pages 15–28, 2018.

[19] Jiacheng Huang, Yunmo Zhang, Junqiao Qiu, Yu Liang, Rachata Ausavarungnirun, Qingan Li, and Chun Jason Xue. More apps, faster hot-launch on mobile devices via fore/background-aware gc-swap co-design. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'24)*, page 654–670, 2024.

[20] Jian Huang, Moinuddin K. Qureshi, and Karsten Schwan. An evolutionary study of linux memory management for fun and profit. In *USENIX Annual Technical Conference (ATC'16)*, page 465–478, 2016.

[21] Huawei. Huawei's memory expansion technology: 8gb ram works as 10gb and 12gb ram as 14gb: Huawei community, 2020. https://consumer.huawei.com/ae-en/community/details/NEWS-Huawei-s-Memory-Expansion-Technology-8GB-RAM-works-as-10GB-and-12GB-RAM-as-14GB/topicId_121496/.

[22] Daeho Jeong, Youngjae Lee, and Jin-Soo Kim. Boosting Quasi-Asynchronous I/O for better responsiveness in mobile devices. In *USENIX Conference on File and Storage Technologies (FAST'15)*, pages 191–202, 2015.

[23] Yongsoo Joo, Junhee Ryu, Sangsoo Park, and Kang G. Shin. FAST: Quick application launch on Solid-State drives. In *USENIX Conference on File and Storage Technologies (FAST'11)*, pages 101–114, 2011.

[24] Linux Kernel. void blk_start_plug(struct blk_plug *plug), 2025. https://docs.kernel.org/core-api/kernel-api.html.

[25] Sang-Hoon Kim, Jinkyu Jeong, and Jin-Soo Kim. Application-aware swapping for mobile systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 16(5s):1–19, 2017.

[26] Sangwook Kim, Hwanju Kim, Joonwon Lee, and Jinkyu Jeong. Enlightening the I/O path: A holistic approach for application performance. In *USENIX Conference on File and Storage Technologies (FAST'17)*, pages 345–358, 2017.

[27] Niel Lebeck, Arvind Krishnamurthy, Henry M. Levy, and Irene Zhang. End the senseless killing: Improving memory management for mobile operating systems. In *USENIX Annual Technical Conference (ATC'20)*, pages 873–887, 2020.

[28] Changlong Li, Yu Liang, Rachata Ausavarungnirun, Zongwei Zhu, Liang Shi, and Chuan Jason Xue. Ice: Collaborating memory and process management for user experience on resource-limited mobile devices. In *Proceedings of the Eighteenth European Conference on Computer Systems (EuroSys'23)*, pages 79–93, 2023.

[29] Changlong Li, Liang Shi, Yu Liang, and Chun Jason Xue. Seal: User experience-aware two-level swap for mobile devices. *Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 39(11):4102–4114, 2020.

[30] Wentong Li, Liang Shi, Hang Li, Changlong Li, and Edwin Hsing-Mean Sha. Iosr: Improving i/o efficiency for memory swapping on mobile devices via scheduling and reshaping. *ACM Transactions on Embedded Computing Systems (TECS)*, 22(5s):1–23, 2023.

[31] Wentong Li, Dingcui Yu, Yunpeng Song, Longfei Luo, and Liang Shi. Elasticzram: Revisiting zram for swapping on mobile devices. In *Proceedings of the 61st ACM/IEEE Design Automation Conference (DAC'24)*, pages 1–6, 2024.

[32] Yu Liang, Jinheng Li, Rachata Ausavarungnirun, Riwei Pan, Liang Shi, Tei-Wei Kuo, and Chun Jason Xue. Acclaim: Adaptive memory reclaim to improve user experience in android systems. In *USENIX Annual Technical Conference (ATC'20)*, pages 897–910, 2020.

[33] Yu Liang, Riwei Pan, Tianyu Ren, Yufei Cui, Rachata Ausavarungnirun, Xianzhang Chen, Changlong Li, Tei-Wei Kuo, and Chun Jason Xue. {CacheSifter}: Sifting cache files for boosted mobile performance and lifetime. In *USENIX Conference on File and Storage Technologies (FAST'22)*, pages 445–459, 2022.

[34] Geunsik Lim, Donghyun Kang, Myungjoo Ham, and Young Ik Eom. Swam: Revisiting swap and oomk for improving application responsiveness on mobile devices. In *Proceedings of the 29th Annual International Conference on Mobile Computing and Networking (MobiCom'23)*, pages 1–15, 2023.

[35] linuxfan says Reinstate Monica. What is memory reclaim in linux, 2025. https://stackoverflow.com/questions/42358745/what-is-memory-reclaim-in-linux.

[36] Buddy Lumpkin. mm: Support multiple kswapd threads per node, 2018. https://lore.kernel.org/lkml/20180417090335.GZ17484@dhcp22.suse.cz/t/.

[37] Yu Mao, Weilan Wang, Hongchao Du, Nan Guan, and Chun Jason Xue. On the compressibility of quantized large language models. *arXiv preprint arXiv:2403.01384*, 2024.

[38] Ivan Mehta. How the ufs 4.0 storage standard will improve your phone's performance, 2022. https://thenextweb.com/news/ufs-4-0-samsung-phone-stroage-analysis.

[39] David T. Nguyen, Gang Zhou, Guoliang Xing, Xin Qi, Zijiang Hao, Ge Peng, and Qing Yang. Reducing smartphone application delay through read/write isolation. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys'15)*, pages 287–300, 2015.

[40] Anastasios Papagiannis, Giorgos Xanthakis, Giorgos Saloustros, Manolis Marazakis, and Angelos Bilas. Optimizing memory-mapped {I/O} for fast storage devices. In *USENIX Annual Technical Conference (ATC'20)*, pages 813–827, 2020.

[41] Yifan Qiao, Chenxi Wang, Zhenyuan Ruan, Adam Belay, Qingda Lu, Yiying Zhang, Miryung Kim, and Guoqing Harry Xu. Hermit: Low-Latency, High-Throughput, and transparent remote memory via Feedback-Directed asynchrony. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 181–198, 2023.

[42] Junhee Ryu, Dongeun Lee, Kang G Shin, and Kyungtae Kang. Fast application launch on personal {Computing/Communication} devices. In *USENIX Conference on File and Storage Technologies (FAST'23)*, pages 425–440, 2023.

[43] SamMobile. Ram plus: Samsung's extra ram feature is arriving on more phones, both mid-range and flagship, 2025. https://www.sammobile.com/news/samsung-galaxy-a52s-5g-virtual-plus-feature/.

[44] Kunal Sareen, Stephen M. Blackburn, Sara S. Hamouda, and Lokesh Gidra. Memory management on mobile devices. In *Proceedings of the 2024 ACM SIGPLAN International Symposium on Memory Management (ISMM'24)*, page 15–29, 2024.

[45] Arjun Sha. What is ufs 4.0 storage and how fast is it compared to ufs 3.1?, 2022. https://beebom.com/what-is-ufs-4-0/.

[46] Sam Son, Seung Yul Lee, Yunho Jin, Jonghyun Bae, Jinkyu Jeong, Tae Jun Ham, Jae W. Lee, and Hongil Yoon. ASAP: fast mobile application switch via adaptive prepaging. In *USENIX Annual Technical Conference (ATC'21)*, pages 365–380, 2021.

[47] Gokul Soundararajan, Vijayan Prabhakaran, Mahesh Balakrishnan, and Ted Wobber. Extending ssd lifetimes with disk-based write caches. In *USENIX Conference on File and Storage Technologies (FAST'10)*, pages 101–114, 2010.

[48] Pyropus technology. Memory tester tool memtester, 2017. https://pyropus.ca/software/memtester/.

[49] Chenxi Wang, Yifan Qiao, Haoran Ma, Shi Liu, Wenguang Chen, Ravi Netravali, Miryung Kim, and Guoqing Harry Xu. Canvas: Isolated and adaptive swapping for Multi-Applications on remote memory. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI'23)*, pages 161–179, 2023.

[50] Yong-Xuan Wang, Chung-Hsuan Tsai, and Li-Pin Chang. Killing processes or killing flash? escaping from the dilemma using lightweight, compression-aware swap for mobile devices. *ACM Transactions on Embedded Computing Systems (TECS)*, 20(5s):1–24, 2021.

[51] Zhuohao Wang, Lei Liu, and Limin Xiao. iswap: A new memory page swap mechanism for reducing ineffective i/o operations in cloud environments. *ACM Transactions on Architecture and Code Optimization (TACO)*, 21(3):1–24, 2023.

[52] Johannes Weiner, Niket Agarwal, Dan Schatzberg, Leon Yang, Hao Wang, Blaise Sanouillet, Bikash Sharma, Tejun Heo, Mayank Jain, Chunqiang Tang, et al. Tmo: Transparent memory offloading in datacenters. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'22)*, pages 609–621, 2022.

[53] Xiaocong. uiautomator, 2025. https://github.com/xiaocong/uiautomator.

[54] Xiaomi. All Xiaomi that will have the RAM expansion function, 2021. https://www.xiaomist.com/2021/07/these-are-all-xiaomi-that-can-already.html.

[55] Wangsong Yin, Mengwei Xu, Yuanchun Li, and Xuanzhe Liu. Llm as a system service on mobile devices. *arXiv preprint arXiv:2403.11805*, 2024.

[56] Yu Zhao. Multigenerational lru framework, 2022. https://lwn.net/Articles/880393/.

[57] Kan Zhong, Wenlin Cui, Xin Chen, Qiao Li, Zhe Yang, Youyou Lu, Xiaodan Yan, Siwei Luo, Qizhao Yuan, and Keji Huang. Revisiting swapping in user-space with lightweight threading. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems(TCAD)*, 42(11):4205–4218, 2023.

[58] Xiao Zhu, Duo Liu, Kan Zhong, Jinting Ren, and Tao Li. Smartswap: High-performance and user experience friendly swapping in mobile systems. In *Proceedings of the 54th Annual Design Automation Conference (DAC'17)*, pages 1–6, 2017.

SAVE: Software-Implemented Fault Tolerance for Model Inference against GPU Memory Bit Flips

Wenxin Zheng, Bin Xu, Jinyu Gu[✉]*, Haibo Chen*
Institute of Parallel and Distributed Systems, School of Computer Science, Shanghai Jiao Tong University

Abstract

Machine learning models are used in safety-critical edge applications such as autonomous driving, industrial robots, and satellites. However, GPU memory bit flips can significantly reduce the model accuracy. Existing mitigations either compromise accuracy or introduce substantial overhead.

Our insight is that *not all hardware bits are created equal* and *bit flips vary in their impact on model inference*. Specifically, for the GPU memory, modern AI accelerators provide bit-flip-free but small reliable memory. For the model inference, due to nonlinear activation functions in the model, some bits are naturally robust against flips, while other vulnerable bits can silently corrupt results. Thus, we prioritize the allocation of vulnerable bits' computations in the reliable memory to enhance the robustness of the model inference.

We propose SAVE, a software-implemented fault tolerance system that protects model inference without modifying the model and with minimal performance impact. SAVE operates in four stages: *S*election to identify vulnerable bits based on the intrinsic characteristics of model inference, *A*llocation to prioritize computations related to more vulnerable bits in reliable memory, *V*erification to efficiently detect errors through asynchronous CPU checks, and *E*dit to recover from detected faults. Evaluation across computer vision, robotics, and decision-making models shows that SAVE maintains model accuracy even under 4K bit flips while incurring less than 9% performance overhead.

1 Introduction

Machine learning models already play an important role in safety-critical edge scenarios including autonomous driving [2, 54, 71], financial systems [6, 43], combat operations [10, 82], and military satellites [32]. The models on local edge devices handle a large amount of data, e.g., 20TB per day for one satellite [89] and 30TB per day for one autonomous driving car [1]. These on-device edge GPUs are more likely to have memory bits unintentionally flip due to unstable voltage in Unmanned Aerial Vehicles (UAVs), changing temperatures in cars, or radiation in space. Protecting inference safety against memory bit flips on these edge devices is crucial while challenging. For example, one satellite may experience approximately 16 million bit flips daily [89], and traditional Error Correction Code (ECC) [35, 68] proves insufficient, as it provides only single bit errors correction [68] while even a single escaped bit flip can cause complete accuracy loss and false decisions [29].

Current approaches to protect model inference against bit flips on edge accelerators fall into two categories based on whether they modify the model structure. The first category modifies the model structure to enhance model robustness, including specialized activation functions [89], compression [37, 76], and quantization [40]. However, these methods often compromise accuracy and lack generality across different models. The second category preserves model structure while implementing protection through redundancy, such as Triple Modular Redundancy (TMR) [5, 13, 57, 60, 79] and ECC [35, 68]. Although these approaches preserve accuracy and generality, they either incur substantial overhead (2× for TMR) or provide limited error correction capabilities (ECC).

In this paper, we focus on the second category and propose an efficient software mitigation named SAVE against bit flips in GPU memory, which improves the reliability of model inference results without retraining the model or sacrificing accuracy.

Our first insight is that **not all hardware bits are created equal**. Modern edge-deployed AI accelerators like NVIDIA Orin provide bit-flip-free safety islands [67], while hardware manufacturers [25, 52] implement SRAM soft error detection in e.MMC [26] for L1 cache protection. SAVE leverages these reliable memory regions for reliable model inference.

However, these reliable memory regions are limited (only around 5-6MB in Orin [67]). Relying solely on them requires frequent CPU-GPU memory swapping, which inevitably increases performance overhead. Our evaluation shows it introduces more than 1,000× performance overhead for both computer vision and robotics decision-making models.

Our second insight is that **not all bit flips in software are fatal or silent**, which offers the chance to hybridly use

reliable memory and normal memory for model inference computation. During computation, the bits can be classified into three categories: *robust bits* that don't affect inference results, *ranging bits* that can be verified through simple range checks (e.g., the Sigmoid function's sign bit must be zero as outputs range from 0 to 1), and *vulnerable bits* that silently corrupt the model. SAVE strategically allocates computations involving vulnerable bits to reliable memory while computing robust bits in normal memory, efficiently minimizing memory swapping while maintaining result correctness.

Specifically, SAVE contains four stages to enhance the reliability of model inference against GPU memory bit flips, namely, *S*election stage, *A*llocation stage, *V*erification stage, and *E*dit stage.

Selection stage: robustness analysis and output range analysis. Machine learning models are programs that are easier to analyze data and control flow compared to general software. Thus, SAVE analyzes the model with data flow, control flow, and mathematical properties to identify robust bits in each value. For example, activation functions like ReLU [64] and GELU [38] yield outputs close to zero for negative inputs, meaning changes to the significand of its input will have little impact. Additionally, SAVE assesses each layer's output range to find ranging bits. For example, for image classification models, the image pixel value gives the range of the input value. The known input range, together with the model parameters, gives the output range of each layer. Some of the exponent bits or sign bits can be verified by these ranges without recomputation. SAVE sets all remaining bits as vulnerable bits. The findings are stored in a bit attribution cache to accelerate further verification.

Allocation stage: robustness-aware GPU memory management. Based on offline robustness analysis, SAVE prioritizes allocating reliable memory to computations with more vulnerable bits. To optimize the use of limited reliable memory, SAVE employs in-place computation techniques, including overwriting input tensors with output results during matrix multiplication. Model parameters are intentionally placed in normal memory because they have copies stored in CPU memory, allowing for bit flip detection through CPU-GPU consistency checks during verification.

Verification stage: lightweight runtime verification. SAVE implements distinct verification mechanisms to accelerate the runtime verification. Values in reliable memory are considered correct and do not require verification. For values in normal memory, SAVE verifies the correctness differently according to their roles. For model parameters, it uses DMA to copy them back to the CPU during runtime for double-checking without impacting model inference. For other values, SAVE verifies the correctness of ranging bits using the valid output range. To reduce the performance overhead, it verifies vulnerability bits with integer computation using the integer computation resources to avoid heavy floating-point recomputation.

Table 1: Model degradation due to bit flips. (1): The models with softmax. (2): ResNet, VGG. (3): 8-bit quantization models. (4): ASR: Attack Success Rate.

	Model	Flipped Bits	Accuracy Degradation
NaN [29]	EfficientNet	3	82.00% → 0%
FIASM [46]	Softmax(1)	1	97.4% → 62%
TBD [40]	CV(2)	1	Up to 99% drop
BFA [73]	ResNet18(3)	13	69.8% → 0.1%
DHB [37]	ResNet20(3)	28	81.39% drop
TBT [74]	ResNet18(3)	84	92% ASR(4)
TA-LBF [9]	ResNet18(3)	1507	100% ASR(4)
T-BFA [75]	ResNet18(3)	27	100% ASR(4)
Our Experiments	ResNet	1	Up to 99% Drop
	ViT		
	CogACT		

Edit stage: eliminating bit flips. When an error is detected, SAVE simply restarts the inference computation from the faulting model layer.

To demonstrate the efficiency and effectiveness of bit flip mitigation, we test SAVE across various computer vision, robotics, and decision-making models. The results show that SAVE incurs less than 9% performance overhead for end-to-end latency in model inference and maintains the model's accuracy even in the case of 4K bit. To compare existing approaches, we propose a new metric, *AccurateLatency*, which penalizes high latency and low accuracy in inference. SAVE is 90% lower than state-of-the-art methods under *AccurateLatency*.

In summary, our contributions are as follows:

- An idea of arranging computation related to vulnerable bits in reliable memory to improve the reliability of model inference.
- A prototype system, SAVE, to mitigate GPU memory bit flips by using robustness-aware GPU memory management and lightweight runtime verification and recomputation.
- An intensive evaluation that demonstrates the efficiency and effectiveness of SAVE.

2 Background

2.1 GPU Bit Flips on Edge Inference

Machine learning models greatly improve decision and recognition accuracy. So, many on-device edge scenarios like autonomous cars [2, 54, 71] and satellite image recognition [32] use these models for local inference. To support these applications, GPUs are widely used for model inference due to their high parallelism and efficiency. These GPUs are often deployed in edge devices, such as drones and satellites, where they can process data locally and make real-time decisions. However, the GPU memory reliability in the edge device can be easily compromised by physical factors such as unstable

cell voltage, radiation, and temperature fluctuations. Such decreased reliability will result in unintended bit changes in GPU memory, commonly referred to as bit flips. Bit flips are a common occurrence across various edge-deployed GPU devices [90]. For instance, the memory flip rate in UAVs operating under low voltage conditions can reach 3.5% [86]. In space devices, radiation exposure can cause flip rates as high as 10%, with multiple cells potentially flipping simultaneously [89]. On some lower-end devices, an average of 3,900 bit flips may occur per day [80]. One satellite can experience 16 million bit flips per day due to cosmic rays [89].

These bit flips can significantly degrade model accuracy. Research indicates that even a single bit flip can reduce model accuracy by more than 60%, as demonstrated in Table 1. Such accuracy degradation can have severe consequences for downstream applications. In financial systems, erroneous results can silently affect stock trading volumes or target prices. In aerospace applications, corrupted decisions or navigation calculations can result in mission failures.

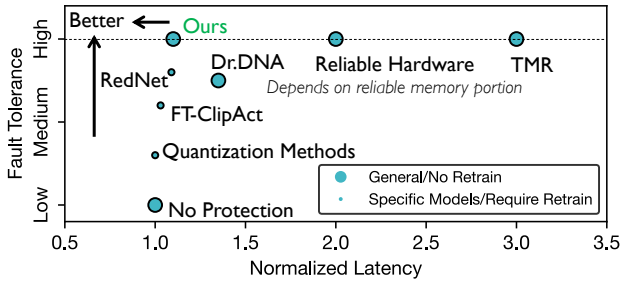


Figure 1: Comparison of different bit flip mitigation methods.

2.2 Existing Approaches

Existing approaches to safeguard inference safety on edge devices against bit flips can be categorized into two types based on whether they modify the AI model. A comparison of these approaches is shown in Figure 1.

The first category involves modifying the model structure or its outputs to enhance model robustness. Some methods [33, 47, 53, 89] employ hashing-based techniques to generate weight group signatures and verify parameters. Other approaches propose using quantization [37, 40, 76] and specialized activation functions, such as FT-ClipAct [39], RedNet [89], and Ranger [16], to mitigate the impact of bit flips. Additionally, Dr.DNA [61] leverages the distribution of activation results to detect and correct bit flips. However, these methods often require retraining the models or result in reduced accuracy.

The second category preserves the original model structure while implementing protection mechanisms through software or hardware redundancy. For instance, ECC can detect and correct limited bit errors. However, ECC is typically deployed only in high-end devices [4, 51] and is unavailable on many off-the-shelf GPUs. Moreover, it incurs additional costs, including a 10% to 20% increase in memory usage and a 2% to 3% reduction in performance [20, 24]. Furthermore, in extreme scenarios such as those involving cosmic ray impacts [89], a large number of bits may flip simultaneously, exceeding the error correction capacity of ECC and causing it to fail. Triple Modular Redundancy (TMR) [60, 79] is a general software solution to improve computational reliability. However, it requires threefold computation, rendering it prohibitively expensive for many scenarios [89].

2.3 Reliable Memory Exists on GPUs

We observe that "not all bits are created equal" on GPUs. Specifically, certain portions of GPU memory exhibit higher reliability, presenting opportunities for reliable computing. This observation is supported by the following two facts.

First, NVIDIA Orin, a widely-used edge AI accelerator, provides safety islands [67], which account for only 0.009% of the total memory and are guaranteed to be free of bit flips. Additionally, hardware manufacturers [25] offer soft error detection features for SRAM in e.MMC [26], enabling hardware-based protection for L1 cache. Second, similar to bit flips observed in DRAM on CPUs [8, 49, 50], a significant portion of GPU memory bits may be prone to flipping, while others remain stable and never flip (effectively functioning as safety islands). The properties of these bits are determined during manufacturing and are influenced by factors such as the chip fabrication process, memory cell voltage stability, and resistance to environmental stress. Defective bits prone to flipping can be identified through experiments using rowhammer-like tools [84, 96, 97] or through environmental testing [12, 19, 34, 87, 89]. Bits that are not prone to flipping can be treated as reliable memory and leveraged during inference.

2.4 Fault Model and Goal

Our work attempts to leverage the reliable memory on GPUs to improve the robustness of model inference against GPU memory bit flips under the following assumptions: 1) initial model parameters and inputs on the CPU side do not experience bit flips, as they are protected by orthogonal reliable methods on CPU-side memory; 2) registers, cache, and a small portion of memory on the GPU are free from bit flips; 3) the GPU reliable memory can be detected by environmental testing like Rowhammer-like tools [11, 14, 22, 51]; 4) the remaining GPU memory may experience random bit flips, including both single-bit and multi-bit flips; 5) the GPU kernel binary could fit within the reliable memory because of its small size after pruning [69].

It is important to note that the goal of SAVE is to *enhance* the robustness of model inference under GPU bit flip conditions, rather than strictly guarantee inference correctness under all possible flip scenarios.

3 Basic Idea and Insights

3.1 Basic Idea

A strawman solution is making all the computation conducted within the reliable GPU memory. However, the size of reliable memory is typically too small (e.g., less than 10MB) to accommodate GPU kernel computation with large matrices. Thus, we need to partition the kernel computation so that the input, model parameters, and output can fit into the small reliable memory. This involves continuously transferring data between the CPU and GPU reliable memory. According to our tests on both one visual model [23] and one decision model [15], this approach would result in a more than 1000× increase in model inference latency, which is unacceptable.

Our basic idea is also utilizing only the reliable GPU memory for improving the reliability of model inference. The key difference from the strawman solution lies in that our system (SAVE) hybridly uses reliable memory and unreliable memory on the GPU.

We first find that it is feasible to pre-load all the model parameters into unreliable memory, which can save more reliable memory for accommodating the kernel input and output. This is because 1) there exists an intact copy of model parameters on the CPU side; 2) the parameters in the unreliable memory can be copied back to the CPU side for verifying consistency, by using the free CPU-GPU PCIe path with the dedicated memory copy engine during kernel execution without hurting performance.

When executing the first layer's kernel computation of the model, SAVE copies the input matrix to reliable memory and launches the kernel for computation. After the first layer completes, it pulls the kernel's parameters back to the CPU and compares them with CPU-side parameters to verify that no bit flips occurred in the unreliable memory during kernel computation. This procedure is overlapped with the kernel execution of the next layer. While there exists an extreme edge case where a bit might flip before computation and revert before CPU verification, this time window is negligibly small, and we do not consider such extreme cases.

The output of the first layer's kernel computation is also placed in reliable memory, in-place overwriting the input to serve as input for the next layer's computation with another copied version to serve as the last reliable output. This process continues layer by layer until completion. During this process, if the total memory required for input/output in any layer's kernel computation exceeds the size of reliable memory, SAVE still partitions the kernel's input to ensure each computation fits within reliable memory capacity, and swaps intermediate results to CPU memory for temporary storage.

Besides model parameters, we also find that some parts of the kernel input can also be placed on the unreliable memory without sacrificing the model inference accuracy, which helps to minimize the needed partition of kernel computation. Next, we introduce our insights on how to select data for computation in unreliable memory and vice versa.

3.2 Insights on Bit Robustness

When placing computation data in unreliable memory, we need to detect the correctness of computation results, specifically to identify errors in computation results caused by bit flips. While detection can be simply achieved through recomputation and result comparison, this would incur significant performance overhead or require double the computing resources.

We discovered that not all bit flips have equal impact on model accuracy, which presents an opportunity to reduce the fault detection overhead. Different from general-purpose computing, the non-linearity of model computation makes some bit flips have little impact on model accuracy. We conduct an experiment with the ResNet-50 model, flipping each bit in every kernel's input values individually and measuring the impact on model accuracy. The experiment results show that there exist *robust bits* whose flips have no impact on model accuracy; the remaining non-robust bits affect the accuracy and can be further categorized into *ranging bits* and *vulnerable bits*. The former ones may cause `NaN` and reduce model accuracy to 0, while the latter causes gradual accuracy degradation. Additionally, we found that some values consist entirely of robust bits and ranging bits (**type-1 values**), while others contain vulnerable bits (**type-2 values**).

For computations involving type-1 values, we compare results against pre-computed result thresholds; if ranging bits flip, results will fall outside these thresholds, which eases the fault detection. For type-2 values, recomputation is necessary to verify result correctness. We designed an asynchronous CPU-based verification method that doesn't impact GPU model inference, which is detailed in §4.4. For reliable memory allocation, we prioritize matrices with more type-2 values to minimize recomputation needs.

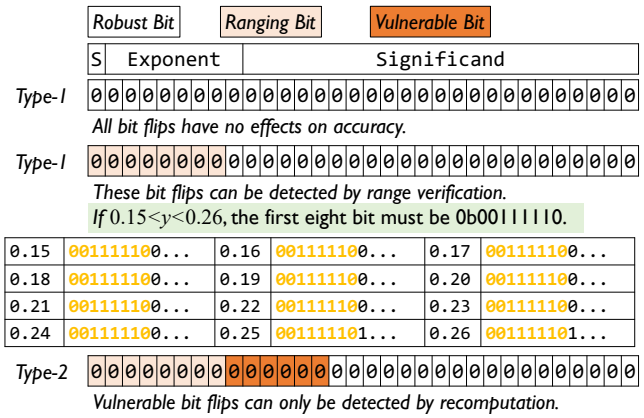
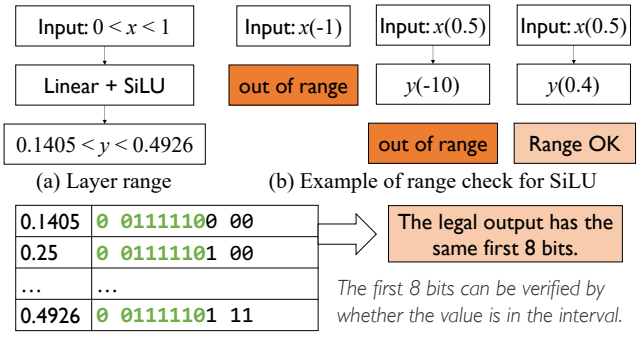


Figure 2: The classification of bits and its mapping.

Finding all the robust bits and ranging bits by testing all the bits is impractical due to the large number of bits. At the same time, the robustness of a single bit cannot be directly

extended to multiple bits. Thus, we explore the mathematical properties of the model to quickly find the robust bits and ranging bits and find a method to extend the single bit robustness to multiple bits. We found that the impact of bit flips on results depends on how big perturbation a model calculation can tolerate. If the change from the bit flip stays within this range, accuracy is not affected. We choose 10^{-5} for perturbation as it is a conservative estimation, ensuring common models remain unaffected by perturbations [3, 44, 72]. Such bit robustness comes from the mathematical properties of the model computation, and thus, we design an offline analysis method of bit identification (§4.2).

Ranging Bits. Current machine learning models rely on nonlinear activation functions. These functions, in combination with the input data range, produce a small, valid output range. This reduces the cost of numerical validation, as values outside this range can be identified as errors without computation, as shown in Figure 3(a, b). It is important to note that the input data range is also predefined. For example, pixel values typically fall within the range of 0-255 or 0-1 (a model program often maps pixel values to this smaller range). These output range constraints can be mapped to the specific bits of the number, reducing the number of bits that need to be verified, as illustrated in Figure 3(c).

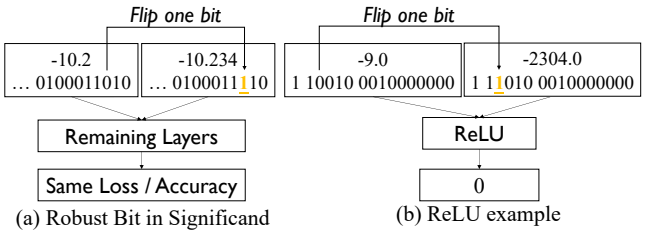


(a) Layer range (b) Example of range check for SiLU

(c) Ranging Bit Mapping

Figure 3: The ranging bits of the value and its fast verification.

Robust Bits. Robust bits can be identified based on the floating-point representation and the model structure. As shown in Figure 4(a), the last few bits of the significand have minimal impact on the results [27, 30, 63, 83], and can, therefore, be directly designated as robust bits. Additionally, the model structure contributes to robust bits. For instance, the negative input of a ReLU function does not affect the output as long as the input remains less than 0, as illustrated in Figure 4(b). These bits are marked as robust bits based on specific input conditions.



(a) Robust Bit in Significand (b) ReLU example

Figure 4: The flipped robust bits and their effects.

4 Design

4.1 Workflow Overview

SAVE is built on PyTorch, a widely used machine learning inference framework. Figure 5 illustrates its entire workflow, which consists of four key stages working together to achieve reliable inference:

Selection Stage performs a one-time offline static analysis to identify three types of bits: robust bits with minimal impact on accuracy, ranging bits that can be verified through range checks, and vulnerable bits identified conservatively as the remaining bits. The analysis results guide runtime memory allocation and verification.

Allocation Stage manages the placement of data between reliable and unreliable memory. Based on the analysis of the prior stage, it prioritizes placing computations with more vulnerable bits (more type-2 values) in reliable memory while assigning computations with more robust bits and ranging bits (more type-1 values) to unreliable memory. An eviction mechanism is also implemented to optimize the use of limited reliable memory resources.

Verification Stage detects faults caused by bit flips. It employs asynchronous CPU-GPU verification along with output range analysis results to minimize fault detection overhead.

Edit Stage addresses error recovery when faults are detected. It ensures that the latency of model inference remains unaffected in normal cases without bit flips. For cases where faults occur, SAVE restarts the inference process from the faulting model layer.

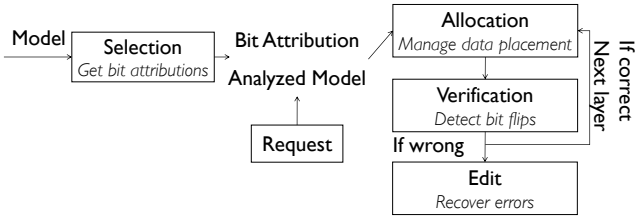


Figure 5: The overview of SAVE.

4.2 Selection Stage

This stage performs static analysis to classify three types of bits in each value: ranging bits that can be verified through

range checks, robust bits with minimal impact on accuracy, and vulnerable bits requiring strict protection. The analysis comprises two phases: range analysis and bit attribution analysis.

Phase 1: Range Analysis. SAVE determines the valid range for each value by propagating ranges through the model's computational graph layer by layer. Since the model structure is a directed acyclic graph (DAG) with deterministic operators (purely computational), this analysis can be performed statically. For simplicity, we assume each layer contains a single operator corresponding to one kernel.

Figure 6 illustrates this process with a concrete example: 1) SAVE initializes the input range as [0, 1] for the input layer. 2) This range is propagated to the next layer's operator (Op), which performs matrix multiplication. Since the weight matrix is fixed (model parameters), SAVE knows the relationship between input X and output Y of the Op. Based on the propagated input range, SAVE determines that the output range will be [-0.75, -0.1]. 3) The subsequent ReLU operator, which outputs zero for negative inputs, allows SAVE to determine that its output range is exactly 0. 4) For the final Add operator, one input is known to be 0 (from ReLU) and the other is the original input X ([0 to 1]), resulting in a final output range of [0 to 1]. The code snippet of this example is shown in Appendix B.

Phase 2: Bit Attribution Analysis. Using the ranges calculated in Phase 1, SAVE identifies ranging bits and robust bits.

Ranging Bits Identification. For each value, SAVE identifies ranging bits as those that must have the same value for all possible values within the computed range. These typically include the sign bit (e.g., must be 1 for range [-0.75 to -0.1]) and exponent bits constrained by the range bounds.

Robust Bits Identification. SAVE identifies two types of robust bits: 1) Value-based robust bits: For both upper and lower bounds of each range, if flipping a bit in the significand causes a numeric difference less than 10^{-5}, it is considered robust. The common robust bits between both bounds become the robust bits for that range. 2) Operator-induced robust bits: Some operators, particularly activation functions, create additional robust bits. For example, with ReLU and negative inputs, only the sign bit needs verification since all outputs will be zero regardless of the input's exact value.

Fine-Grained Analysis Results. Some operators, such as Op in the example above, exhibit points where the monotonicity of the input-output relationship changes. Some other operators may have extremely wide output ranges. In these situations, SAVE partitions the output range into multiple intervals and performs separate ranging and robust bit analyses for each interval. This approach enables more precise bit attribution during runtime based on actual values. To reuse the analyzed bit attribution results efficiently, SAVE stores the analyzed results in a bit attribution cache.

To manage cache pressure caused by storing bit properties across numerous intervals, SAVE optimizes memory usage by combining each linear operator with its subsequent nonlinear operator into a single composite operator. The cache then stores the ranging and robust bits for these combined operators instead of for individual ones.

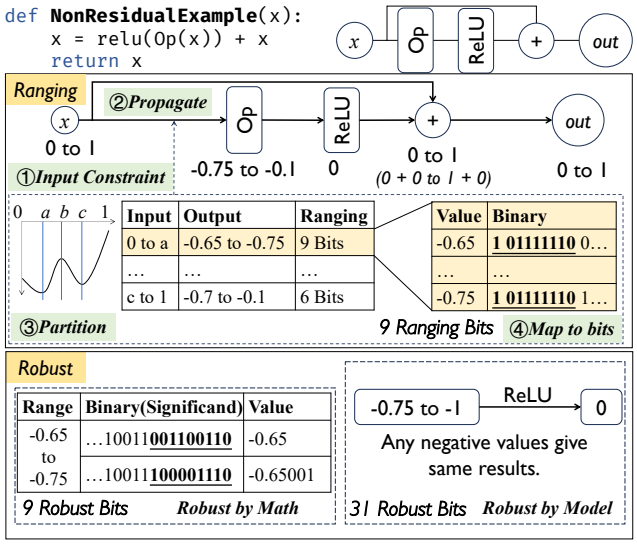


Figure 6: The process of range analysis in Selection stage.

4.3 Allocation Stage

While bit attributes can help reduce reliable memory usage in theory, existing computing and memory management interfaces do not support bit-level management. To address this limitation, SAVE employs matrix partitioning to manage memory allocation based on the proportion of vulnerable bits in each computation block after partitioning.

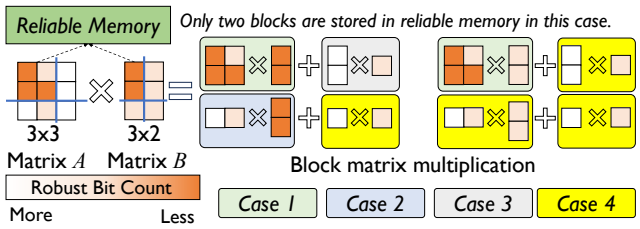


Figure 7: Blocking matrices multiplication examples and partitioning. Both matrix A and B in the figure are non-parameter matrices.

Specifically, when executing GPU kernels, SAVE partitions the input matrices into four sub-matrices, with the partitioning strategy designed to create significant differences in the proportion of vulnerable bits across blocks, as shown in Figure 7. During memory allocation, sub-matrices with higher

proportions of vulnerable bits are prioritized for placement in reliable memory, and sub-matrices with lower proportions of vulnerable bits are preferentially allocated to unreliable memory. The weight matrices of kernels (model parameters) are always allocated in the unreliable memory. To implement this, SAVE maintains two separate memory pools for reliable and unreliable (normal) memory, and hooks into the memory management interface in the framework.

The allocation strategy also carefully considers the output memory placement for each sub-matrix computation. For example, computation results containing more ranging bits and robust bits are preferentially allocated to unreliable memory, as they can be efficiently verified through range checks. This creates four distinct scenarios for each partitioned matrix, depending on the input and output memory placements, as illustrated in Figure 7:

- Case 1: Both input and output in reliable memory.
- Case 2: Input in reliable memory, output in unreliable memory.
- Case 3: Input in unreliable memory, output in reliable memory.
- Case 4: Both input and output in unreliable memory.

For the cases above, SAVE adheres to the rules of in-place modification after the CPU has stored the values. Specifically, SAVE transfers a copy of the output to the CPU.

For computations with output stored in reliable memory (Cases 1 and 3), in-place modification significantly reduces memory usage. However, since SAVE needs to maintain a reliable result on the CPU, immediately performing in-place modifications would erase the trace of reliable computations, impacting subsequent verification. To address this, SAVE retains a copy in reliable memory until the CPU has fully received the result. For computations with output stored in unreliable memory (Cases 2 and 4), SAVE transfers the output to the CPU for verification. Once the CPU verifies the output, it is deemed reliable, and the corresponding data in unreliable memory is in-place modified. For input values stored in unreliable memory (Cases 3 and 4), SAVE verifies their correctness by comparing the input with the recorded reliable input.

4.4 Verification Stage

SAVE requires fault detection for computations involving unreliable memory. A straightforward approach would be to recompute each kernel and verify result consistency, as shown in Figure 8(a). However, since both recomputation and verification would occur on the GPU, subsequent kernel computations would be blocked until verification is completed, significantly impacting model inference time.

Instead, SAVE introduces an efficient verification mechanism that offloads most verification work to the CPU through PCIe and unified memory, minimizing the overhead on model inference latency. This mechanism handles two distinct cases:

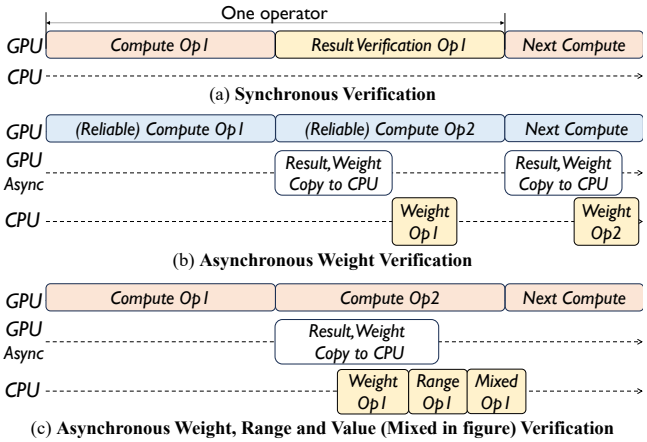


Figure 8: The overview of verification stage. The computation in red means the input is in unreliable memory. The computation in blue means the input is in reliable memory. The weight matrices are in unreliable memory.

Case 1: Kernel Input in Reliable Memory. When input matrices are placed in reliable memory (some partitioned matrices can fit in the reliable memory owing to the allocation stage), only the model parameters (weights) stored in unreliable memory need verification. SAVE asynchronously copies weights from GPU memory to CPU after each computation and uses parallel CPU threads to compare against the original CPU-side copy. As illustrated in Figure 8(b), weight transfers occur through the GPU's DMA engine without blocking kernel execution, allowing weight verification to run concurrently with the next kernel's computation.

Case 2: Kernel Input in Unreliable Memory. When input matrices reside in unreliable memory, SAVE performs three types of verification: weight verification as described above, range verification for ranging bits, and lightweight recomputation for vulnerable bits.

Range Verification. To verify numerical ranges efficiently, SAVE splits the work between GPU and CPU. Specifically, SAVE implements a range verification kernel as shown in Appendix C. It retrieves valid ranges for each value from the selection stage cache and launches the kernel. The kernel computes the differences between the value and its range bounds, uses the `signbit` function to get the sign bits of these differences, and performs OR on the sign bits to produce the output (where a zero indicates the value falls within range). After the kernel completes, the output matrix is transferred to CPU memory, where multiple CPU cores verify that all elements equal zero, confirming no ranging bit flips occurred.

This GPU-CPU split design is intentional because performing all verification on the CPU would introduce significant latency that would impact end-to-end model inference time. By dividing the work between GPU (numerical operations)

and CPU (final verification), SAVE achieves better overall performance. During the whole verification, the bandwidth of PCIe is not the bottleneck since not all values need to be verified on the CPU. To further minimize overhead, the range verification kernel is fused with its corresponding computation kernel.

Mixed Precision Recomputation Verification. Recomputation is required to verify the calculated results involving vulnerable bits. Directly recomputing multiplication or division kernels on the CPU would be highly time-consuming. To address this, SAVE aims to minimize the recomputation burden, enabling it to be conducted asynchronously on the CPU without introducing significant performance overhead.

Since ranging bits are verified by their ranges and robust bits are ignored, the number of bits requiring verification is reduced to less than half of the original count. This reduction allows the verification process to shift from high-precision computation (heavyweight) to low-precision computation (lightweight). Figure 9 illustrates the conversion process.

Specifically, for multiplication or division of x and y, the sign bit of the result can be obtained by XORing the sign bits of x and y. The exponent bits of the result can be obtained by adding or subtracting the exponent parts of x and y. Once the new exponent bits are determined, SAVE re-estimates the robust bits and removes the robust bits for further low-precision verification. The significand of the result can be computed by converting the significand of x and y (excluding robust bits) into 16-bit or 8-bit integers (low and mixed precision), then performing multiplication or division. CPU SIMD instructions are leveraged to accelerate these low-precision integer operations in parallel.

Appendix D presents the pseudocode for multiplication recomputation running on CPU. Lines 7 and 11 extract the significand of x and y. Since normalized floating-point numbers (1.significand) omit the leading 1 in their integer part, lines 24-25 restore this implicit 1. Then, robust bits are removed through shift operations before performing the final multiplication.

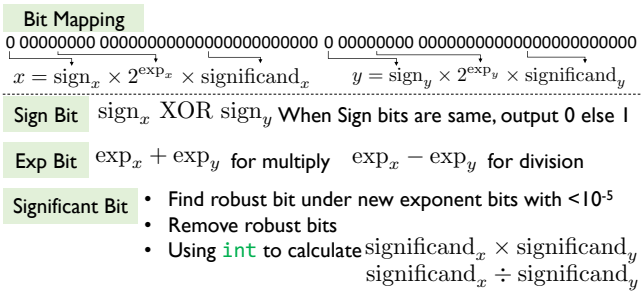


Figure 9: The verification process of multiplication and division.

The addition and subtraction kernels are very lightweight on GPU (and are hard to efficiently implement on CPU) thereby, their recomputation does not incur much overhead on the overall model inference latency. So, SAVE verifies the results by simply recomputing them.

5 Evaluation

To show the effectiveness of SAVE, we evaluate it by answering the following questions:

- What is the overhead of SAVE for normal execution?
- How effective is SAVE in protecting against bit flips?
- What is the overhead of each part of SAVE?

5.1 Evaluation Setup

We evaluate the effectiveness of our method through the approach of simulated flipping. We use GPUs for desktop PCs and embedded devices as the platform to evaluate the performance of SAVE. We follow our fault model, wherein the registers, cache, and a specific portion of memory (exactly 6MB) are assumed to be free from bit flips, and the CPU is considered free from bit flips. Our bit flip setting aligns with previous work [89], using real satellite flip characteristics from RedNet [89] to build a physical flip bit trace to test SAVE's effectiveness.

Baseline. To demonstrate the effectiveness of SAVE, we compare it against the unmodified original model (Original), Triple Modular Redundancy (TMR), and state-of-the-art protection methods (Dr.DNA [61] and RedNet [89]). Since RedNet cannot be directly applied to models, we retrain the model with RedNet, ensuring that the training configurations match those of the original model. For Dr.DNA, which includes several mitigation methods outlined in the original paper, we adopt the fastest mitigation strategy to evaluate end-to-end performance without errors. For TMR, we employ its mitigation strategy to measure recovery accuracy.

5.2 Metrics: *AccurateLatency*

In a fault-tolerant system, maintaining model correctness is crucial. To evaluate model performance, we propose a new metric, *AccurateLatency*, defined as: $\text{AccurateLatency} = \text{Latency} \times (1 + \text{ErrorRate})$.

This metric represents the average inference time under the assumption that a second inference is always correct when given an *ErrorRate*. If the error rate is 0%, the *AccurateLatency* is equal to the original latency. Instead, if the error rate is 100%, the *AccurateLatency* is equal to double the original latency, meaning the first inference does not produce any correct results. It shows the optimal average correct inference time. For a system with inference protection, an increase in the *ErrorRate* suggests that some errors have slipped through the system or that there has been an impact on the accuracy rate compared to the original model.

5.3 End-to-end Performance

We first evaluate the overall performance impact of adding SAVE and analyze the time overhead required for model in-

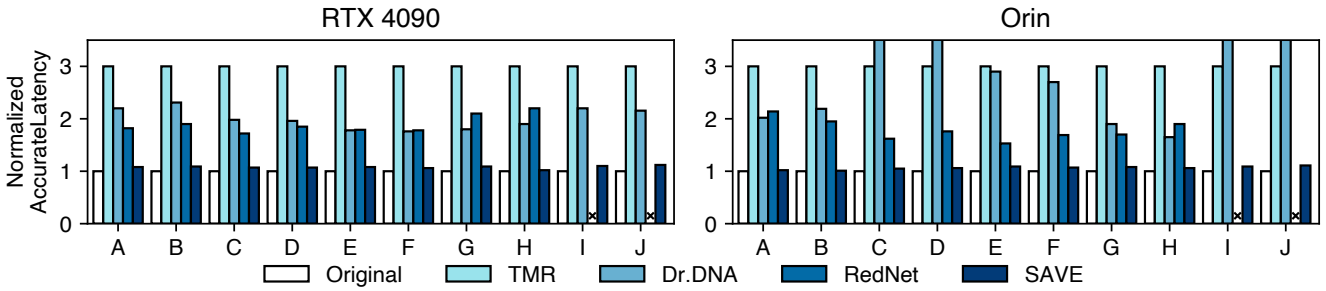


Figure 10: The accurate latency of several models with SAVE. ×: RedNet cannot apply to CogACT and RDT due to the activation function changes. The models used in this evaluation are shown in Table 2.

Table 2: The models used in our evaluation. *: We evaluate the validness of the model output by checking whether the output action is reachable.

	Model	Dataset
A	ViT [23]	ImageNet-1K [21]
B	ViT [23]	AID [93]
C	ResNet-50 [36]	ImageNet-1K [21]
D	ResNet-50 [36]	AID [93]
E	MobileNetV2 [42]	ImageNet-1K [21]
F	MobileNetV2 [42]	AID [93]
G	Decision Transformer [15]	ValidCheck*
H	Decision Transformer-M [15]	ValidCheck*
I	CogACT [55]	ValidCheck*
J	RDT [59]	ValidCheck*

ference. Normalized latency is used to present performance across different models and datasets.

As shown in Figure 10, SAVE introduces minimal overhead compared to the original model execution while maintaining high accuracy. On average, SAVE adds only 8% latency overhead across various models and datasets, which is significantly lower than that of TMR (3×). This low overhead is achieved through our efficient vulnerable bit identification and selective protection strategy. For vision models such as ViT, the overhead is slightly higher (8–10%) due to the increased number of matrix operations requiring protection. Compared to other state-of-the-art methods, SAVE achieves better accuracy. For Decision and Embodied AI models, RedNet's accuracy drops by more than half due to changes in activation functions, leading to substantial performance degradation.

5.4 Accuracy under Bit Flips

We evaluate the fault tolerance capability of SAVE under two bit flip patterns with varying bit flip probabilities. The flip patterns are virtual consecutive address flips and physically contiguous region flips. The first pattern is more likely to occur during memory access, while the second pattern is

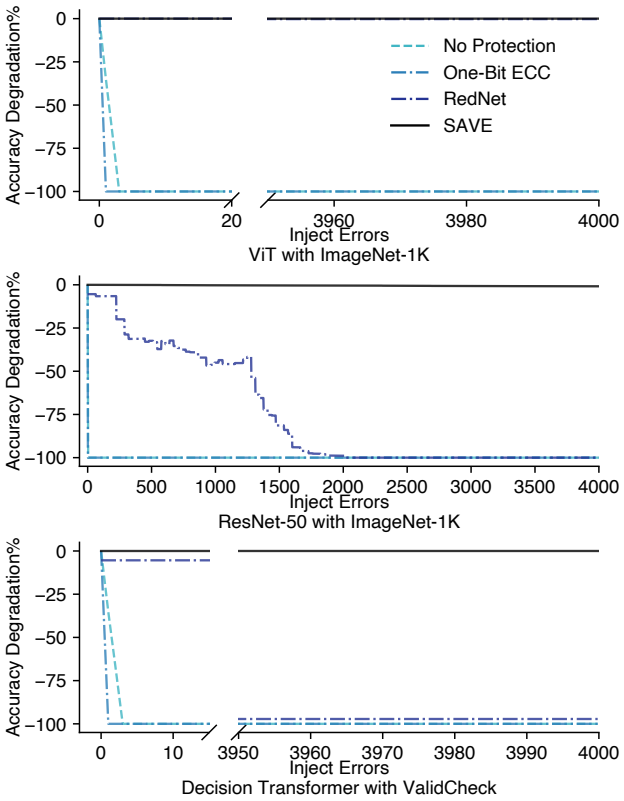


Figure 11: The accuracy degradation under virtual consecutive address flips.

typically caused by physical interference. We test the overall inference accuracy and inference time after attacks with different bit flip probabilities. The accuracy refers to the correctness rate after flipping bits across 100 consecutive inferences at the same position. Additionally, the correctness of Dr.DNA is comparable to SAVE in our tests under the fastest mitigation selecting strategy. We also examine single-bit error correction using ECC, noting that when two-bit errors occur, ECC fails to repair them. This results in program termination or infer-

ence failure. We mark this situation as 0% accuracy. If ECC's correction capability is extended to n-bit errors, the accuracy curve will shift to the right by n bits.

5.4.1 Virtual Consecutive Address Flips

We simulate virtual address consecutive memory flips by randomly modifying the bits on the addresses, and we test the accuracy after attacks involving selecting random flips among all bits. As shown in Figure 11, SAVE maintains high accuracy for up to 4096 consecutive bit flips for both ResNet-50 model and Decision Transformer. In contrast, the original model's accuracy drops significantly to below 5% with just 3 consecutive bit flips. Due to the activation function changes, RedNet's accuracy on Decision Transformer quickly drops below 10% after 32 bit flips, while on ResNet-50 with ImageNet-1K, RedNet's truncation effect causes the accuracy to continuously decline, dropping below 10% after 1600 bit flips. Due to error value detection and repair mechanisms, both Dr.DNA and SAVE maintain zero accuracy degradation. The results demonstrate that SAVE's selective protection strategy effectively preserves model accuracy under bit flip attacks while introducing minimal overhead.

5.4.2 Physically Contiguous Region Flips

When facing the physical interference, the bit flip impact is not continuous in a row but continuous in the physical space. This makes the impact of the model appear to be flipped at intervals. For this, SAVE tests the physical flip pattern given in RedNet with exactly the same proportion and simulation setup.

As shown in Figure 12, The physically contiguous bit flips cause changes in values at multiple-spaced positions, which differs from virtual consecutive address flips, where 32 consecutive bits need to be flipped to affect a single value. The results also show the same trend. RedNet, which employs modified activation functions, experiences significantly faster accuracy degradation under physically contiguous flips compared to virtual consecutive flips, with the model becoming almost completely ineffective after 1500 bit flips. For the Decision Transformer, due to the non-adjacency between flipped bits and spatial bits, errors rapidly propagate throughout the entire model, causing complete model failure with just 6 bit flips. Both Dr.DNA and SAVE maintain accuracy fluctuations within 1% across all models, which can be attributed to normal performance variations.

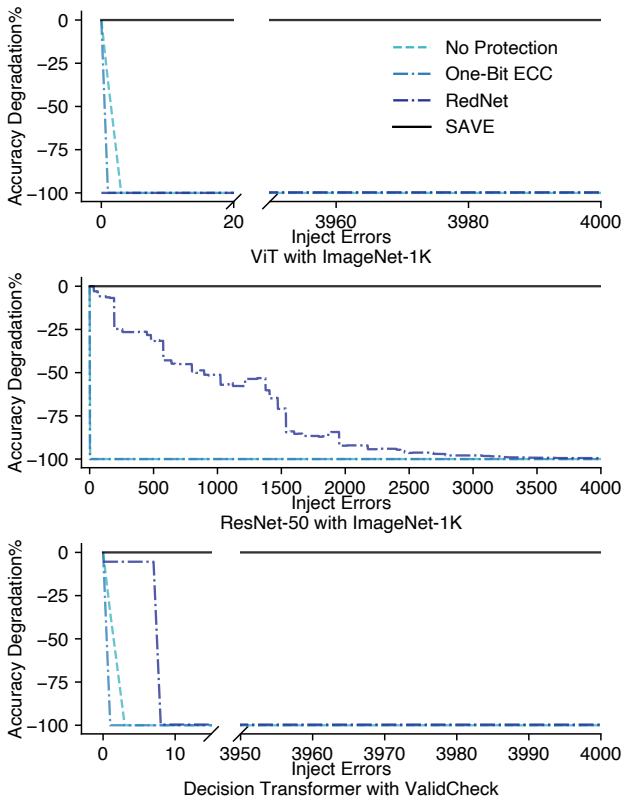


Figure 12: The accuracy degradation under physical contiguous region flips.

5.5 Sensitivity Analysis

5.5.1 Performance of different reliable memory sizes

One important part of SAVE is the size of reliable memory. To evaluate this, we test the performance overhead for different models with various working sets under different sizes of reliable memory. The results are shown in Figure 13. When the working set size is smaller than the reliable memory, reliability is guaranteed with nearly zero overhead, and SAVE shows almost no noticeable overhead. When the working set size exceeds the reliable memory, extra computation is required to ensure reliability in inference. By combining bit properties with ranging properties, the verification cost is reduced by 71%. The reduced calculations come mainly from skipping the validation of robust bits, and using the properties of the range to assess the results.

5.5.2 Performance of different GPUs

Different GPUs produce different performance due to their varying computing capability. We test the performance overhead of SAVE on different GPUs by measuring their overhead on latency. As shown in Figure 14, the additional overhead generated by the model on all platforms is less than 9%. The error rate of the model does not increase during our test.

5.6 Ablation Study

To show the detailed characteristics of each stage in SAVE, we analyze the effect of each stage.

5.6.1 Selection Stage: Analysis Time.

Model analysis in SAVE is a one-time task. Identical models do not require re-analysis. If a model is updated, re-analysis is necessary. A full analysis is required if the model's structure

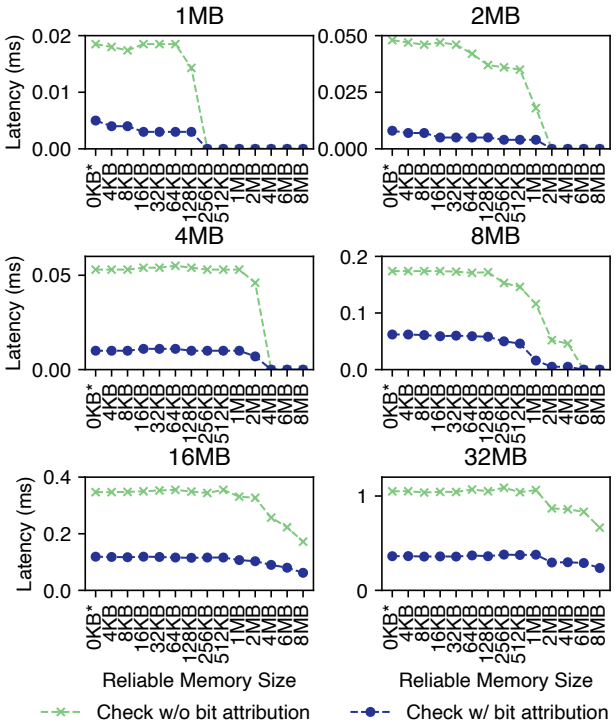


Figure 13: The performance overhead over different size working sets and reliable memory. 0KB means that the reliable memory is not used.

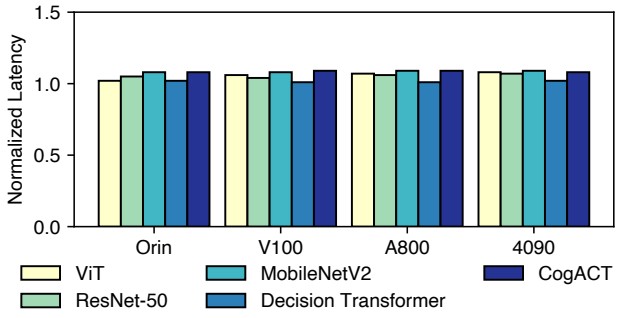


Figure 14: The performance overhead of different GPUs.

changes or all parameters are replaced. If only some parameters change, only the affected parts need analysis. To evaluate this, we test the time required for the same type of model to analyze different data sizes. As shown in Figure 15, the analysis time is under 10 seconds for all models when a full model change occurs. As the size of the modified parameters decreases, the analysis time in SAVE correspondingly decreases.

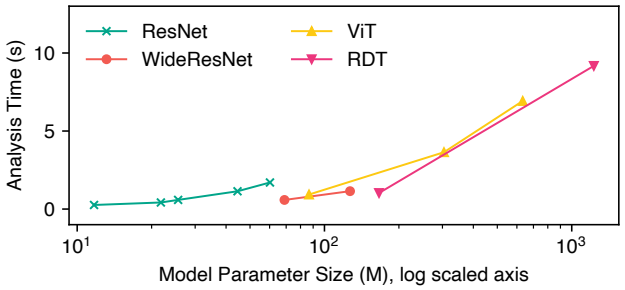


Figure 15: The analysis time of different models.

5.6.2 Selection Stage: Correctness Rate.

The key to the result reliability comes from the selection stage of SAVE. If there are too many false negatives, SAVE will fail to protect the model due to ignoring too many vulnerable bits. If there are too many false positives, SAVE will incur significant extra overhead due to protecting too many robust bits. To show the correctness, we scan all 374,372,224 weight bits in ResNet-18 model with ImageNet-1K dataset and count the number of bits that are correctly identified as robust or non-robust, as shown in Table 3. This robustness property refers to single bit flip effects. Although the proportion of robust bits exceeds 76.5%, they are widely distributed throughout the entire region, which means it is infeasible to drop the 76.5% robust bits.

Table 3: The correctness rate of bit analysis in the selection stage.

	True Robust	False Robust
Predicted Non-Robust	40,866,576 10.9%	47,061,573 12.5%
Predicted Robust	286,444,075 76.5%	0 0.0%

The distribution of different bits is highly related to the operator. For matrix multiplication of ViT, each number has at least one ranging bit, with a total of 90% robust bits. For the normalization operator, each number has at least three vulnerable bits, with a total of only 55% robust bits. We show the bit attribution distribution for some outputs after ResNet's activation function and for ViT's output after normalization, as shown in Figure 16.

5.6.3 Allocation Stage: Memory Movement Size.

SAVE reduces the number of bits to be protected by identifying robust bits in the previous step, thereby reducing the number of times data is transferred using reliable memory and thus reducing memory transfer overhead. We conducted an experiment with the ViT model and showed the amount of reliable memory used under different configurations of SAVE.

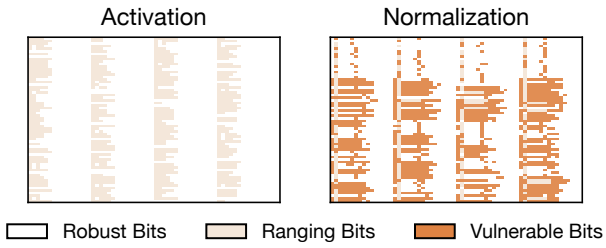


Figure 16: The distribution of different bits.

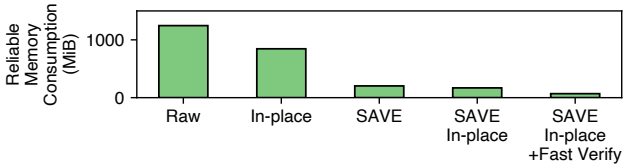


Figure 17: The reliable memory consumption of SAVE.

After completing the selection stage offline, SAVE places only 17% of the original inference memory into reliable memory, significantly reducing the memory footprint, as shown in Figure 17. Moreover, since many values only have the sign bit and the highest bit of the exponent being unreliable, SAVE uses range check to fast verify the correctness, thus replacing reliable memory calculation with judgment. This fast path can further reduce the memory footprint by 95% compared to the model's single inference, as shown in Figure 17.

5.6.4 Verification Stage: Verification Efficiency.

SAVE uses asynchronous verification on the CPU to reduce the overhead of verification. We first test the performance of using the CPU for mixed precision verification, as shown in Figure 18. Through fully utilizing the SIMD and dedicated copy engine on GPU, the matrix multiplication of 1024×1024 only takes 38ms, which is 98.2% less overhead compared to direct CPU floating-point multiplication. We also test the performance of asynchronous verification, as shown in Figure 19. We execute 100 times of verification and calculate the total time to reduce the fluctuation. The results show that the asynchronous verification incurs 0.05% overhead of frontend computation. Compared with synchronous verification, the overhead is reduced by 22%.

Resource Overhead. SAVE uses asynchronous verification, which requires additional CPU resources for verification. We test the CPU overhead of SAVE and found that it incurs 20% additional CPU resource overhead compared to the original model. For the host memory, SAVE only incurs an additional size exactly the same as the model parameter size.

5.6.5 Edit Stage: Recovery Time.

The recovery time is also important after an error occurs. We inject a single bit flip to trigger `NaN` value to test the recovery time of different methods.

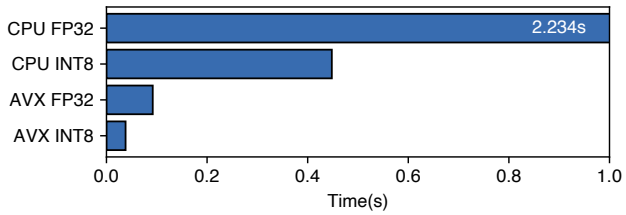


Figure 18: The mixed precision verification performance of SAVE.

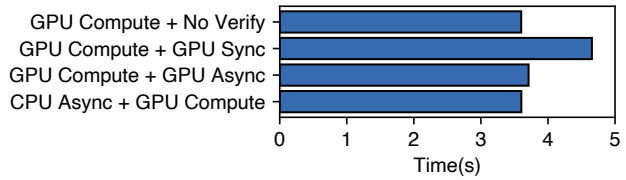


Figure 19: The asynchronous verification performance of SAVE.

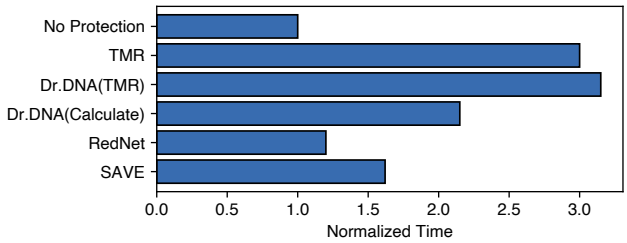


Figure 20: The recovery time of different methods.

As shown in Figure 20, the recovery time of SAVE is higher than that of RedNet since RedNet simply clips out-of-range values, whereas SAVE must synchronize valid values and perform partial recomputation in reliable memory. The major difference in recovery time between SAVE and RedNet is the overhead of recomputation. In the correct method system comparison, the recovery time of SAVE is 24% less than Dr.DNA and 46% less than TMR, which is due to SAVE's reduced computation, only recomputing the range where the error occurs.

5.6.6 Support for lower precision models

When performing inference on the edge, fp16 or even 8-bit integer quantization is often used. SAVE is not bound to the model precision, so it can be directly applied to other precision models. In other precision models, the total number of robust bits will decrease due to the reduction in the model's overall bit length, leading to increased overhead. To show the performance of SAVE in lower precision models, we test the ViT model with ImageNet-1K dataset. Since we cannot

remove the quantization effects on the accuracy and neither Dr.DNA nor RedNet can be applied to lower precision models, we only show the latency of SAVE. As shown in Figure 21, SAVE achieves 10% to 15% overhead compared to the original model.

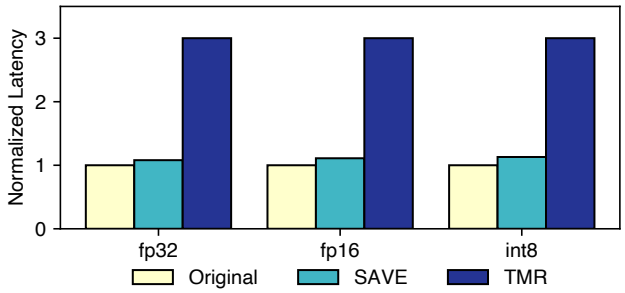


Figure 21: The performance of SAVE in lower precision models.

6 Discussion

We are currently focusing on small-scale edge models because they are more safety-critical in scenarios like decision-making on self-driving vehicles, drones, industrial robotic arms, or satellites. In contrast, large language models supporting chat systems or multi-modal generation systems may tolerate occasional inference errors, as users can easily detect irrelevant responses or generated images with minor imperfections (which may not even affect user experience). The design of SAVE is flexible across different models and can withstand multiple bit flips as long as the changes of these bits are smaller than the model's allowed perturbation. While SAVE could potentially be applied to large language models, additional work would be needed, such as protecting KV Cache, which we leave as future work.

SAVE relies on a portion of reliable memory to enhance model inference reliability, which can be much smaller than the average memory usage during model inference. We believe this research has important implications for hardware-software co-design supporting reliable model inference in space and other extreme environments. To our knowledge, there are currently no GPU computing cards that can fully resist cosmic radiation. Our design demonstrates that through the combination of hardware-implemented small reliable memory and software-implemented fault tolerance, we can effectively improve inference reliability against bit flips.

7 Related Work

Hardware based protection. Various hardware mechanisms have been proposed to mitigate bit flip attacks efficiently. Some work [49, 70, 78, 81, 95] increase DRAM refresh rate or limit row access times within one refresh interval to protect potential victim rows. ZebRAM [50], GuardION [85] and CATT [11] protect sensitive data from vulnerable rows. DEACT [31] prevents vulnerable rows from being accessed further. Some work [77, 91, 92, 100] leverage in-DRAM swapping to shuffle the DRAM rows. AEP [98] uses masks on hardware to protect the model. DNN-Defender [99] use gradients to detect critical bit and swap the rows in DRAM. HARDeNN [48] adds three-module redundancy for weights and activation layers. Compared to SAVE, these work requires complex hardware implementation, focusing on bit flips by RowHammer attack. They support for specific-structured models. SAVE provides a general solution to all models without hardware or model modification. Hardware-based ECC also serves as a hardware bit flip mitigation method. RowHammer [22, 49] states ECC is useful to defend RowHammer attack despite of the expensive cost. However, some work [14, 18] show that carefully-designed ECC-aware RowHammer could bypass these ECC mechanisms and the ECC can only correct one bit and detect two bits typically [66] and GPUs [65]. SAVE can tolerant multiple bit flips.

Software based protection. Some software-implemented solutions have been proposed to reduce the impact of bit flips on the entire model. Software Error Resilience [45] use TMR for protecting silent data corruption. TBD [40] proposes quantization to restrict the bound of DNN parameters. Parameter binarization [37, 76] construct binarized DNN to flatten the bit flip's impact to parameter's neighbors. Some work [17, 56, 62, 94] enhance the ability of models to resist adversarial attacks. Some work [7, 28, 41, 58, 88] use different encoding schemes to protect the model parameters. Modelshield [33], Hashtag [47] and RADAR [53] introduce hashing-based methods to generate weights group signature and make verification for parameter. RedNet [89] gives the first systematic study of bit flips in DNNs in space and proposes a software-based solution to mitigate bit flips in DNNs. Compared to these work, SAVE keeps the model accuracy and ensures the inference latency by introducing fast verification.

8 Conclusion

SAVE provides a software-based inference fault tolerance solution, which preserves model structure to provide the largest compatibility with existing models. It uses GPU memory management and lightweight runtime verification to mitigate bit flips with acceptable overhead. The results show that SAVE incurs less than 9% performance overhead for end-to-end latency in model inference while maintaining accuracy.

Acknowledgments

We sincerely thank our shepherd, Jongyul Kim, and the anonymous reviewers of ATC 2025 and SOSP 2024, whose reviews, feedbacks, and suggestions have significantly strengthened our work. This research was supported in part by the National Natural Science Foundation of China (No. 62432010,62132014,62202292). Corresponding author: Jinyu Gu (gujinyu@sjtu.edu.cn).

References

[1] Training ai for self-driving vehicles: the challenge of scale.

[2] Designing an optimal ai inference pipeline for autonomous driving | nvidia technical blog. https://developer.nvidia.com/blog/designing-an-optimal-ai-inference-pipeline-for-autonomous-driving/, March 2022. (Accessed on 04/04/2024).

[3] Abulikemu Abuduweili and Changliu Liu. Estimating neural network robustness via lipschitz constant and architecture sensitivity. *arXiv preprint arXiv:2410.23382*, 2024.

[4] Shahanur Alam, Chris Yakopcic, Qing Wu, Mark Barnell, Simon Khan, and Tarek M Taha. Survey of deep learning accelerators for edge and emerging computing. *Electronics*, 13(15):2988, 2024.

[5] Tooba Arifeen, Abdus Sami Hassan, and Jeong-A Lee. Approximate triple modular redundancy: A survey. *IEEE Access*, 8:139851–139867, 2020.

[6] Sahar Arshad, Seemab Latif, Ahmad Salman, and Saadia Irfan. Increasing profitability and confidence by using interpretable model for investment decisions. *ArXiv*, abs/2312.16223, 2023.

[7] Ali Asgari Khoshouyeh, Florian Geissler, Syed Qutub, Michael Paulitsch, Prashant Nair, and Karthik Pattabiraman. Structural Coding: A Low-Cost Scheme to Protect CNNs from Large-Granularity Memory Faults. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–17, Denver CO USA, November 2023. ACM.

[8] Fatemeh Ayatolahi, Behrooz Sangchoolie, Roger Johansson, and Johan Karlsson. A study of the impact of single bit-flip and double bit-flip errors on program execution. In Friedemann Bitsch, Jérémie Guiochet, and Mohamed Kaâniche, editors, *Computer Safety, Reliability, and Security - 32nd International Conference, SAFECOMP 2013, Toulouse, France, September 24-27, 2013. Proceedings*, volume 8153 of *Lecture Notes in Computer Science*, pages 265–276. Springer, 2013.

[9] Jiawang Bai, Baoyuan Wu, Yong Zhang, Yiming Li, Zhifeng Li, and Shu-Tao Xia. Targeted attack against deep neural networks via flipping limited weight bits. *arXiv preprint arXiv:2102.10496*, 2021.

[10] Scotty Black and Christian Darken. Scaling artificial intelligence for digital wargaming in support of decision-making. *arXiv preprint arXiv:2402.06075*, 2024.

[11] Ferdinand Brasser, Lucas Davi, David Gens, Christopher Liebchen, and Ahmad-Reza Sadeghi. Can't touch this: Practical and generic software-only defenses against rowhammer attacks. *arXiv preprint arXiv:1611.08396*, 2016.

[12] Jakub Breier, Xiaolu Hou, Dirmanto Jap, Lei Ma, Shivam Bhasin, and Yang Liu. DeepLaser: Practical Fault Attack on Deep Neural Networks, September 2018. arXiv:1806.05859 [cs].

[13] Martí Caro, Axel Brando, and Jaume Abella. Software-only semantic diverse redundancy for high-integrity ai-based functionalities. In *ERTS2024*, 2024.

[14] Anirban Chakraborty, Sarani Bhattacharya, Sayandeep Saha, and Debdeep Mukhopdhyay. Rowhammer induced intermittent fault attack on ecc-hardened memory. *Cryptology ePrint Archive*, 2020.

[15] Lili Chen, Kevin Lu, Aravind Rajeswaran, Kimin Lee, Aditya Grover, Michael Laskin, Pieter Abbeel, Aravind Srinivas, and Igor Mordatch. Decision transformer: Reinforcement learning via sequence modeling. In Marc'Aurelio Ranzato, Alina Beygelzimer, Yann N. Dauphin, Percy Liang, and Jennifer Wortman Vaughan, editors, *Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021, NeurIPS 2021, December 6-14, 2021, virtual*, pages 15084–15097, 2021.

[16] Zitao Chen, Guanpeng Li, and Karthik Pattabiraman. A low-cost fault corrector for deep neural networks through range restriction. In *51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2021, Taipei, Taiwan, June 21-24, 2021*, pages 1–13. IEEE, 2021.

[17] Moustapha Cisse, Piotr Bojanowski, Edouard Grave, Yann Dauphin, and Nicolas Usunier. Parseval networks: Improving robustness to adversarial examples, 2017.

[18] Lucian Cojocar, Kaveh Razavi, Cristiano Giuffrida, and Herbert Bos. Exploiting correcting codes: On the effectiveness of ecc memory against rowhammer attacks. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 55–71. IEEE, 2019.

[19] U.S. Election Assistance Commission Darrick Forester. Environmental hardware test plan, October 2018. [Online; accessed 2025-01-12].

[20] David. What is ecc memory? ecc memory benefits - assured systems, 3 2019. [Online; accessed 2025-01-08].

[21] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pages 248–255, 2009.

[22] Andrea Di Dio, Koen Koning, Herbert Bos, and Cristiano Giuffrida. Copy-on-flip: Hardening ecc memory against rowhammer attacks. In *NDSS*, 2023.

[23] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. An image is worth 16x16 words: Transformers for image recognition at scale. In *International Conference on Learning Representations*, 2021.

[24] Douglas Eadline. Close to the edge: If i don't use ecc, have i made an error? | edge computing series | edge computing, 6 2019. [Online; accessed 2025-01-08].

[25] ATP Electronic. Blog list | atp electronics, April 2020. [Online; accessed 2025-01-02].

[26] ATP Electronics. Sram soft error detection feature in e.mmc | atp electronics, April 2020. [Online; accessed 2025-01-02].

[27] Parisa Eslami and Houbing Song. Stability analysis of deep neural networks under adversarial attacks and noise perturbations. In *2024 International Wireless Communications and Mobile Computing (IWCMC)*, pages 150–155. IEEE, 2024.

[28] Xianglong Feng, Mengmei Ye, Ke Xia, and Sheng Wei. Runtime fault injection detection for fpga-based dnn execution using siamese path verification. In *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 786–789. IEEE, 2021.

[29] Ninnart Fuengfusin and Hakaru Tamukoh. NaN Attacks: Bit-Flipping Deep Neural Network Parameters to NaN or Infinity. In *2024 1st International Conference on Robotics, Engineering, Science, and Technology (RESTCON)*, pages 33–37, Pattaya, Thailand, February 2024. IEEE.

[30] Ido Galil and Ran El-Yaniv. Disrupting deep uncertainty estimation without harming accuracy. *Advances in Neural Information Processing Systems*, 34:21285–21296, 2021.

[31] Tesfamichael Gebregziabher Gebrehiwot, Fitsum Assamnew Andargie, and Mohammed Ismail. DEACT: Hardware Solution to Rowhammer Attacks. *Journal of Computer Science*, 19(7):861–876, July 2023.

[32] Max Ghiglione and Vittorio Serra. Opportunities and challenges of ai on satellite processing units. In *Proceedings of the 19th ACM international conference on computing Frontiers*, pages 221–224, 2022.

[33] Yanan Guo, Liang Liu, Yueqiang Cheng, Youtao Zhang, and Jun Yang. ModelShield: A Generic and Portable Framework Extension for Defending Bit-Flip based Adversarial Weight Attacks. In *2021 IEEE 39th International Conference on Computer Design (ICCD)*, pages 559–562, Storrs, CT, USA, October 2021. IEEE.

[34] Jon Duncan Hagar. Iot test planning and strategy for hardware and software. In *IoT System Testing: An IoT Journey from Devices to Analytics and the Edge*, pages 83–113. Springer, 2022.

[35] Richard W Hamming. Error detecting and error correcting codes. *The Bell system technical journal*, 29(2):147–160, 1950.

[36] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, 2015.

[37] Zhezhi He, Adnan Siraj Rakin, Jingtao Li, Chaitali Chakrabarti, and Deliang Fan. Defending and Harnessing the Bit-Flip Based Adversarial Weight Attack. In *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 14083–14091, Seattle, WA, USA, June 2020. IEEE.

[38] Dan Hendrycks and Kevin Gimpel. Bridging nonlinearities and stochastic regularizers with gaussian error linear units. *CoRR*, abs/1606.08415, 2016.

[39] Le Ha Hoang, Muhammad Abdullah Hanif, and Muhammad Shafique. Ft-clipact: Resilience analysis of deep neural networks and improving their fault tolerance using clipped activation. In *2020 Design, Automation & Test in Europe Conference & Exhibition, DATE 2020, Grenoble, France, March 9-13, 2020*, pages 1241–1246. IEEE, 2020.

[40] Sanghyun Hong, Pietro Frigo, Yigitcan Kaya, Cristiano Giuffrida, and Tudor Dumitras. Terminal brain damage: Exposing the graceless degradation in deep neural networks under hardware fault attacks. pages 497–514, August 2019.

[41] Fateme S Hosseini, Qi Liu, Fanruo Meng, Chengmo Yang, and Wujie Wen. Safeguarding the intelligence of neural networks with built-in light-weight integrity marks (lima). In *2021 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 1–12. IEEE, 2021.

[42] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications, 2017.

[43] Boming Huang, Yuxiang Huan, Li Da Xu, Lirong Zheng, and Zhuo Zou. Automated trading systems statistical and machine learning methods and hardware implementation: a survey. *Enterprise Information Systems*, 13(1):132–144, 2019.

[44] Xiaowei Huang, Daniel Kroening, Wenjie Ruan, James Sharp, Youcheng Sun, Emese Thamo, Min Wu, and Xinping Yi. A survey of safety and trustworthiness of deep neural networks: Verification, testing, adversarial attack and defence, and interpretability. *Computer Science Review*, 37:100270, 2020.

[45] Younis Ibrahim, Haibin Wang, Man Bai, Zhi Liu, Jianan Wang, Zhiming Yang, and Zhengming Chen. Soft error resilience of deep residual networks for object recognition. *IEEE Access*, 8:19490–19503, 2020.

[46] Dirmanto Jap, Yoo-Seung Won, and Shivam Bhasin. Fault injection attacks on softmax function in deep neural networks. In *Proceedings of the 18th ACM International Conference on Computing Frontiers*, pages 238–240, 2021.

[47] Mojan Javaheripi and Farinaz Koushanfar. Hashtag: Hash signatures for online detection of fault-injection attacks on deep neural networks. In *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, pages 1–9. IEEE, 2021.

[48] Navid Khoshavi, Mohammad Maghsoudloo, Arman Roohi, Saman Sargolzaei, and Yu Bi. Hardenn: Hardware-assisted attack-resilient deep neural network architectures. *Microprocessors and Microsystems*, 95:104710, 2022.

[49] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping bits in memory without accessing them: An experimental study of dram disturbance errors. *ACM SIGARCH Computer Architecture News*, 42(3):361–372, 2014.

[50] Radhesh Krishnan Konoth, Marco Oliverio, Andrei Tatar, Dennis Andriesse, Herbert Bos, Cristiano Giuffrida, and Kaveh Razavi. {ZebRAM}: Comprehensive and compatible software protection against rowhammer attacks. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 697–710, 2018.

[51] Jingwen Leng, Alper Buyuktosunoglu, Ramon Bertran, Pradip Bose, and Vijay Janapa Reddi. Asymmetric resilience for accelerator-rich systems. *IEEE Computer Architecture Letters*, 18(1):83–86, 2019.

[52] Hannu Leppinen. Current use of linux in spacecraft flight software. *IEEE Aerospace and Electronic Systems Magazine*, 32(10):4–13, 2017.

[53] Jingtao Li, Adnan Siraj Rakin, Zhezhi He, Deliang Fan, and Chaitali Chakrabarti. RADAR: Run-time Adversarial Weight Attack Detection and Accuracy Recovery. In *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 790–795, February 2021. arXiv:2101.08254 [cs].

[54] Jingtao Li, Adnan Siraj Rakin, Yan Xiong, Liangliang Chang, Zhezhi He, Deliang Fan, and Chaitali Chakrabarti. Defending Bit-Flip Attack through DNN Weight Reconstruction. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6, San Francisco, CA, USA, July 2020. IEEE.

[55] Qixiu Li, Yaobo Liang, Zeyu Wang, Lin Luo, Xi Chen, Mozheng Liao, Fangyun Wei, Yu Deng, Sicheng Xu, Yizhong Zhang, et al. Cogact: A foundational vision-language-action model for synergizing cognition and action in robotic manipulation. *arXiv preprint arXiv:2411.19650*, 2024.

[56] Fangzhou Liao, Ming Liang, Yinpeng Dong, Tianyu Pang, Jun Zhu, and Xiaolin Hu. Defense against adversarial attacks using high-level representation guided denoiser. *CoRR*, abs/1712.02976, 2017.

[57] Fabiano Libano, Brittany Wilson, J Anderson, Michael J Wirthlin, Carlo Cazzaniga, Christopher Frost, and Paolo Rech. Selective hardening for neural networks in fpgas. *IEEE Transactions on Nuclear Science*, 66(1):216–222, 2018.

[58] Liang Liu, Yanan Guo, Yueqiang Cheng, Youtao Zhang, and Jun Yang. Generating Robust DNN With Resistance to Bit-Flip Based Adversarial Weight Attack. *IEEE Transactions on Computers*, 72(2):401–413, February 2023.

[59] Songming Liu, Lingxuan Wu, Bangguo Li, Hengkai Tan, Huayu Chen, Zhengyi Wang, Ke Xu, Hang Su, and Jun Zhu. Rdt-1b: a diffusion foundation model for bimanual manipulation. *arXiv preprint arXiv:2410.07864*, 2024.

[60] Robert E Lyons and Wouter Vanderkulk. The use of triple-modular redundancy to improve computer reliability. *IBM journal of research and development*, 6(2):200–209, 1962.

[61] Dongning Ma, Fan Fred Lin, Alban Desmaison, Joel Coburn, Daniel Moore, Sriram Sankar, and Xun Jiao. Dr. DNA: combating silent data corruptions in deep learning using distribution of neuron activations. In Rajiv Gupta, Nael B. Abu-Ghazaleh, Madan Musuvathi, and Dan Tsafrir, editors, *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3, ASPLOS 2024, La Jolla, CA, USA, 27 April 2024- 1 May 2024*, pages 239–252. ACM, 2024.

[62] Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. Towards deep learning models resistant to adversarial attacks. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net, 2018.

[63] Naman Maheshwari, Nicholas Malaya, Scott Moe, Jaydeep P. Kulkarni, and Sudhanva Gurumurthi. An estimator for the sensitivity to perturbations of deep neural networks, 2023.

[64] Vinod Nair and Geoffrey E. Hinton. Rectified linear units improve restricted boltzmann machines. In Johannes Fürnkranz and Thorsten Joachims, editors, *Proceedings of the 27th International Conference on Machine Learning (ICML-10), June 21-24, 2010, Haifa, Israel*, pages 807–814. Omnipress, 2010.

[65] NVIDIA. Computing the probability of ecc errors on a gtx gpu - cuda / cuda programming and performance - nvidia developer forums. https://forums.developer.nvidia.com/t/computing-the-probability-of-ecc-errors-on-a-gtx-gpu/40938, January 2016. (Accessed on 04/17/2024).

[66] NVIDIA. Dram error-correcting code memory on the platform. https://docs.nvidia.com/drive/drive_os_5.1.6.1L/nvvib_docs/index.html, November 2019. (Accessed on 04/17/2024).

[67] NVIDIA. Functional safety island (fsi) | nvidia docs, December 2023. [Online; accessed 2025-01-02].

[68] NVIDIA. 1. overview — nvidia gpu memory error management r555 documentation, May 2024. [Online; accessed 2025-01-12].

[69] NVIDIA. Cuda binary utilities, November 2024. [Online; accessed 2025-01-13].

[70] Ataberk Olgun, Yahya Can Tugrul, Nisa Bostanci, Ismail Emir Yuksel, Haocong Luo, Steve Rhyner, Abdullah Giray Yaglikci, Geraldo F. Oliveira, and Onur Mutlu. ABACuS: All-Bank Activation Counters for Scalable and Low Overhead RowHammer Mitigation, December 2023. arXiv:2310.09977 [cs].

[71] Ashish Pandharipande, Chih-Hong Cheng, Justin Dauwels, Sevgi Z Gurbuz, Javier Ibanez-Guzman, Guofa Li, Andrea Piazzoni, Pu Wang, and Avik Santra. Sensing and machine learning for automotive perception: A review. *IEEE Sensors Journal*, 23(11):11097–11115, 2023.

[72] Jonathan Peck, Joris Roels, Bart Goossens, and Yvan Saeys. Lower bounds on the robustness to adversarial perturbations. *Advances in Neural Information Processing Systems*, 30, 2017.

[73] Adnan Siraj Rakin, Zhezhi He, and Deliang Fan. Bit-flip attack: Crushing neural network with progressive bit search. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 1211–1220, 2019.

[74] Adnan Siraj Rakin, Zhezhi He, and Deliang Fan. Tbt: Targeted neural network attack with bit trojan. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 13198–13207, 2020.

[75] Adnan Siraj Rakin, Zhezhi He, Jingtao Li, Fan Yao, Chaitali Chakrabarti, and Deliang Fan. T-BFA: Targeted Bit-Flip Adversarial Weight Attack, January 2021. arXiv:2007.12336 [cs, stat].

[76] Adnan Siraj Rakin, Li Yang, Jingtao Li, Fan Yao, Chaitali Chakrabarti, Yu Cao, Jae-sun Seo, and Deliang Fan. RA-BNN: Constructing Robust & Accurate Binary Neural Network to Simultaneously Defend Adversarial Bit-Flip Attack and Improve Accuracy, March 2021. arXiv:2103.13813 [cs, eess].

[77] Gururaj Saileshwar, Bolin Wang, Moinuddin Qureshi, and Prashant J. Nair. Randomized row-swap: mitigating Row Hammer by breaking spatial correlation between aggressor and victim rows. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1056–1069, Lausanne Switzerland, February 2022. ACM.

[78] André Schaller, Wenjie Xiong, Nikolaos Athanasios Anagnostopoulos, Muhammad Umair Saleem, Sebastian Gabmeyer, Stefan Katzenbeisser, and Jakub Szefer. Intrinsic rowhammer pufs: Leveraging the rowhammer effect for improved security. *CoRR*, abs/1902.04444, 2019.

[79] Christoph Schorn, Andre Guntoro, and Gerd Ascheid. Accurate neuron resilience prediction for a flexible reliability management in neural network accelerators. In

2018 Design, Automation & Test in Europe Conference & Exhibition (DATE), pages 979–984. IEEE, 2018.

[80] Bianca Schroeder, Eduardo Pinheiro, and Wolf-Dietrich Weber. Dram errors in the wild: a large-scale field study. *ACM SIGMETRICS Performance Evaluation Review*, 37(1):193–204, 2009.

[81] Seyed Mohammad Seyedzadeh, Alex K. Jones, and Rami Melhem. Counter-Based Tree Structure for Row Hammering Mitigation in DRAM. *IEEE Computer Architecture Letters*, 16(1):18–21, January 2017.

[82] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of go without human knowledge. *nature*, 550(7676):354–359, 2017.

[83] Yusuke Tsuzuku, Issei Sato, and Masashi Sugiyama. Lipschitz-margin training: Scalable certification of perturbation invariance for deep neural networks. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018.

[84] Victor van der Veen, Yanick Fratantonio, Martina Lindorfer, Daniel Gruss, Clémentine Maurice, Giovanni Vigna, Herbert Bos, Kaveh Razavi, and Cristiano Giuffrida. Drammer: Deterministic rowhammer attacks on mobile platforms. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 1675–1689. ACM, 2016.

[85] Victor Van der Veen, Martina Lindorfer, Yanick Fratantonio, Harikrishnan Padmanabha Pillai, Giovanni Vigna, Christopher Kruegel, Herbert Bos, and Kaveh Razavi. Guardion: Practical mitigation of dma-based rowhammer attacks on arm. In *Detection of Intrusions and Malware, and Vulnerability Assessment: 15th International Conference, DIMVA 2018, Saclay, France, June 28–29, 2018, Proceedings 15*, pages 92–113. Springer, 2018.

[86] Zishen Wan, Nandhini Chandramoorthy, Karthik Swaminathan, Pin-Yu Chen, Kshitij Bhardwaj, Vijay Janapa Reddi, and Arijit Raychowdhury. Mulberry: Enabling bit-error robustness for energy-efficient multi-agent autonomous systems. In Rajiv Gupta, Nael B. Abu-Ghazaleh, Madan Musuvathi, and Dan Tsafrir, editors, *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2024, La Jolla, CA, USA, 27 April 2024- 1 May 2024*, pages 746–762. ACM, 2024.

[87] Haoda Wang, Steven Myint, Vandi Verma, Yonatan Winetraub, Junfeng Yang, and Asaf Cidon. Mars attacks!: Software protection against space radiation. In *Proceedings of the 22nd ACM Workshop on Hot Topics in Networks, HotNets 2023, Cambridge, MA, USA, November 28-29, 2023*, pages 245–253. ACM, 2023.

[88] Jialai Wang, Ziyuan Zhang, Meiqi Wang, Han Qiu, Tianwei Zhang, Qi Li, Zongpeng Li, Tao Wei, and Chao Zhang. Aegis: Mitigating targeted bit-flip attacks against deep neural networks. pages 2329–2346, 2023.

[89] Meiqi Wang, Han Qiu, Longnv Xu, Di Wang, Yuanjie Li, Tianwei Zhang, Jun Liu, and Hewu Li. A case for application-aware space radiation tolerance in orbital computing. *arXiv preprint arXiv:2407.11853*, 2024.

[90] Shaobu Wang, Guangyan Zhang, Junyu Wei, Yang Wang, Jiesheng Wu, and Qingchao Luo. Understanding silent data corruptions in a large production cpu population. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 216–230, 2023.

[91] Minbok Wi, Jaehyun Park, Seoyoung Ko, Michael Jaemin Kim, Nam Sung Kim, Eojin Lee, and Jung Ho Ahn. SHADOW: Preventing Row Hammer in DRAM with Intra-Subarray Row Shuffling. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 333–346, Montreal, QC, Canada, February 2023. IEEE.

[92] Steven C. Woo, Wendy Elsasser, Mike Hamburg, Eric Linstadt, Michael R. Miller, Taeksang Song, and James Tringali. Rampart: Rowhammer mitigation and repair for server memory systems. October 2023.

[93] Gui-Song Xia, Jingwen Hu, Fan Hu, Baoguang Shi, Xiang Bai, Yanfei Zhong, Liangpei Zhang, and Xiaoqiang Lu. Aid: A benchmark data set for performance evaluation of aerial scene classification. *IEEE Transactions on Geoscience and Remote Sensing*, 55(7):3965–3981, July 2017.

[94] Weilin Xu, David Evans, and Yanjun Qi. Feature squeezing: Detecting adversarial examples in deep neural networks. *CoRR*, abs/1704.01155, 2017.

[95] A. Giray Yaglikci, Minesh Patel, Jeremie S. Kim, Roknoddin Azizi, Ataberk Olgun, Lois Orosa, Hasan Hassan, Jisung Park, Konstantinos Kanellopoulos, Taha Shahroodi, Saugata Ghose, and Onur Mutlu. BlockHammer: Preventing RowHammer at Low Cost by

Blacklisting Rapidly-Accessed DRAM Rows. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 345–358, Seoul, Korea (South), February 2021. IEEE.

[96] Shaza Zeitouni, David Gens, and Ahmad-Reza Sadeghi. It's hammer time: how to attack (rowhammer-based) dram-pufs. In *Proceedings of the 55th Annual Design Automation Conference, DAC 2018, San Francisco, CA, USA, June 24-29, 2018*, pages 65:1–65:6. ACM, 2018.

[97] Zhi Zhang, Wei He, Yueqiang Cheng, Wenhao Wang, Yansong Gao, Minghua Wang, Kang Li, Surya Nepal, and Yang Xiang. Bitmine: An end-to-end tool for detecting rowhammer vulnerability. *IEEE Trans. Inf. Forensics Secur.*, 16:5167–5181, 2021.

[98] Lei Zhao, Youtao Zhang, and Jun Yang. AEP: an error-bearing neural network accelerator for energy efficiency and model protection. In Sri Parameswaran, editor, *2017 IEEE/ACM International Conference on Computer-Aided Design, ICCAD 2017, Irvine, CA, USA, November 13-16, 2017*, pages 765–771. IEEE, 2017.

[99] Ranyang Zhou, Sabbir Ahmed, Adnan Siraj Rakin, and Shaahin Angizi. DNN-Defender: An in-DRAM Deep Neural Network Defense Mechanism for Adversarial Weight Attack, May 2023. arXiv:2305.08034 [cs].

[100] Ranyang Zhou, Sabbir Ahmed, Arman Roohi, Adnan Siraj Rakin, and Shaahin Angizi. DRAM-Locker: A General-Purpose DRAM Protection Mechanism against Adversarial DNN Weight Attacks, December 2023. arXiv:2312.09027 [cs].

Appendix

A Artifact Appendix

This artifact provides the source code of SAVE, a detailed readme, and scripts to reproduce the main experimental results of the USENIX ATC 2025 paper—"SAVE: Software-Implemented Fault Tolerance for Model Inference against GPU Memory Bit Flips" by Wenxin Zheng, Bin Xu, Jinyu Gu, Haibo Chen. SAVE is a software-based solution that enhances the reliability of GPU-based model inference. We provide instructions to build the software package and run experiments. Our artifact obtained the "Artifacts Available", "Artifacts Functional" and "Results Reproduced" badges from the Artifact Evaluation process of USENIX ATC 2025.

Artifact repository. All project source code, along with comprehensive instructions for building and running the main experiments on SAVE, is available in the following git repository: https://github.com/peterzheng98/gpu-memory.

B Example Code Snippet for Figure 6

```python
import struct
from typing import Callable, Tuple, List

class Interval:
    def __init__(self, lo: float, hi: float):
        self.lo, self.hi = min(lo, hi), max(lo, hi)

    # affine
    def affine(self, w: float, b: float = 0.0) -> "Interval":
        lo, hi = w * self.lo + b, w * self.hi + b
        return Interval(min(lo, hi), max(lo, hi))

    # ReLU
    def relu(self) -> "Interval":
        return Interval(max(self.lo, 0.0), max(self.hi, 0.0))

    # element-wise +
    def __add__(self, other: "Interval") -> "Interval":
        return Interval(self.lo + other.lo, self.hi + other.hi)

    def __repr__(self) -> str:
        return f"[{self.lo:.6g}, {self.hi:.6g}]"

def _float_to_bits(f: float) -> str:
    return "".join(f"{b:08b}" for b in struct.pack(">f", f))

def robust_bits(interval: Interval, dtype_bits: int = 32):
    if interval.lo == interval.hi:
        bits = _float_to_bits(interval.lo)[:dtype_bits]
        return dtype_bits - 1, bits, bits
    lo_bits, hi_bits = _float_to_bits(interval.lo), _float_to_bits(interval.hi)
    robust = sum(b1 == b2 for b1, b2 in zip(lo_bits, hi_bits))
    return robust, lo_bits, hi_bits

# x in [0, 1]
x = Interval(0.0, 1.0)
op_out = x.affine(-0.65, -0.1)

relu_out = op_out.relu()
sum_out  = relu_out + x        # out = ReLU(Op(x)) + x

for name, iv in [("Op(x)", op_out), ("ReLU", relu_out), ("Final", sum_out)]:
    n, lo_b, hi_b = robust_bits(iv)
```

C Example Code Snippet for range verification of GPU kernel

```
__global__ void Ranging(const float *input,
                const float *cache_low,
                const float *cache_high,
                int *output) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    output[idx] = signbit(input[idx] - cache_low[idx])
                | signbit(cache_high[idx] - input[idx]);
}
```

D Example Code Snippet for mixed precision verification

```
float x, y;
uint32_t x_bits = *(uint32_t*)&x;
uint32_t y_bits = *(uint32_t*)&y;
// Extract the bit of x
uint32_t S_x = (x_bits >> 31) & 0x1;
uint32_t E_x = (x_bits >> 23) & 0xFF;
uint32_t F_x = x_bits & 0x7FFFFF;
// Extract the bit of y
uint32_t S_y = (y_bits >> 31) & 0x1;
uint32_t E_y = (y_bits >> 23) & 0xFF;
uint32_t F_y = y_bits & 0x7FFFFF;
// If it is smaller than 2^(-20),
// It is smaller than 1e-6, it is all robust bits.
if (E_x <= 108) return 0.0f;
if (E_y <= 108) return 0.0f;
// Compute the exponent of the result
int32_t E_r = (int32_t)(E_x) +
                (int32_t)(E_y) - 127;
// Here value must greater than 2^(-20)
// Since both E_x and E_y are greater than 2^(-20)
// Compute the sign of the result
uint32_t S_r = S_x ^ S_y;
// Reconstruct the full significand bits
uint8_t M_x = ((1U << 23) | F_x) >> ROBUST_BIT_COUNT;
uint8_t M_y = ((1U << 23) | F_y) >> ROBUST_BIT_COUNT;

// Multiply the significand
uint16_t M_r = M_x * M_y;
```

Identifying and Analyzing Pitfalls in GNN Systems

Yidong Gong, Arnab Kanti Tarafder, Saima Afrin, and Pradeep Kumar

William & Mary

Abstract

Papers on recent graph neural network (GNN) systems have established a clear trend of not showing training accuracy results, and directly or indirectly relying on smaller datasets for evaluations majorly. Our in-depth analysis shows that the omission of accuracy results leads to a chain of pitfalls in the system design, implementation, framework integration, and evaluation process, questioning the practicality of many of the proposed system optimizations, and affecting conclusions, lessons learned. We analyze many GNN systems and show the fundamental impact of these pitfalls. We further develop hypotheses, recommendations, and evaluation methodologies, and provide future directions. Finally, a new prototype, GRAPHPY, is developed to show the quantitative impact of the pitfall and establish baseline memory consumption and runtime information for GNN training. GRAPHPY also establishes a new line of optimizations rooted in solving the system-design pitfalls efficiently and practically that can be productively integrated into prior works.

1 Introduction

In today's data-driven applications, deep learning (DL) has gained prominence. Within this ecosystem, many real-world data can be stored as *sparse matrices* or *graphs*. To this end, graph neural network (GNN) models, e.g., GCN [33], GAT [49], GIN [58], GraphSage [20], and others [5, 67] are playing an increasingly important role. System optimizations play a critical role in improving the run-time, which has become exceedingly time-consuming even when using powerful accelerators, such as GPUs. There are several *semantic-preserving* GNN systems, which keep the same underlying GNN model, have proposed numerous system-level optimizations. These works can achieve up to a 15× speedup in training runtime on a single GPU [9, 14, 16, 26, 27, 44, 54, 56, 57, 64, 66].

In this work, we analyze such system-level optimizations from the prospect of the two most important aspects of GNN system design, implementation, and evaluation as listed next.

1) Unique need for GNN computation. *Forward computation* calls *kernels* (operations) from input layer to output layer, deriving prediction values. They are compared with ground truths to derive the loss. *Backward computation* uses the loss and invokes kernels from the output layer to the input layer to compute gradients and update model parameters. It accesses *state-tensors*, which hold results that are produced in forward computation (maybe *sparse*) and may require a *transpose* (Fig. 1a). Forward and backward computations have many design trade-offs (studied in this paper) that require careful navigation in system building, optimization, and evaluation.

2) Role of framework. The framework helps invoke and manage the forward and backward computation through *computation graph* that helps in invoking the right computation operation (kernel), and manages the memory. Hence, the framework code represents the overhead, while the kernel represents the actual usage of the computation device, e.g., GPU. So, a custom GNN system is likely to have a different overhead apart from different kernel performance.

We argue that the above aspects have not been understood well, leading to many *system design, implementation, and evaluation pitfalls* (Fig. 2). This paper, identifies these pitfalls, still unknown to the community, when building a GNN system in which individual kernel-level optimizations and the framework both play significant roles. The in-depth systematic analysis is backed by re-interpreting existing evaluation results, performing new experiments using additional methodologies, code-study, and a comprehensive prevalence study.

To this end, we make three main contributions:

a) Introducing Pitfalls. Majority of single-GPU GNN systems have established a clear trend of *not measuring training accuracy* [14, 24, 26, 27, 29, 32, 35, 52, 54, 56, 57, 60, 64–66], while a few [14, 27] do not even implement backward computation. Our measurements show *abnormal accuracy* in many GNN systems published in the top-tier systems conferences. Our investigation confirms *system design pitfalls* and numerous *significant implementation issues* in those systems.

Further, our *novel analysis on framework-runtime overhead* reveals that training speedup on smaller datasets is primarily due to low framework overhead and not by individual kernel and system optimizations (claimed). Unfortunately, many prior works concluded better training time largely or exclusively using smaller datasets as they report frequent out-of-memory (OOM) in popular baselines of PyTorch-Geometric [12] (PyG) and DGL [53] during peer comparison. Moreover, mini-batching and sampling-based GNNs [20] always produce smaller graphs for training.

Lastly, *memory consumption measurement by our methodology* shows that DGL, an almost universally and many times the only used baseline, suffers from *framework-memory overhead* pitfall. Its internal memory management leads to around

13 GB of additional GPU memory usage in a well-known open-source dataset, which is majorly responsible for its huge memory consumption and OOM. This has allowed memory-saving works to report highly inflated memory saving by using DGL as the only baseline; we question their analysis of the baseline and reported gain in memory saving.

We note that these pitfalls are not measurement oversight or software bugs as they create serious flaws in requirement understanding, system design, implementation, and evaluation, solving which reinforces the need to correctness tests alongside performance results, and for detailed analysis across the stack to identify changes in resources use in the evaluation process and critical system design thinking.

b) Impact Analysis and Hypothesis. We aim to increase awareness of the existence of these flaws by answering: *why* do such pitfalls occur despite the best intentions of the researchers and community (*Hypothesis*); and *what* are the major implications of these pitfalls in current GNN systems (*Impact*). They are important as these pitfalls have led to a compromised system design and implementation, misunderstanding of design trade-offs, inflated performance gain, and affecting other aspects of GNN workflow: wrong conclusions, improper lessons learned, and hindering the adoption of these research advances. Such pitfalls also *stifle innovations* as a correct system is unlikely to outperform such works.

c) Recommendation, Future Direction, and a Reference System. We address *how* to ensure future works do not suffer from similar pitfalls (*Recommendation*) and outline directions to further tackle these issues, specifically framework overhead establishes a new area of research and draws an analogy to Operating Systems research for future direction.

To help make these recommendations concrete, we introduce GRAPHPY, a prototype GNN system that: a) quantitatively shows the impact of these pitfalls, demonstrating the significance of identifying these issues and highlighting the extent of overstated results they cause; b) establishes baseline memory consumption and runtime to advance future research by protecting them from inflated gains; c) offers *practical* design choices to address a few pitfalls, which can be productively integrated into prior works.

The proposed system designs enable *data-locality* along with correct *exploitation of dataset symmetry* that can be used by almost all GNN systems, thereby GRAPHPY opens a new territory of system optimizations rooted in solving system design pitfalls. This simple design establishes many components of GRAPHPY as the state-of-the-art design.

Evaluations are done throughout the paper to show that pitfalls exist. They show that pitfalls in these systems lead to not fully representative performance results, and many are even slower than the same baseline on mid-size datasets (§7). GRAPHPY reduces memory consumption on average by 6.92×, 3.4×, and 1.96× and achieves on average 1.69×, 1.22×, 2.20× training runtime speedup, respectively, for GCN, GIN, and GAT-1 models over DGL. Further, GRAPHPY can train GCN on a billion-edge graph on a GPU due to its memory-efficient design and removal of framework-memory pitfall, while DGL cannot train even a 500 million-edge graph, highlighting inefficiencies in existing baselines.

Using GRAPHPY as a baseline against a recent work of dgNN [66], a kernel-fusion based system, shows that the latter is slower by 1.48× on training time while providing only 6.4% average memory saving for GAT. This brings critical discussion on trade-offs in kernel-fusion, so far not known to the community.

The remainder of the paper is organized as follows. §2 presents the prevalence analysis and scope, while §3 provides background and an overview. §4 and §5 discuss two categories of pitfalls, §6 outlines future directions and introduces our proposed prototype, GRAPHPY, while evaluations are done in §7. §8 reviews related works on similar papers that present pitfalls in other domains, and we finally conclude in §9.

2 Prevalence Analysis and Scope

The paper focuses on *single-GPU GNN systems* by analyzing over 20 systems [4, 9, 12, 14, 25–27, 29, 32, 35, 47, 50, 53, 54, 56, 57, 59, 60, 64, 66] in depth in the last 5 years. Most of these are GNN training systems and were published in top systems conferences, such as OSDI, ASPLOS, ATC, MLSys, EuroSys, SC, IPDPS, HPDC, PPoPP, etc., to avoid being termed as cherry-picking. We note the following important points:

- The paper is not related to reproduction issues. *We indeed can successfully reproduce similar results using their artifact description and evaluation (AE) documents. We sincerely appreciate the authors' efforts in providing such comprehensive documentation and their help through emails; it would have been impossible to build this work without them.*
- The paper is based on our interpretation of published results, which we reproduced, and additional evaluations that we performed on accuracy, framework overhead, and kernel run-time through newer methods. We confirm the observations through a code study. It may be possible that our extensive search may have missed a paper which do not suffer from pitfalls. However, that does not solve the case that the studied pitfalls are prevalent. It is essential to note that not all GNN papers succumb to every pitfall, and the paper does not seek to cast blame on all techniques proposed by prior works.
- We have discussed with the authors of GNN systems having pitfalls over email/teleconference/face-to-face to avoid misunderstandings or potential conflicts regarding the paper. Overall, we received positive and encouraging feedback. Supplementary material (appendix) shows code analysis and discussion, while we open-sourced GRAPHPY and other artifacts for healthy discussions[1].
- Single-GPU systems are *foundational* because their performance and memory consumption form the basis for

[1] https://github.com/the-data-lab would contain the open-source material

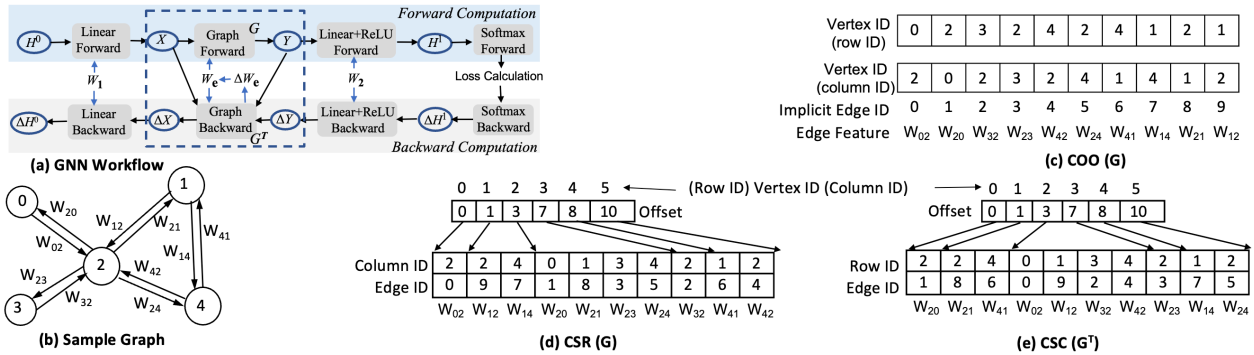


Fig. 1: (a) A GNN layer: X, Y, and W_e (edge-level tensor, e.g., attention score in GAT) are *state tensors*, which are generated in the forward pass and then retained for use in backward computation. G is a sparse matrix, and G^T its transpose. (b) Sample graph/sparse matrix (c, d, and e) CSR, CSC, and COO formats along with edge ID indirection used in DGL where consecutive edge ID is used in COO, so CSR and CSC store explicit edge ID arrays (discussed in §5.2).

other settings. Additionally, with the increase in memory size in modern GPUs and the advent of numerous memory-saving techniques, such as lower precision and quantization [8, 19, 23, 28], gradient check-pointing and/or re-computation [66], kernel fusion [57, 66], better mini-batching plus sampling [50] have allowed a GPU to train larger datasets. Hence, our analysis and corresponding evaluation methodologies emphasize single-GPU systems to establish the right foundation and methods first. We then briefly extend one pitfall of framework runtime overhead to multi-GPU systems [6, 7, 15, 30, 40, 46, 48, 52, 52, 55, 61, 62, 68, 69].

3 Background and Overview

Storage Formats. In a graph G = (V, E), V and E refer to the vertex/row set and the edges/non-zero elements (NZE), respectively. We continue to use both the graph and sparse linear algebra terminologies. Specifically, features and computation are referred to as vertex-level and edge-level, while rows, columns, and non-zero elements refer to datasets. Fig. 1 shows a sample graph and its storage formats.

The *coordinate list (COO) format* stores two arrays of rows and columns where a row and column pair indicates one NZE (or edge) of the graph. Kindly note its implicit edge ID array in the figure, which shows the format is not sorted by row ID. DGL uses COO in this way. The *compressed sparse row* (CSR) format stores NZE in a row sequentially and uses the offset array to point to the start of the row. For a directed graph, its transpose is also stored, called *compressed sparse column* (CSC) format, which stores the columns consecutively. The *degree* of a row is the row length. The CSR and CSC formats also show edge ID arrays that DGL introduces due to its specific COO format layout, and are discussed later.

3.1 Taxonomy on GNN Kernels and Models

Sparse Kernels: The sparse kernels used in GNN models can be broadly classified into two types [53]: **SpMM** (sparse matrix dense matrix multiplication) and **SDDMM** (sampled dense dense matrix multiplication) and their variants.

We first focus on two SpMM ($Y \leftarrow AX$) variants, namely, weighted (**SpMMve**) and unweighted (**SpMMv**) versions. In the former, edges contain a weight requiring an *edge-level tensor* (W_e) of size $|E|$, which can only be interpreted using the graph topology. The unweighted version assumes 1.0 as edge weight, so storing W_e is unnecessary. Kindly note that X and Y are *vertex-level tensors* as their size is $|V| \times |F|$, where $|F|$ is feature-length of each vertex.

Backward Computation and Transpose of Sparse Matrix. For SpMMve ($Y \leftarrow AX$), its backward computation needs to include both *transposed SpMMve* ($\delta X \leftarrow A^T \delta Y$) and SDDMM ($\delta W_e \leftarrow A \odot (X^T \triangle Y)$), where δ represents gradients of the tensors. We denote transposed SpMMve as $SpMMve^T$. The sparse matrix (A) itself can be viewed as two parts: the static part(the graph topology) and the dynamic part(the edge-level state tensor W_e). More details are in §4.1.2.

SDDMM kernel ($\delta W_e \leftarrow A \odot (X^T \triangle Y)$) generates an edge-level tensor (δW_e) from two vertex-level tensors read using graph topology to perform dot product and their reduction.

The *degree-norm* is a graph-based normalization kernel that divides vertex-level tensors by the corresponding vertex degrees to stabilize GNN training and mitigate the influence of high-degree nodes. GNN systems usually implement degree-norm along with SpMM and SDDMM.

Two Classes of GNN Models: In this work, we broadly divide the GNN models into two classes for their in-depth analysis.

- **Class A (Attention-Based GNNs): Vertex-Level + Edge-Level Tensors.** Example models include GAT and GaAN [49, 67], etc., that learn edge-level attention scores. The attention score represents the importance of neighboring nodes during training. The edge-level tensors encode trainable attention scores derived from graph vertices using trainable parameters. As a result, this class employs SpMM (weighted) and SDDMM kernels, involving both vertex-level and edge-level tensors, making the computation more complex.
- **Class B (Non-Attention-Based GNNs): Vertex-Level Tensors Only.** This GNN class does not employ dynamic

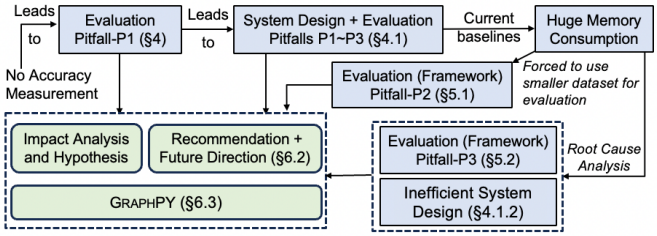


Fig. 2: The complex relationship between pitfalls and our recommendation. GRAPHPY prototype helps quantify the impact of the pitfalls while providing some practical and deployable solutions to a few pitfalls.

attention scores. Notable examples include GCN [33] and GIN [58]. These models use unweighted SpMM, relying solely on vertex-level tensors since no attention mechanism is involved. Hence, it does not employ SDDMM kernels. Further, it usually deploys degree-norm kernels. Thus, this GNN class significantly differs from class A GNNs.

3.2 Overview of Pitfalls

This paper unveils many GNN system related pitfalls. Fig. 2 shows a high-level overview of various pitfalls and their relations. It categorizes them into two main evaluation-related types, which leads to many system design and framework-level pitfalls. A brief overview is discussed next.

Accuracy Related Evaluation Pitfalls (EVAL-P1, §4). This category of pitfalls is due to not reporting the achieved training accuracy by prior works on the semantic preserved models. Our evaluations show abnormal accuracy. Prevalence studies and detailed code analyses unearth several underlying system design (SYS-*) pitfalls (§4.1.1, §4.1.2, §4.1.3, §4.1.4) that directly cause or exacerbate these evaluation inaccuracies, but end up providing runtime speedup.

Framework Related Evaluation Pitfalls (EVAL-P2 and EVAL-P3, §5). The framework refers to the *Python and C++ integration code that integrates individual computational kernels (written in CUDA) to deep learning platforms* such as PyTorch, TensorFlow, etc., where the latter manages the computation graph and memory.

When training runtime is evaluated using small datasets, the measured time primarily reflects framework-runtime overhead rather than actual time spent on the GPU. We show that different systems differ in framework overhead, while the usage of smaller datasets for evaluation is universal, such as in sampling-based GNNs that produce a smaller sampled graph to train the model on a GPU. We further observe excessive memory consumption by framework integration-related inefficiencies in DGL [53], a popular GNN baseline, which remain unknown to the community. This has caused inflated results on memory-saving techniques and often forces runtime evaluations to be conducted on small-sized datasets due to frequent out-of-memory (OOM) errors on mid-sized datasets.

After identifying these pitfalls, we present recommendations and future directions (§6.2), and propose a baseline system, called GRAPHPY (§6.3), that helps in quantitatively demonstrating the impact of these pitfalls (§7) and outlines design solutions that can be integrated in prior pitfall infested systems to overcome a few pitfalls.

4 EVAL-P1: Lack of Accuracy Measurement

Introduction and Prevalence Study. Several GNN systems have established a clear trend to not report training accuracy in their papers. [14, 24, 26, 27, 29, 32, 35, 51, 52, 54, 56, 57, 60, 64–66] Fig. 3 shows that many papers with which we experimented exhibit abnormal accuracy. DGL accuracy is for reference. All experiments are run on an Nvidia A100 GPU. Datasets and other details are presented in §7.

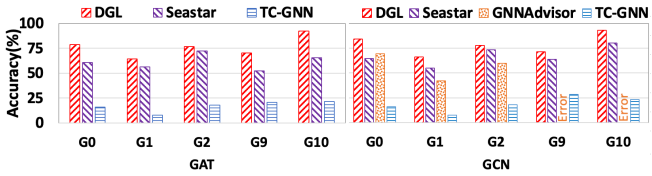


Fig. 3: Accuracy comparison: TC-GNN does not provide GAT, hence AGNN is substituted here as both are attention-based (Class A GNN). We also discuss other works in the text. DGL is for reference.

Class A GNN. Seastar [57] has 4.5% – 26.9% accuracy drop, while TC-GNN [56] has very low accuracy on all datasets for GAT. FuseGNN [9] GAT shows NaN during training on Reddit and could achieve only 58% accuracy, way below DGL's 92.4%. Huang et al. [27], and TLPGNN [14] are not measured as they do not implement backward computation. GNNAdvisor has not implemented GAT.

Class B GNN. GCN training accuracy in TC-GNN is abnormally low. Seastar accuracy drops by 4.5% - 19.8%. GNNAdvisor [54] drops by 14.8% – 24%. Also, GNNAdvisor throws a memory corruption error for Reddit and OGBN-products, which is also reported by others [14]. Huang et al and TLPGNN do not have backward computation implemented. Ge-SpMM [26], which does not report accuracy, could not be run due to major code reorganization in DGL to which it was integrated. Featgraph [25], which is integrated into DGL, we could not install it correctly despite help from its authors and trying several GitHub branches though it did report accuracy on Reddit. Except for DGL, only PyG and FuseGNN have normal accuracy on GCN.

Fundamental Problem or Evaluation Oversight? Significant accuracy drop is not an evaluation oversight, as these systems usually aim for the *semantic-preserving model*, the same model as the baseline (DGL in this case), with changes being tailored for their system optimizations. §4.1 analyzes these systems to show that the lack of accuracy evaluation has allowed many fundamental issues in design, implementation, and analysis to persist.

4.1 Misunderstood Backward Computation

Our analysis for drop in accuracy is attributed to *insufficient inclusion both of the backward computation and of its impact on the opportunities to optimize the forward pass.* Indeed, backward computations are complex as they often enforce the saving of state tensors during forward computation, introduce transpose requirements, and can be computed in an order determined by the computation order in forward computation. Thus one can find runtime performance-enhancing bugs in their wrongly designed system. We also note that correct baselines deploy inefficient designs despite the availability of efficient choices. The next few sub-sections present these system design and implementation pitfalls.

4.1.1 SYS-P1: Omitted State Tensor

Definition and Example. Many forward-computation-focused optimizations fuse kernels to optimize the performance of their forward computation, while neglecting state tensor requirements because the backward computation is not implemented. The backward computation of any operator needs to access the *state-tensor*– the input and/or output tensors of the corresponding forward operator. The requirement to materialize state tensors is not limited to sparse operations; other commonly used operations, e.g., ReLU, Dropout, etc., also require storing state tensors. Such tensors must be *saved during forward computation* for use during backward computation, or recomputed during the backward computation.

Huang et al. [27] and TLPGNN [14] prototyped one giant fused forward kernel that fused many sparse kernels along with other operations, such as ReLU, etc. TLPGNN omits the materialization of the attention scores in GAT, which are edge-level state tensors. Similarly, both skip materializing state tensors for ReLU, which are essential for correct gradient computation during backpropagation. It is important to emphasize that both works aim to optimize GNN training, not inference, and therefore are not evaluated in the context of inference performance.

Fundamental Issues. The performance gain claimed by aggressively fusing operations in a forward-only computation does not hold if one attempts to fix this pitfall by properly materializing or recomputing the state tensors. Doing so introduces additional steps that lead to performance degradation, invalidating the originally reported speedups. For example, *recomputation* of state-tensors [66] itself requires saving some other tensors as state-tensors during forward computation, which is used during backward computation to recompute the actual state-tensor(s). This saving and re-computation procedure leads to additional slowdown, meaning that the claimed speedup is not practical, even if the fused kernel optimizations are technically sound.

4.1.2 SYS-P2: Regarding Sparse Matrix Transpose

Definition and Examples. Backward SpMM computation ($\delta X \leftarrow A^T \delta Y$) requires the transpose of A. The sparse matrix (A) itself can be viewed as two parts: the static part (the graph topology) and the dynamic part (the trainable and generated edge-level state-tensor W_e). Though one can pre-process the graph to keep the transpose of the topology, the *generated* W_e requires a transpose at run-time. W_e is never symmetric, even if the sparse matrix (graph topology) is symmetric, due to the GNN model design. This sub-section discusses the absence and efficiency of transpose in prior works, as cuSPARSE already provides a high-performing native $SpMM^T$.

• *SYS-P2-1: No Transpose.* Many works [14, 27] don't implement backward computation by ignoring this problem, TC-GNN [56] substitutes forward SpMM in place of backward SpMM for its attention-based GNN. Kindly note that the popular GNN benchmarking datasets are square sparse matrices, so not performing a transpose doesn't lead to a dimension mismatch, hence it cannot be caught during compilation/runtime unless accuracy is measured. Thus, they also gain non-practical performance.

• *SYS-P2-2: Omitting comparison with a Key Baseline of cuSPARSE $SpMM^T$.* Many systems developed additional mechanisms for transpose even when relying on cuSPARSE. However, they completely ignore the fact that cuSPARSE natively offers $SpMM^T$ without performing the explicit transpose of A. Our measurements (Fig. 10) show that SpMM^T by cuSPARSE is still the fastest. Thus, this pitfall allowed many sub-optimal system designs to dominate the GNN solutions. Even worse, as we explain next, such additional mechanisms introduce additional memory consumption. We first analyze DGL for its memory-intensive SpMM^T approach, followed by alternate mechanisms. This also serves as an understudy for our reference system (§6.3).

DGL. To support SpMM^T, DGL relies on *edge ID abstraction* in graph storage. Each edge in the DGL graph is assigned a unique numerical identifier in the range of $[0, |E|)$. Its allocation is determined by its COO format (used for SDDMM implementation): the offset of each edge in COO is its implicit edge ID (see Fig. 1c). However, the order of edges in DGL's COO (i.e., storage layout) is determined by the dataset file or parsing module which is often performed by the user (external factor), and hence its COO is neither stored in CSR-style (rows are laid out consecutively) nor in CSC-style (columns are laid out consecutively). In such cases, both CSR and CSC contain an explicit array of edge IDs (Fig. 1(d) and 1(e)).

DGL introduces eShuffle, an internal GPU kernel, that rearranges the input edge-level tensors using the edge ID array of CSR and CSC to generate a new edge-level tensor for SpMMve and SpMMve^T respectively so that DGL can use the same SpMM API of cuSPARSE after eShuffle. The eShuffle kernel slows down performance by 64% of cuSPARSE SpMM on Reddit (§7) on both forward and backward, hence it is very costly. So any system integrated into DGL suffers from similar issues, but an independently developed system can get additional performance benefits if it does not perform transpose, while forward-focused systems automatically gain 64% speedup without any optimization.

Memory consumption in DGL for storage formats is $2|V|+6|E|$ due to CSR ($|V|+|E|+|E|$), CSC ($|V|+|E|+|E|$), and COO ($|2E|$) irrespective of GNN models. Due to eShuffle, DGL also consumes an additional $|E|$ memory.

PyG and MariusGNN. PyG implements SpMM and SpMM^T by materializing dot products of size $|E| \times |F|$ using GPU global memory before performing reduction in its original design of using the COO format. Hence, PyG exhibits OOM more frequently, as it cannot run on an A100 GPU with 40GB of memory, and this issue has been reported by prior works [9, 27]. However, cuSPARSE does offer an efficient SpMM and $SpMM^T$ on COO, which is not used. Lately, PyG uses CSR without materializing dot products on GCN, however, GAT (which uses SpMMve/SpMMve^T) still materializes them. MariusGNN uses a PyG-like solution using CSR format, i.e., it materializes the dot products in GPU global memory. This mechanism is slower than native cuSPARSE APIs (§7.1).

Other Mechanisms. Seastar and FeatGraph also rely on edge ID indirection for GAT in both the forward and backward computations. FuseGNN relies on *cusparseCsr2cscEx2*() API (which transposes the topology and edge-level tensor) plus their custom forward SpMM to implement SpMM^T. The cusparseCsr2cscEx2() API is slower by 130% of cuSPARSE SpMM on Reddit, thus, this mechanism is even more costly.

Fundamental Issues? Dropping the transpose requirement (A^T) in SpMM^T provides GNN systems with unfair performance gain; while introducing an inefficient solution by ignoring the key baseline of cuSPARSE leads to no progress to the state-of-the-art in basic kernel design of $SpMM^T$ (§7.1). Both of these issues are fundamental from a research perspective.

4.1.3 SYS-P3:Incorrect Order of Backward Operations

A few works that attempt kernel fusion in the forward computation do not order their backward operation correctly, which leads to better performance at the expense of wrong training.

Definition and Example. The backward operations, as the name suggests, invoke the kernels in the reverse order to the forward operations. For example, if operator A is called first followed by operator B, then the backward computation first needs to call the backward of operator B followed by the backward of operator A. However when the operators A and B are fused in forward, their backward requires a different custom fused kernel to take care of the ordering.

For example, GNNAdvisor [54] fuses the SpMM and degree-norm operators during the forward. However, its backward computation did not reverse the order. For Seastar [57] in GCN, the accuracy drop is similar to GNNAdvisor. This is because Seastar relies on a lambda function specification for fused forward SpMM and degree-norm in the forward() API of the GCN layer to generate CUDA code. The generated code always calls degree-norm after SpMM. We suspect that code-generation logic may not be intelligent enough to generate correct backward fusion.

Fundamental Issues? An independent *degree-norm* fetches the degree of every *row*, thus occurring $|V|$ data load. It can easily be fused with forward SpMM, as degree-norm usually happens on the output tensor of SpMM. However, during backward, it should be called on the input tensor of the SpMM. Hence, a correct fusion during backward computation needs to fetch the degree of column ID of every NZE, thereby performing $|E|$ data load fetch. Hence, a wrongly fused kernel gains performance due to less data load and computation.

4.1.4 SYS-Others: Additional Pitfalls

The following additional errors have been observed in GCN: no model bias parameter in GNNAdvisor, Huang, et al [27], and TLPGNN in their GCN; and absence of bias and *degree-norm* array in TC-GNN in its GCN. In DGL, the bias parameter is enabled by default, while performing degree-norm operation. In GAT, no dropout layer is there in the fused GAT model of Seastar, TLPGNN, and Huang et al., introducing which requires complex changes to their backend due to the fusion. Huang et al use the same allocated memory to keep the attention score for every model layer without recomputation. TC-GNN does not implement SpMM and SDDMM kernels with odd dimensions of feature-length needed in the last layer of GCN and AGNN models, yet its training script calls these kernels with those feature lengths. These errors indicate design and evaluation issues.

4.2 Discussion and Hypothesis

The accuracy evaluation is effort-intensive exercise due to various compilation, installation, and usage issues that are common with academic prototypes. Additional supporting software has since changed substantially, like deprecated APIs. Many systems rely on generated features and labels, while each system has its own way of loading the dataset, which takes time to understand and modify to read from the labeled dataset. Hence we limited accuracy evaluation to top-tier conferences only. Further, our goal is to be fully sure when we report abnormal accuracy. It is possible that other systems that did not show accuracy may be correct, but will not solve the pitfall as future works may continue to make erroneous decisions if accuracy is not measured.

Hypothesis: Automatic Backward Computation. We hypothesize that the complex differences between forward and backward computations in the DL frameworks' abstraction could be a reason: the model and layer definitions only explain the forward computation, while the backward code is invoked automatically for each forward operation and remains hidden from the usual code walk-through, but depends on the output of the forward pass. Hence, without correctness evaluation, it is hard to get the implementation right. Such pitfalls are independent of hardware settings, though we only analyzed the code of single-GPU GNN systems.

5 Framework Overhead + Evaluation Pitfalls

Introduction and Prevalence Study. Many prior works showed *frequent OOM on popular baselines* (DGL and PyG) thereby almost all single-GPU GNN training systems [4,9,14,24,26,32,47,47,53,54,56,57,59,63,64,66] have exclusively or majorly relied on *smaller datasets to conclude better training runtime*. E.g., GNNAdvisor, Ge-SpMM, TC-GNN, and others [4, 24] relied *exclusively* on smaller datasets for training runtime comparison. FuseGNN [9], dgNN [66], and Seastar used only one mid-size dataset(Reddit), while the remainder are smaller datasets. However, many have shown OOM on this mid-size dataset for some DGL models.

Though *sampling-based* GNN [29,31,36,37,43,50,62–64] evaluates large datasets, it should be noted that they sample only a *batch* of vertices of the dataset in each iteration to generate a smaller graph to run training/inference. Hence, *the resultant sampled graph is small on which training is run*.

Fundamental Problem or Evaluation Oversight? As discussed earlier (§3.2), framework implies Python and C++ code to glue the individual kernel to PyTorch (or TensorFlow, etc.) that runs CPU in a GPU-based system. We show (§5.1) that training runtime on smaller graphs is dominated by the framework runtime overhead in GPU-based systems. So, the training speedup in such cases is due to the lower framework overhead (unknowingly) instead of better kernel runtime (claimed). This invalidates the analysis done by prior system evaluations on this metric. Further, the framework design has become a major reason for OOM in DGL (§5.2), introducing framework memory overhead which is very large compared to memory consumption due to inefficient GNN design(§4.1.2). As DGL has been a universal baseline, it has affected almost every GNN system evaluation because prior works reported highly inflated memory saving, frequent OOM in DGL, and better training runtime on smaller datasets.

This section analyzes the role of the framework– the runtime and memory overhead– to show that we have not fully understood the fundamental role of the framework in GNN system building. To illustrate these facts, the next few subsections run new evaluations using a novel methodology and analyze the source of different frameworks in DGL versus other systems. We use an Nvidia A100 GPU. Datasets and other details are the same as in §7.

5.1 EVAL-P2: Framework Runtime Overhead

We define the *training time* as the total wall-clock time to finish the training. Kindly note that prior works do not include data transfer time, if any, as part of training time measurement in a GPU-only setting [14, 25–27, 54, 56, 57, 66].

The *framework-runtime overhead* is defined by Eq. (1): sum of the time consumed by the CPU in a GPU-only training system without counting the time spent on waiting for GPU, if any. Hence, the source of the overheads is roughly Python/C++ code execution in the CPU, including the cost of kernel launches, which is usually asynchronous to the CPU.

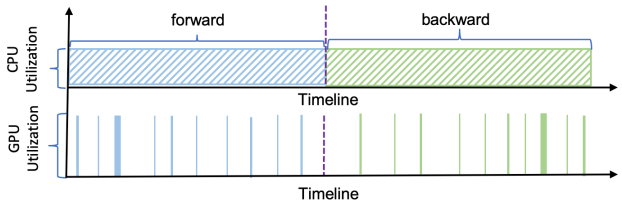


Fig. 4: Illustration of CPU and GPU activity timeline during a GCN training iteration on DGL on the Cora dataset, clearly showing the GPU idle time where framework overhead executes exclusively on CPU. This illustration is a simplified version of the profiler output.

$$Overhead = \sum_{k}(CPUTime - CPUWaitForGPUTime) \quad (1)$$

Challenges and Approach. The overhead is a problem only if it can not completely be overlapped by GPU execution, i.e., *significant* GPU idle time between kernel executions is a problem where only overhead is executing (in CPU).

A naive method to gather this information is to observe it from the timeline view of execution, e.g., using the Nsight tool, as shown in a simplified Fig. 4 for one training iteration. As shown, the CPU remains consistently active throughout the training iteration, whereas the GPU experiences frequent idle periods. This indicates that the GPU is underutilized, leading to CPU-side overheads (framework runtime overhead) dominating the overall training time. However, the process is manual and error-prone, takes a huge amount of human effort, and must be repeated for all combinations of GNN systems, models, and datasets– a gigantic task nonetheless.

In light of these observations, our approach is to collect the runtimes: training (TT), overhead (OT), and GPU execution (GT). OT(%) identifies the overhead percentage compared to training time, while 100 – GT(%) is the percentage time where overhead is executed *exclusively* and is the metric of interest.

5.1.1 Measurement and Analysis

Fig. 5 shows the overhead during DGL GCN training for different-sized graphs. **a)** Framework overhead is 100% for up to 4 million edges: reduction in framework overhead would automatically improve training runtime, while any GPU runtime improvement or slight slowdown would bear no consequence to training runtime. E.g., the GPU runtime is only

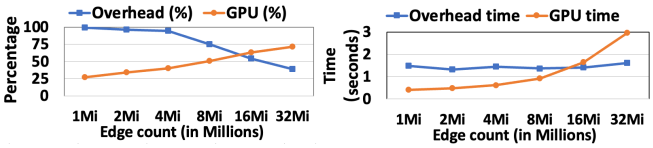


Fig. 5: DGL framework-runtime overhead when training GCN (200 iterations) on graphs ($|V|$=32,768). **Left:** framework overhead is 100% of training time when edge-count is $<$ 4 Million. **Right:** Overhead remains similar irrespective of the edge count.

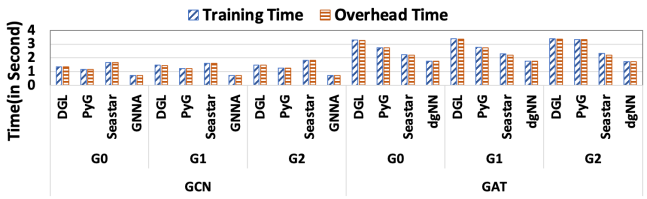


Fig. 6: Training and framework runtime overhead for DGL, PyG, GNNAdvisor (GNNA), and dgNN for GNN models training for 200 epochs. Seastar also has almost 100% overhead for these datasets.

around 25% at 1 Million edge, implying almost 75% overhead is not overlapped with GPU. The overhead (close to 100%) is almost the training time. **b)** Up to 32 million edges: idle GPU time (100 - GT(%)) remains *significant* where overhead is executed alone, thus playing a major role in training time. **c)** Only when the graph size becomes large, e.g., in Reddit (around 229 million edges, not plotted) DGL framework overhead is only 5.42% while the GPU runtime is close to 100%, resulting in almost no GPU idle time, implying no role of overhead in training time.

Fig. 6 shows that the overall training time is dominated by the framework overhead when evaluated for multiple GNN models across three widely used small datasets: Cora, Pubmed, and Citeseer; all have less than 100k edges.

Overhead varies in different systems due to Python/C++ code differences. In DGL, its generic Python-based message passing infrastructure, and conversion of PyTorch tensor to DLPack tensor to DGL-specific tensor are a few things that allow it to be extensible, supporting PyTorch, Tensorflow, MxNet, etc., but all of them are framework overheads. Systems that develop GNN from scratch hardly rely on the DGL front-end. E.g., GNNAdvisor has lowered its overhead mostly due to no implementation of message passing, removing bias parameters in its GCN layers, and using C++ to write GNN layers instead of Python to lower framework overhead.

Systems integrated into DGL, e.g., GE-SpMM [26] (not shown in Fig. 6), which replaced DGL with its sparse kernels, have not shown performance gain in their paper for GCN for smaller datasets on which it relied exclusively, despite showing better SpMM kernel time than cuSPARSE. However, it attributes this observation to the number of prediction classes not being a multiple of 32 without providing any evidence.

5.1.2 Measurement on Sampling-based GNN Systems

We now focus on a 3-layer GraphSage model with a batch size of 256 and fan-out as [10, 10, 10], which produces three sampled graphs, one for each GraphSage layer. In Reddit and OGBN-Products, our measurements show that the maximum edge count in the sampled graph is always less than 0.45 million in DGL, which is very small (see Fig. 5). Hence, further measurements indicate a close to 100% overhead for training on these two datasets. For example, the actual training procedure took around 48 seconds in the OGBN-Products dataset for 10 epochs (1 epoch has 768 iterations). The total overhead was almost 100%.

When doing the same measurement in MariusGNN [50], the training time improved to around 22 seconds but still had close to 100% overhead. Even for the paper100M dataset, one of the largest datasets used in sampled GNN training, the generated sampled graph has between 13k–2M edges [50], which is small enough as per Fig. 5.

GraphSage uses GCN for training on sampled graphs. So, we conclude that the framework overhead dominates training time in sample-based GNN systems. In other words, GPU kernels play no significant role in training time in this case.

5.1.3 Hypothesis and Discussion

GNNs are primarily impacted by framework runtime overhead because their computation cost is proportional to the batch size, as the trainable parameters are not large. E.g., in GCN and GraphSage, the trainable weights are of size $|F_{input}| \times |F_{output}|$ (plus bias), which are usually small. The computationally significant computations in these models are GeMM (matrix-matrix multiplication) and SpMM. GeMM depends on batch size and feature-length, while SpMM depends on graph size and feature-length. Both are lightweight in powerful GPUs for smaller graphs.

To this end, multi-GPU training time on sampling-based GNN is also impacted by the overhead, as we measured DGL on up to a 4-GPU single-node system using the data-parallel method. The data-parallel GNN training divides the mini-batches among GPUs while replicating the model. Partitioning mini-batches causes an even smaller mini-batch size per GPU, but replicating the model forces the replication of the same framework overhead to each GPU. This results in lesser computation per GPU, leading to large GPU idle time (100-GT(%)) where the framework overhead runs exclusively. Moreover, the network communication time is insignificant in GNN as it only communicates (in bulk) the gradients of trainable parameters, whose count is small as discussed above.

Indeed, any other model where a small batch size makes computation lightweight will likely suffer from the same framework runtime overhead. Exploring those models remains out of the scope of this paper due to the focus on GNN. A few additional examples are presented in the appendix.

More importantly, as GPU performance continues to improve, kernel execution will be faster, but the CPU overhead runtime would remain almost similar (single-threaded execution). Hence, it would lead to comparatively large framework overhead (100 - GT(%)) for the problems discussed in the paper. This trend suggests that the performance issues discussed in this paper would be even more critical in the future as increasingly powerful GPUs are adopted. Hence, we believe that relying on the specific GPU (Nvidia A100) did not significantly influence our observations.

5.2 EVAL-P3: Framework Memory Overhead

Many papers have shown frequent OOM on slightly older GPUs for mid-size datasets on PyG and DGL, which is aligned with our measurement. However, the excessive mem-

ory consumption has forced prior GNN systems to heavily or even exclusively rely on smaller datasets for comparing training runtime leading to EVAL-P1 (§5.1). We have described OOM for PyG in §4.1.2. This subsection explores the role of the framework for large memory consumption in DGL.

Observation. GCN training consumes 23.2GB of GPU memory (Fig. 16(a)) on the Reddit dataset. Our code-study in DGL GCN locates these inefficiencies: *a)* storage cost (§4.1.2); and *b)* to implement SpMMv in GCN, DGL uses a *dummy edge-level tensor* ($|E|$ additional memory) with each value as 1.0 to use cuSPARSE, which offers only SpMMve API.

However, these known memory inefficiencies aren't the primary reason, as an $|E|$ and $|V|$ memory implies only around 875 MB and 1 MB in Reddit, respectively. PyTorch memory profiler results attributed the largest memory to $torch :: empty()$, a memory allocator API, without providing any higher-level information.

Methodology. To develop insights, we evaluate memory consumption at storage format, kernel, and iteration levels. **1)** For DGL's SpMM kernel-level memory analysis, we printed the memory usage right before and after its call on Reddit ($|F|$=32) and found a massive 12,982 MB memory consumption. However, our direct evaluation of cuSPARSE SpMM shows only around 936 MB consumption. **2)** DGL SDDMM kernel consumed exactly 936 MB of memory.

A simple estimate also shows 936 MB as the actual memory consumption for both sparse kernels. SpMM includes the input tensors (vertex-level tensor and edge-level tensor: $|V| \times |F| + |E|$) and the output tensor (vertex-level tensor: $|V| \times |F|$). The scratchpad memory for cuSPARSE SpMM is tiny, while SDDMM does not need it. *Thus, we note that DGL introduces a framework memory overhead of around 12,046 MB (12,982 - 936) in SpMM, as known memory consumption cannot account for it.* Another DGL version also introduced a 6,340 MB of memory overhead in SpMM, thus confirming that it is not a version-specific problem.

DGL SpMM uses a PyTorch Python API to allocate memory for the input and output tensors of sparse kernels but SpMM scratchpad memory allocation is performed internally in the back-end that uses a different mechanism. Specifically, DGL's usage of DLPack tensor makes its back-end independent of PyTorch. So the back-end brings PyTorch through non-standard integration and uses its C/C++ API for this allocation. This non-standard integration of PyTorch is the reason for the memory overhead, as the PyTorch profiler pointed to allocation routine $torch :: empty()$ as the largest memory consumer while SDDMM has neither any internal allocation nor memory overhead.

The novelty of our method lies in the flexible granularity of measuring memory consumption to locate issues. A kernel-level granularity has shown issues with non-standard PyTorch integration. When we changed the granularity to an iteration level, we noticed a memory leak in dgNN (§7.2.3). PyTorch profiler alone is not sufficient in either case.

6 Future Direction and GraphPy

Pitfalls presented in §4 and §5 have renewed our understanding of system design and implementation for GNN models and the role of the framework respectively. Hence, the natural progression is awareness of the impact of these pitfalls on research progress and human resource development (§6.1), and overcoming them collaboratively through recommendations, future research direction (§6.2), and a prototype to quantitatively measure the impact (§6.3).

6.1 Additional Impacts

We presented the impact of the pitfalls when discussing them, including questionable or inflated speedup in runtime and memory saving. Here we present some high-level impacts.

Extensive and Wider Effect. A GNN system with lower training accuracy due to pitfalls impacts inference when deployed. Framework overhead impacts almost every sampling-based GNN system. Moreover, as DGL extends the same design to its multi-GPU setting, and is a widely used system as a baseline and even for system integration [15, 25, 26, 57], it has impacted almost every GNN system in their evaluation analysis due to the framework overheads.

Stifling Innovations. A new system with valid optimization may not outperform pitfall-infested works without committing similar or other pitfalls unless one characterizes the issues prevalent in prior works in their research paper, leaving them with less space to describe their contributions. Hence, if the community is unaware of them, genuine progress in system optimization may not happen.

Inadequate Training to Future Leaders. The discussed pitfalls indicate questionable research practices [41]; otherwise, they would not have open-sourced their code. These issues indicate that senior researchers are not dedicating enough time to review trainees' deliverables or teach them appropriate research methods [2].

6.2 Recommendation and Future Direction

Measuring training accuracy is the most important feedback for the correctness, and hence should be enforced or must be reasoned on why it is not needed (Recommendation #1).

Performance Runtime Related. Smaller datasets alone cannot be used to conclude a better runtime if the system is not specifically designed and rightly evaluated for such cases (Recommendation #2), such as by treating the framework runtime overhead as a primary metric. For non-sampled GNN, if available labeled datasets are not of mid-size or larger, we recommend relying on datasets through generation tools [17, 18] and other graph repositories [42, 45] for training runtime comparison (Recommendation #3).

Framework runtime overhead presents a new area of research, analogous to Operating Systems. E.g., when we zoom into one kernel operation, it is becoming faster due to powerful GPUs, better algorithms, and other innovations such as quantization, pruning, sampling, etc. At the same time, frame-

work overhead is like a system call overhead, which remains constant and understudied, and runs in a single CPU core. Thus, *the DL framework is more like an Operating System to allow hardware usage without introducing much overhead like the system calls*. Thus, the future lies in studying those overheads (Future Works #1).

Lastly, the systems with no backward implementation or accuracy issues risk being sidelined for evaluations by future works, as some pitfalls can only be addressed via invasive efforts. A systematic approach is a standardized system benchmarking tool. Such work remains a work in progress (Future-Work #2). Till then, *directly evaluating kernel runtime can be a stop-gap solution* (Recommendation #4) for non-sampled based GNN works, to not discard their advances in basic kernel design (SpMM and SDDMM). This is impossible if prior works have only fused kernels with pitfalls.

Memory Consumption Related. *Frequent OOM and memory consumption should be analyzed* (Recommendation #5) like runtime with the same zeal (as we do in §5.2) without any pre-assumption while attributing memory saving quantitatively. Otherwise, we will continue to have highly inflated results. For GNN systems with non-standard integration, additional verification of memory consumption is needed.

6.3 GRAPHPY Design

Quantitative demonstration of the impacts of the pitfalls remains an open challenge, specifically to claim that the progress of the state-of-the-art has been compromised as they have allowed inefficient design to be declared better. Such analysis requires a new prototype where various pitfall-infested design choices could be evaluated quantitatively using ablation studies and controlled experiments. This motivates GRAPHPY, the proposed prototype.

By design, GRAPHPY mostly imitates DGL and manually fixes pitfalls by removing DGL's: *a)* framework-memory overhead, *a)* message passing, *c)* inefficient design of using eShuffle in forward computation, and *d)* avoiding dummy edge-level tensor in SpMMv. The design choices let us evaluate framework-level and GNN system design pitfalls while establishing baseline memory consumption as discussed next.

6.3.1 Framework-Level Design

GRAPHPY removes *framework-memory overhead* by following PyTorch plugin documentation rather than using the DGL style of non-standard PyTorch integration (§5.2). Hence, all memories are allocated using Pytorch APIs imported using the standard integration.

Removal of message passing is achieved by directly relying on basic sparse kernels of SpMM and SDDMM, as both are equivalent. Message passing APIs are mapped to sparse kernels anyway by DGL. Hence, GRAPHPY does not lose anything from it but minimizes the framework overhead. Hence, when we keep the same kernel as the one used by DGL, it allows us to evaluate framework-level pitfalls.

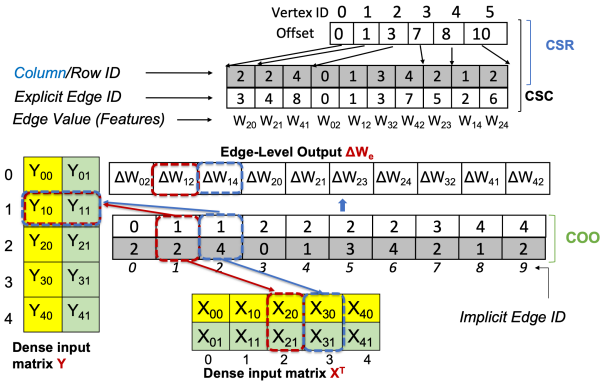


Fig. 7: GRAPHPY storage format for sample graph (Fig. 1b): The CSR format is assigned the implicit edge IDs. The COO is then generated from it, which enables caching of Y data in SDDMM. There is only one copy of the grayed array representing the Column ID array, which is shared among COO, CSR, and CSC. The Offset array is shared between CSR and CSC. The edge ID array is for CSC only.

6.3.2 GNN Kernel Design

In the forward computation, GRAPHPY substitutes DGL's eShuffle+SpMM with only SpMM. The removal of eShuffle is achieved by fine-tuning the storage format: GRAPHPY assigns an implicit edge ID to its CSR format by treating edge position as edge ID. In contrast, DGL assigns an implicit edge ID to its randomly ordered COO (§4.1.2). An implicit edge ID array in CSR, as shown in Fig. 7, makes the eShuffle in forward computation a no-op, hence, it is removed. Kindly note that CSC needs an explicit edge ID array, as shown in the figure. However, both share the storage structure due to dataset symmetry, which is achieved by storing the graph edges in both directions as an enrichment process, which almost all prior GNN systems do.

GRAPHPY then generates COO from the CSR (CSR-way): edges are stored in row-major as shown in Fig. 7. In comparison, DGL generated CSR from the randomly ordered COO. Our design choice allows the column ID array to be shared between the CSR and COO, while the COO also gets an implicit edge ID array.

In the backward computation, GRAPHPY implements $SpMM^T$ either as eShuffle+SpMM or using cuSPARSE-native SpMM^T. We implement both for comparison. However, we implement eShuffle+SpMM as one kernel because eShuffle is an intermediate kernel whose output is used by SpMM only. This saves the memory of eShuffle output ($|E|$). Thus, it becomes almost a drop-in replacement for cuSPARSE-native SpMM^T and is not a GNN kernel fusion technique.

Another advantage of eShuffle removal from forward is an automatic data-locality/caching enabled in SDDMM (on CSR-way COO), which is implemented as edge-parallel as in DGL. For example, when a warp processes two edges, such as (0,5) and (0,10), the features of source vertex 0, fetched during the first edge, can be reused when processing the latter,

Table 1: Dataset details. * indicates labeled dataset. Others use 150 generated features ($|F|$) and 7 prediction classes ($|C|$).

Graph Dataset Name	Vertex Count	Edge Count	$\|F\|$	$\|C\|$
Cora(G0)*	2,708	10,858	1,433	7
Citeseer(G1)*	3,327	9,104	3,703	6
Pubmed(G2)*	19,717	88,648	500	3
Amazon(G3)	400,727	6,400,880	150	7
As-Skitter(G4)	1,696,415	22,190,596	150	7
Cit-Patent(G5)	3,774,768	33,037,894	150	7
Stackoverflow(G6)	2,601,977	95,806,532	150	7
Hollywood(G7)	1,069,127	112,613,308	150	7
LiveJournal(G8)	4,847,571	137,987,546	150	7
OGBN-Products(G9)*	2,449,029	123,718,280	100	47
Reddit(G10)*	232,965	229,231,784	602	41
Orkut(G11)	3,072,627	234,370,166	150	7
UK-2002(G12)	18,520,486	596,227,524	150	7
Kron-25(G13)	33,554,432	1,073,741,824	150	7

as it would automatically be cached in the hardware cache. DGL's randomly ordered COO has no such advantage.

SpMMv is implemented using cuSPARSE SpMMve, which uses a dummy edge-level tensor like DGL, while we also provide a native SpMMv that does not use a dummy edge-level tensor. This enables benchmarking of the slowdown caused by the dummy edge-level tensor. Kindly note that backward SpMMv requires the transpose of the topology only due to the absence of edge weights. Hence, SpMMv^T is always SpMMv on CSC. We have an in-place degree-norm.

In total, the format costs in GRAPHPY is only $|V|+3|E|$ where CSR/CSC needs ($|V|+|E|$) + COO ($|E|$) + Edge ID array ($|E|$) in CSC for class A GNN. For GCN, GRAPHPY need only CSR ($|V|+|E|$).

GRAPHPY provides additional advantages. *a)* Its *practical* design choices are deployable to address a few pitfalls, thereby influencing a large number of existing systems. *b)* It protects future works from showing inflated gains, otherwise, they might continue to rely on current inefficient baselines or other pitfall-manifested systems for benchmarking, leading to compromised evaluations.

7 Experiments

The datasets are listed in Table 1. Only Cora, Pubmed, Citeseer, Reddit, and OGBN-Products datasets are labeled. The first three datasets are referred to as *small*. We use an Nvidia A100 GPU (40GB memory) and CUDA version 11.3. We focus on node classification using GCN, GIN, and GAT (with 1 and 3 heads implying edge feature length). In GCN and GAT, the intermediate feature length is 16, which implies that GAT-3 has 48 as the feature-length for each vertex. For GIN, the intermediate vertex feature length is 64. The feature lengths have been set based on the code of the original model paper, thus covering various feature lengths.

We used DGL version 1.1.0 throughout the paper. We also compare against Ge-Spmm [26], GNNAdvisor [54], Huang et al [27], FeatGraph [25], TC-GNN [56] and MariusGNN [50]. PyG runs out of memory on Reddit and OGBN-Products, so it is not benchmarked. We use dgNN [66] as a case study to understand the true impact of its kernel fusion and memory-saving techniques (§7.2.3) in the presence of no framework memory overhead and correctly designed SpMM^T.

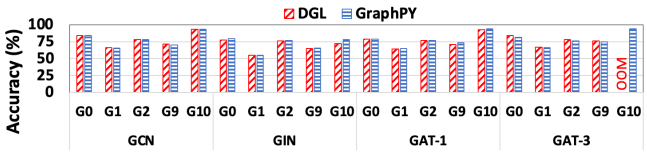


Fig. 8: Accuracy measure of GRAPHPY and DGL. Other systems have accuracy issues(§4, Fig. 3).

Accuracy Check Before Comparison. Fig. 8 shows the accuracy of GRAPHPY compared to DGL to show that GRAPHPY has normal accuracy. Thus, GRAPHPY can be used for other evaluations to show the impact of pitfalls in later sections.

7.1 GNN System Design Pitfalls Evaluations

Fig. 9 shows that the simple design changes have brought large speedup benefits to GRAPHPY over DGL. For the Reddit dataset, GRAPHPY achieves around 2.30× speedup for GCN, 1.36× for GIN, and 2.37× for GAT-1 over DGL. For GAT-3, DGL resulted in OOM. Smaller datasets are plotted later in Fig. 17 to discuss framework overhead. This subsection discusses the reason behind the speedup by focusing on kernel evaluations. We also discuss the impact of system design pitfalls on accuracy and runtime using the GRAPHPY prototype and other systems.

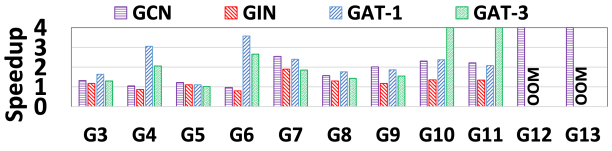


Fig. 9: GRAPHPY speedup over DGL for GNN training time (200 iterations). A speedup of 4 means that only DGL has OOM. OOM means that both DGL and GRAPHPY have OOM. (Higher is better.)

7.1.1 SpMMve^T and eShuffle: Quantitative Analysis

Fig. 10 plots the execution time of the SpMMve^T kernel. The SpMMve^T is natively implemented by cuSPARSE, but DGL used eShuffle+cuSPARSE SpMMve as a substitute (§4.1.2).

a) SYS-P2: DGL's eShuffle+SpMMve is 1.64× slower than the cuSPARSE-native *SpMMve^T* API. Worse, DGL also used eShuffle during forward computation, further contributing

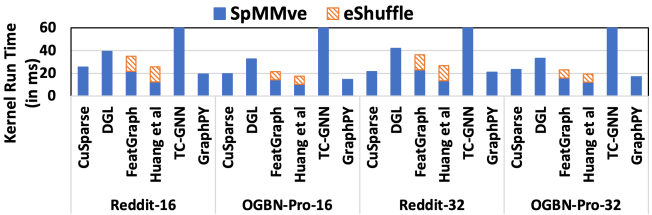


Fig. 10: SpMMve^T kernel time for 16 and 32 features. TC-GNN is very slow and has been clipped. (Lower is better.)

to the slowdown. FeatGraph also performs slower as it focuses only on forward SpMM, and not in $SpMM^T$. As it is integrated into DGL, it would suffer DGL's eShuffle issues.

b) EVAL-P2: TC-GNN uses SpMMve in its backward computation instead of SpMMve^T. Despite this advantage, its kernel is slower by over 3× and 54× on OGBN-Products and Reddit, respectively, for feature-length 32. Because it relies on fine-grained tiling to use tensor core, which does not suit the unstructured sparsity of mid-size graph datasets. *This result demonstrates that exclusively focusing on smaller datasets (EVAL-P2) by TC-GNN leads to misleading conclusions.*

c) SYS-P1: Huang et al. do not implement backward computation (SYS-P1), so we add our eShuffle time to their SpMMve time to simulate SpMMve^T. Other mechanisms, such as cusparseCsr2cscEx2() API, are not considered as it is almost 2× more costly than eShuffle. Its performance over cuSPARSE SpMMve^T is not noticeable and is slower in some cases.

d) GRAPHPY*'s Fused eShuffle+SpMMve is a state-of-the-art SpMMve^T solution* as it achieves 31.97×, 1.60×, 1.27×, and 1.24× speedup compared to TC-GNN, FeatGraph, cuSPARSE, and Huang et al., respectively. Kindly note that GRAPHPY is a vanilla vertex-parallel SpMMve, which is slower than the SpMMve of Huang et al due to the latter's custom format and workload-balanced design. Despite that, GRAPHPY SpMMve^T outperforms it and others.

e) GRAPHPY*'s SDDMM is state of the art*. Focusing on designing transposed SpMM has indirectly allowed GRAPHPY to gain huge performance in SDDMM due to the data-locality (§6.3). Fig. 11 shows that GRAPHPY achieves 2.99× speedup over DGL SDDMM. Both systems use edge-level parallelism on GPUs using the COO format, but GRAPHPY's COO is stored in a CSR-way (row-major) while DGL's COO is randomly laid out. Many other GNN systems do not have a separate SDDMM. FeatGraph does provide SDDMM, but its idea has been merged with DGL. cuSPARSE is very slow [64], which we have also observed, and hence has not been plotted. We compare against dgNN SDDMM in §7.2.3. However, the fact is that GRAPHPY achieved a huge performance with just a small change in DGL's COO to make it a CSR-way COO.

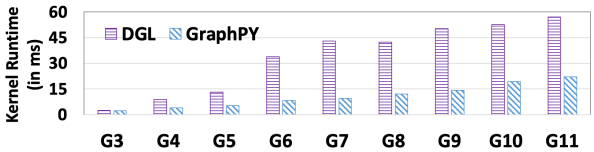


Fig. 11: SDDMM kernel runtime for DGL and GRAPHPY for feature-length 32. Both are edge-parallel. (Lower is better.)

f) SYS-P2-2: Fig. 12 shows that substituting SpMM^T with SpMMve leads to achieving an average of 1.34×, a notable training runtime speedup, for two widely used mid-size datasets, when GRAPHPY is used as an underlying system. Further, this pitfall also corrupts the training accuracy (Fig. 13) by 26.8%, 16.1%, 18%, 15.63%, and 11.96% in G0 – G2, G9, and G10 respectively.

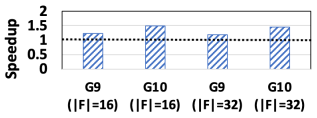


Fig. 12: Omitting the transpose operation (§4.1.2) from GAT backward computation wrongly leads to faster training.

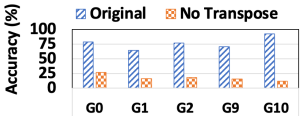


Fig. 13: Removing the transpose operation in GAT backward computation drops training accuracy.

In summary, DGL's performance slowdown is marred by bad design choices despite having easier options. Other systems suffer from various other pitfalls affecting their performance or accuracy, However, the takeaway is that a simple format change has allowed GRAPHPY to achieve state-of-the-art performance. This change is very practical to easily integrate into other systems to either gain performance or implement SpMMve^T to fix their system.

7.1.2 SpMMv and Dummy Edge-Level Tensor

A similar analysis for SpMMv is plotted in Fig. 14. We use cuSPARSE as the baseline.

a) EVAL-P2: TC-GNN and MariusGNN (both published in 2023) are the slowest systems. TC-GNN is slower by 3.07× and 38.67× on OGBN-Products and Reddit, respectively, over cuSPARSE. MariusGNN, a sampling-based GNN system, is 6.18× and 9.71× slower, respectively, due to the materialization of dot products in the GPU global memory (§4.1.2). GNNAdvisor is also slower (1.20×) than cuSPARSE despite using a workload-balanced technique, and is 1.98× slower than the workload-imbalanced solution of Ge-SpMM, even though it uses its caching design. Unfortunately, it materializes the dot products in GPU shared memory, leading to lower performance. As they have specifically mentioned their kernel design to be a reason for better performance, *they got impacted by EVAL-P2 (evaluation on smaller datasets only).*

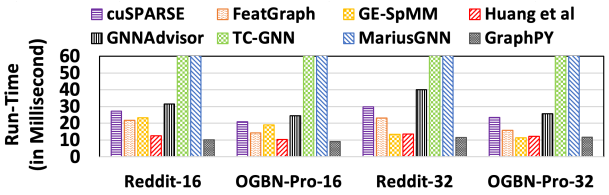


Fig. 14: SpMMv kernel time: TC-GNN and MariusGNN runtime have been clipped.(Lower is better.)

b) Dummy edge-level Tensor: cuSPARSE and Huang et al provide SpMMv using their SpMMve + dummy edge-level tensor with each value as 1.0, which incurs additional data-load. Hence, Huang et al (despite a workload-balanced solution) still perform similarly to SpMMv of the workload-imbalanced solution of Ge-SpMM. We believe it could have made a greater impact had it implemented a native SpMMv.

c) GRAPHPY's vanilla vertex-parallel SpMMv is better

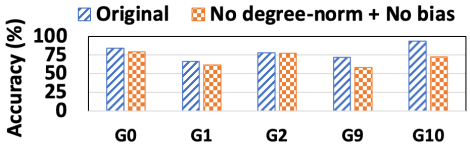


Fig. 15: Removing the degree-norm and bias operation in GCN leads to a large accuracy drop in mid-size datasets. (Higher is better.)

than all prior works, e.g., it outperforms workload-balanced GNNAdvisor's SpMMv by 2.87×, outperforms MariusGNN and TC-GNN SpMMv by 14.89× and 56.99× respectively. The reason for its better performance is purely poor design decisions of prior works as discussed above.

7.1.3 Removal of Degree-Norm and Bias Parameter

Fig. 15 shows accuracy drop of 5.1%, 4.4%, 0.7%, 13.1%, and 20.9% in G0, G1, G2, G9, and G10 datasets, respectively, when eliminating the degree-norm and bias parameter for the GCN model. Larger datasets have a larger drop in accuracy. This confirms that such system design issues lower accuracy.

7.2 Framework Level Pitfalls Evaluation

This section focuses on both the framework overheads: runtime and memory. We then use dgNN to discuss the trade-off of its kernel fusion and memory-saving techniques.

7.2.1 Framework-Memory Overhead Impact

Fig. 16 shows that GRAPHPY reduces memory consumption on average by 6.92× on GCN, 3.4× on GIN, and 1.96× on GAT-1 over DGL. We exclude results for G3–G8 as the focus is to study OOM behavior, though we observe a similar pattern for those datasets. On Reddit (G10), GRAPHPY (DGL) consumes only 2.1GB (23.2GB), 4.0GB (23.7GB), 13.3GB (30.3GB), and 30.9GB(OOM) on GCN, GIN, GAT-1, and GAT-3, respectively.

GRAPHPY provides over 20GB of memory saving in GCN for Reddit over DGL. The memory benefit of GRAPHPY is first due to the removal of framework-memory overhead in DGL (§5.2), while other techniques, such as better storage format, removal of dummy edge-level tensor in GCN SpMMv, removal of eShuffle kernel in GAT SpMMve to save the memory of an edge-level tensor, etc., provide true savings.

The true memory consumption allows GRAPHPY *to train GCN on a billion-edge graph (G13: Kron-25), a first on single-GPU*, consuming 29.8 GB only, while DGL cannot even run a graph with around half a billion edges (G12: UK-2002), showing the inefficiencies in DGL.

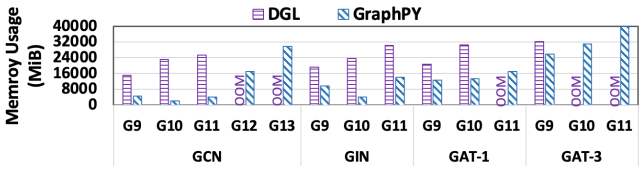


Fig. 16: Memory consumption comparison. OOM=out-of-memory. (Lower is better.)

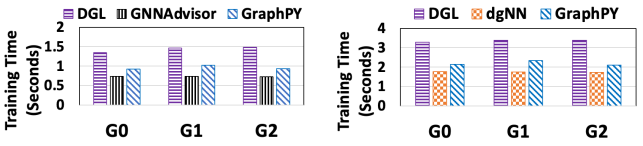


Fig. 17: Training runtime on smaller datasets. **Left:** GCN **Right:** GAT. (Lower is better.)

7.2.2 Framework-Runtime Overhead Impact

Fig. 17 shows training runtime on smaller datasets to focus on runtime overhead. We first notice that GRAPHPY performed much better than DGL when GRAPHPY used the cuSPARSE kernel, same as DGL except for the removal of the message passing. Kindly note that GRAPHPY performed the same irrespective of using cuSPARSE-based SpMM or its vanilla vertex-parallel SpMM (not plotted). Both observations align with our framework-runtime overhead analysis (§5.1).

We also compare with GNNAdvisor and dgNN. The former moved a lot of Python code to C++ (it also has removed bias and performs degree-norm wrongly in the backward). At the same time, the latter uses kernel fusion, resulting in a smaller Python/C++ code footprint than GRAPHPY. Both approaches are expected to reduce overhead, which is proven by the result.

In the absence of knowledge about framework-runtime overhead pitfalls, one can easily claim that GNNAdvisor and dgNN are faster than GRAPHPY as both systems have strong arguments for better performance: GNNAdvisor has workload-balanced SpMMv, while dgNN has a superior kernel-fusion design compared to GRAPHPY. However, both arguments are not valid for mid-sized datasets (see §7.1.2 for GNNAdvisor and §7.2.3 for dgNN) where they matter.

7.2.3 Case-Study: Kernel-Fusion + Memory Saving

GRAPHPY still keeps the baseline characteristics of DGL, hence we use it to benchmark dgNN [66]: superior **kernel fusion** in GAT, whose forward computation stores attention scores ($|E|$) in shared memory to enable its faster fusion with forward SpMMve. The backward phase recomputes the attention (gradient checkpointing) to store in shared memory to fuse it with backward SpMM. Kindly note that the backward recomputation of the attention score in dgNN directly generates a transposed layout of the attention score; hence, fused backward needs SpMMve and not SpMMve^T, which is another way to achieve SpMMve^T.

dgNN is slower than GRAPHPY by 1.48× for datasets G3–G11 (Fig. 18). The SpMMve in GRAPHPY is similar to dgNN,

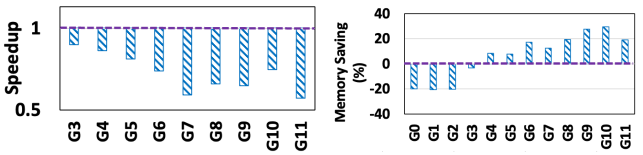


Fig. 18: dgNN GAT training speedup over GRAPHPY. **Left**: Runtime speedup. **Right**: Memory saving speedup. (Higher is better.)

which has been fused with other kernels in dgNN. Hence, the reasoning lies in the trade-off in GNN of kernel fusion.

GRAPHPY has SpMM and SDDMM in vertex-parallel and edge-parallel fashion respectively, like DGL. However, this mixed approach does not allow fusing these two. Hence, dgNN, like other kernel fusion works [9,57], forces SDDMM variants to be performed in vertex-parallel fashion to fuse it with their vertex-parallel SpMM. Our measurements (not plotted) confirm that GRAPHPY has 1.72× speedup over dgSparse [1] (vertex-parallel SDDMM, which is used by dgNN for fusion) on OGBN-Products dataset.

Fundamentally, SDDMM is naturally edge-parallel, as computation on one edge is independent of others. On the other hand, vertex-parallel SDDMM has a workload imbalance as vertices have varying neighborhood sizes. Hence, when one keeps the same SpMM implementation, the advantage of kernel fusion boils down to its benefit versus the loss that occurs by downgrading SDDMM to vertex-parallel. *Hence, even though dgNN is faster than DGL is a correct assertion, it is not faster than the latter's approach of vertex-parallel SpMM + edge-parallel SDDMM, when implemented rightly as by* GRAPHPY. *This represents more fundamental knowledge.*

Average Memory Saving is modest 6.4%. This shows that gradient checkpointing provides memory saving, which we expected. However, dgNN theoretically provides only $|E|$ - $2|V|$ memory saving for state-tensors per GAT convolution layer: $|E|$ memory saving is achieved by not storing attention (edge-level) tensor as a state-tensor, while recomputing is based on introducing two new vertex-level state tensors ($2|V|$). Other kernel fusion techniques save only ephemeral memory as they are not state-tensors.

For some datasets, the saving is negative. Our measurements show that dgNN leaks memory around 150 MB, i.e., the memory consumption around the 200^{th} iteration of training is larger by 150 MB than that of around the first few iterations. GRAPHPY (and DGL) shows the same memory consumption throughout the training procedure. This leak becomes significant in smaller datasets of G0 – G3, leading to negative memory saving in dgNN.

In summary, framework-memory overhead in DGL offered inflated evaluations in dgNN that reported up to 3× memory saving. Our evaluation with GRAPHPY shows its true saving.

8 Related Works

Several papers have uncovered pitfalls or offered alternative perspectives that diverge from established research directions in machine learning (ML) and deep learning (DL) applications. For example, a distinguished award paper [3], systematically describes the ML pitfalls in the security domain. Earlier, another best paper award [11] explains pitfalls in the recommendation system.

From the systems perspective, some studies challenge prevailing assumptions. McSherry et al. [39] show that many graph analytics systems merely parallelized the overhead rather than improving performance when scaling to a distributed deployment. GBench [34] highlights existing pitfalls when benchmarking graph streaming systems. HalfGNN [44] identifies training accuracy flaws in prior GNN systems when they substitute single-precision floating point features with half-precision.

These studies highlight the existence and importance of recognizing pitfalls in the ML/DL domain and beyond, and their recognition by the computer science community underscores their significance. In contrast to these works, our study focuses on system-specific pitfalls in GPU-based GNN systems. Our discussion is based on semantic-preserving GNN models: the model architecture, algorithm, and datasets remain unchanged, and only the underlying system or backend is modified. Under such settings, an ideally optimized system should deliver comparable accuracy while achieving improved performance. Our work investigates why this is often not the case, as we show that design decisions at the system and framework level can lead to significant degradation in both accuracy and performance.

It is important to clarify that while it is widely acknowledged that changes to model architecture or algorithm can affect accuracy and performance, this paper focuses on semantic-preserving models.

Moreover, GNN optimizations have wider applications in exploiting sparsity in the broader DL community, where models are either pruned [10,13,21,22] to make their layers sparse, which requires re-training or defined initially [38] as sparse, thereby requiring similar to GNN sparse kernels. Hence, pitfalls in GNN systems will likely impact this area as well.

9 Conclusion and Future Works

Our in-depth analysis highlighted many critical pitfalls in current GPU-only GNN systems which questions our understanding of system design, implementation, and evaluation. We identified the consequences of ignoring accuracy measurements and unawareness of the role of the framework in training time and memory consumption. We presented their impact, recommendations, and future directions. We also designed a practical and compelling solution to solve some of these pitfalls. The important takeaway is that a simple yet careful design rooted in requirement understanding leads to better runtime and memory usage. We hope our analysis, recommendations, and GRAPHPY can be used by future research to genuinely advance GNN system optimizations. We leave overcoming framework-runtime overhead as a future work.

Acknowledgments

We would like to thank the anonymous reviewers for their valuable feedback and Tim Harris for his suggestions and guidance throughout the shepherding process. This work is supported in part by National Science Foundation (NSF) grant 2245849. The views, opinions, and findings of the paper are of the authors and do not represent the official views or policies of NSF.

References

[1] dgSparse. https://dgsparse.github.io/.

[2] M. S. Anderson, A. S. Horn, K. R. Risbey, E. A. Ronning, R. De Vries, and B. C. Martinson. What do mentoring and training in the responsible conduct of research have to do with scientists' misbehavior? Findings from a national survey of NIH-funded scientists. *Academic Medicine*, 82(9):853–860, 2007.

[3] D. Arp, E. Quiring, F. Pendlebury, A. Warnecke, F. Pierazzi, C. Wressnegger, L. Cavallaro, and K. Rieck. Dos and don'ts of machine learning in computer security. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 3971–3988, 2022.

[4] T. Baruah, K. Shivdikar, S. Dong, Y. Sun, S. A. Mojumder, K. Jung, J. L. Abellán, Y. Ukidave, A. Joshi, J. Kim, et al. GNNMark: A Benchmark Suite to Characterize Graph Neural Network Training on GPUs. In *2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 13–23. IEEE, 2021.

[5] X. Bresson and T. Laurent. Residual gated graph convnets. *arXiv preprint arXiv:1711.07553*, 2017.

[6] Z. Cai, X. Yan, Y. Wu, K. Ma, J. Cheng, and F. Yu. DGCL: An Efficient Communication Library for Distributed GNN Training. In *Proceedings of the Sixteenth European Conference on Computer Systems*, pages 130–144, 2021.

[7] Z. Cai, Q. Zhou, X. Yan, D. Zheng, X. Song, C. Zheng, J. Cheng, and G. Karypis. DSP: Efficient GNN Training with Multiple GPUs. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, PPoPP '23, page 392–404, 2023.

[8] Z. Chen, Z. Qu, L. Liu, Y. Ding, and Y. Xie. Efficient tensor core-based GPU kernels for structured sparsity under reduced precision. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '21, New York, NY, USA, 2021. Association for Computing Machinery.

[9] Z. Chen, M. Yan, M. Zhu, L. Deng, G. Li, S. Li, and Y. Xie. fuseGNN: Accelerating Graph Convolutional Neural Network Training on GPGPU. In *Proceedings of the 39th International Conference on Computer-Aided Design*, pages 1–9, 2020.

[10] R. Child, S. Gray, A. Radford, and I. Sutskever. Generating Long Sequences with Sparse Transformers. 2019.

[11] M. Ferrari Dacrema, P. Cremonesi, and D. Jannach. Are we really making much progress? A worrying analysis of recent neural recommendation approaches. In *Proceedings of the 13th ACM conference on recommender systems*, pages 101–109, 2019.

[12] M. Fey and J. E. Lenssen. Fast Graph Representation Learning with PyTorch Geometric. In *ICLR 2019 Workshop on Representation Learning on Graphs and Manifolds*, 2019.

[13] J. Frankle and M. Carbin. The Lottery Ticket Hypothesis: Finding Sparse, Trainable Neural Networks. In *Seventh International Conference on Learning Representations (ICLR)*, 2019.

[14] Q. Fu, Y. Ji, and H. H. Huang. TLPGNN: A Lightweight Two-Level Parallelism Paradigm for Graph Neural Network Computation on GPU. In *Proceedings of the 31st International Symposium on High-Performance Parallel and Distributed Computing*, pages 122–134, 2022.

[15] S. Gandhi and A. P. Iyer. P3: Distributed Deep Graph Learning at Scale. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI'21)*, pages 551–568, 2021.

[16] Y. Gong and P. Kumar. GNNOne: A Unified System Optimizations for GNN Kernels. In *Proceedings of the 33rd International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '24, page 15–27, New York, NY, USA, 2024. Association for Computing Machinery.

[17] Graph500. http://www.graph500.org/.

[18] GTgraph: A suite of synthetic random graph generators. http://www.cse.psu.edu/~madduri/software/GTgraph/.

[19] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan. Deep learning with limited numerical precision. In *Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37*, ICML'15, page 1737–1746. JMLR.org, 2015.

[20] W. Hamilton, Z. Ying, and J. Leskovec. Inductive Representation Learning on Large Graphs. In *Advances in neural information processing systems*, pages 1024–1034, 2017.

[21] S. Han, H. Mao, and W. Dally. Deep Compression: Compressing Deep Neural Network with Pruning, Trained Quantization and Huffman Coding. In *4th International Conference on Learning Representations (ICLR-16)*, 2016.

[22] S. Han, J. Pool, J. Tran, and W. Dally. Learning both weights and connections for efficient neural network. In *Advances in neural information processing systems*, pages 1135–1143, 2015.

[23] N.-M. Ho and W.-F. Wong. Exploiting half precision arithmetic in Nvidia GPUs. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7, 2017.

[24] J. Hu, S. Qian, Q. Fang, Y. Wang, Q. Zhao, H. Zhang, and C. Xu. Efficient graph deep learning in tensorflow with tf_geometric. In *Proceedings of the 29th ACM International Conference on Multimedia*, pages 3775–3778, 2021.

[25] Y. Hu, Z. Ye, M. Wang, J. Yu, D. Zheng, M. Li, Z. Zhang, Z. Zhang, and Y. Wang. FeatGraph: A Flexible and Efficient Backend for Graph Neural Network Systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–13, 2020.

[26] G. Huang, G. Dai, Y. Wang, and H. Yang. GE-SpMM: General-purpose Sparse Matrix-Matrix Multiplication on GPUs for Graph Neural Networks. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12. IEEE, 2020.

[27] K. Huang, J. Zhai, Z. Zheng, Y. Yi, and X. Shen. Understanding and Bridging the Gaps in Current GNN Performance Optimizations. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 119–132, 2021.

[28] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio. Quantized neural networks: training neural networks with low precision weights and activations. *J. Mach. Learn. Res.*, 18(1):6869–6898, jan 2017.

[29] A. Jangda, S. Polisetty, A. Guha, and M. Serafini. Accelerating Graph Sampling for Graph Machine Learning using GPUs. In *Proceedings of the Sixteenth European Conference on Computer Systems*, 2021.

[30] Z. Jia, S. Lin, M. Gao, M. Zaharia, and A. Aiken. Improving the Accuracy, Scalability, and Performance of Graph Neural Networks with Roc. *Proceedings of Machine Learning and Systems*, 2:187–198, 2020.

[31] T. Kaler, N. Stathas, A. Ouyang, A.-S. Iliopoulos, T. Schardl, C. E. Leiserson, and J. Chen. Accelerating Training and Inference of Graph Neural Networks with Fast Sampling and Pipelining. *Proceedings of Machine Learning and Systems*, 4:172–189, 2022.

[32] I. Kim, J. Jeong, Y. Oh, M. K. Yoon, and G. Koo. Analyzing GCN Aggregation on GPU. *IEEE Access*, 10:113046–113060, 2022.

[33] T. N. Kipf and M. Welling. Semi-Supervised Classification with Graph Convolutional Networks. In *5th International Conference on Learning Representations (ICLR-17)*, 2017.

[34] P. Kumar and S. Revillar. G-Bench: Fair Benchmarking to Support Innovations in Streaming Graph Systems. In *2023 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 179–188, 2023.

[35] S. Liang, Y. Wang, C. Liu, L. He, L. Huawei, D. Xu, and X. Li. EnGN: A High-Throughput and Energy-Efficient Accelerator for Large Graph Neural Networks. 70(9):1511–1525, 2020.

[36] Z. Lin, C. Li, Y. Miao, Y. Liu, and Y. Xu. PaGraph: Scaling GNN Training on Large Graphs via Computation-Aware Caching. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, pages 401–415, 2020.

[37] T. Liu, Y. Chen, D. Li, C. Wu, Y. Zhu, J. He, Y. Peng, H. Chen, H. Chen, and C. Guo. BGL: GPU-Efficient GNN Training by Optimizing Graph Data I/O and Preprocessing. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, 2023.

[38] Z. Liu, M. Sun, T. Zhou, G. Huang, and T. Darrell. Rethinking the Value of Network Pruning. In *Seventh International Conference on Learning Representations (ICLR)*, 2019.

[39] F. McSherry, M. Isard, and D. G. Murray. Scalability! but at what cost? In *Proceedings of the 15th USENIX Conference on Hot Topics in Operating Systems*, HOTOS'15, page 14, USA, 2015. USENIX Association.

[40] V. Md, S. Misra, G. Ma, R. Mohanty, E. Georganas, A. Heinecke, D. Kalamkar, N. K. Ahmed, and S. Avancha. DistGNN: Scalable Distributed Training for Large-Scale Graph Neural Networks. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14, 2021.

[41] N. A. of Sciences, N. A. of Engineering, and I. of Medicine. Responsible Science: Ensuring the Integrity of the Research Process: Volume I. 1992.

[42] SNAP: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data/.

[43] S. Song and P. Jiang. Rethinking Graph Data Placement for Graph Neural Network Training on Multiple GPUs. In *Proceedings of the 36th ACM International Conference on Supercomputing*, pages 1–10, 2022.

[44] A. K. Tarafder, Y. Gong, and P. Kumar. Optimization of GNN Training Through Half-precision. In *Proceedings of the 34th International Symposium on High-Performance Parallel and Distributed Computing*, 2025.

[45] The University of Florida: Sparse Matrix Collection. http://www.cise.ufl.edu/research/sparse/matrices/.

[46] J. Thorpe, Y. Qiao, J. Eyolfson, S. Teng, G. Hu, Z. Jia, J. Wei, K. Vora, R. Netravali, M. Kim, and G. H. Xu. Dorylus: Affordable, Scalable, and Accurate GNN Training with Distributed CPU Servers and Serverless Threads. In *USENIX Symposium on Operating Systems Design and Implementation*, 2021.

[47] C. Tian, L. Ma, Z. Yang, and Y. Dai. PCGCN: Partition-Centric Processing for Accelerating Graph Convolutional Network. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 936–945. IEEE, 2020.

[48] A. Tripathy, K. Yelick, and A. Buluç. Reducing Communication in Graph Neural Network Training. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14. IEEE, 2020.

[49] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Lio, and Y. Bengio. Graph Attention Networks. *6th International Conference on Learning Representations (ICLR-18)*, 2018.

[50] R. Waleffe, J. Mohoney, T. Rekatsinas, and S. Venkataraman. MariusGNN: Resource-Efficient Out-of-Core Training of Graph Neural Networks. In *Eighteenth European Conference on Computer Systems (EuroSys' 23)*, 2023.

[51] C. Wang, D. Sun, and Y. Bai. PiPAD: Pipelined and Parallel Dynamic GNN Training on GPUs. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, PPoPP '23, page 405–418, New York, NY, USA, 2023. Association for Computing Machinery.

[52] L. Wang, Q. Yin, C. Tian, J. Yang, R. Chen, W. Yu, Z. Yao, and J. Zhou. FlexGraph: A Flexible and Efficient Distributed Framework for GNN Training. In *Proceedings of the Sixteenth European Conference on Computer Systems*, pages 67–82, 2021.

[53] M. Wang, D. Zheng, Z. Ye, Q. Gan, M. Li, X. Song, J. Zhou, C. Ma, L. Yu, Y. Gai, T. Xiao, T. He, G. Karypis, J. Li, and Z. Zhang. Deep Graph Library: Towards Efficient And Scalable Deep Learning on Graphs. In *ICLR 2019 Workshop on Representation Learning on Graphs and Manifolds*, 2019.

[54] Y. Wang, B. Feng, G. Li, S. Li, L. Deng, Y. Xie, and Y. Ding. GNNAdvisor: An Adaptive and Efficient Runtime System for GNN Acceleration on GPUs. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 515–531, 2021.

[55] Y. Wang, B. Feng, Z. Wang, T. Geng, K. Barker, A. Li, and Y. Ding. {MGG}: Accelerating Graph Neural Networks with {Fine-Grained}{Intra-Kernel}{Communication-Computation} Pipelining on {Multi-GPU} Platforms. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 779–795, 2023.

[56] Y. Wang, B. Feng, Z. Wang, G. Huang, and Y. Ding. {TC-GNN}: Bridging Sparse {GNN} Computation and Dense Tensor Cores on {GPUs}. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 149–164, 2023.

[57] Y. Wu, K. Ma, Z. Cai, T. Jin, B. Li, C. Zheng, J. Cheng, and F. Yu. Seastar: Vertex-centric Programming for Graph Neural Networks. In *Proceedings of the Sixteenth European Conference on Computer Systems*, pages 359–375, 2021.

[58] K. Xu, W. Hu, J. Leskovec, and S. Jegelka. How Powerful are Graph Neural Networks? *7th International Conference on Learning Representations (ICLR-19)*, 2019.

[59] M. Yan, Z. Chen, L. Deng, X. Ye, Z. Zhang, D. Fan, and Y. Xie. Characterizing and Understanding GCNs on GPU. *IEEE Computer Architecture Letters*, 19(1):22–25, 2020.

[60] M. Yan, L. Deng, X. Hu, L. Liang, Y. Feng, X. Ye, Z. Zhang, D. Fan, and Y. Xie. HyGCN: A GCN Accelerator with Hybrid Architecture. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 15–29. IEEE, 2020.

[61] D. Yang, J. Liu, J. Qi, and J. Lai. WholeGraph: A Fast Graph Neural Network Training Framework with Multi-GPU Distributed Shared Memory Architecture. 2022.

[62] J. Yang, D. Tang, X. Song, L. Wang, Q. Yin, R. Chen, W. Yu, and J. Zhou. GNNLab: A Factored System for Sample-Based GNN Training over GPUs. In *Proceedings of the Seventeenth European Conference on Computer Systems*, pages 417–434, 2022.

[63] S. Yang, M. Zhang, W. Dong, and D. Li. Betty: Enabling Large-Scale GNN Training with Batch-Level Graph Partitioning. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS 2023, page 103–117, New York, NY, USA, 2023. Association for Computing Machinery.

[64] Z. Ye, R. Lai, J. Shao, T. Chen, and L. Ceze. SparseTIR: Composable Abstractions for Sparse Compilation in Deep Learning. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pages 660–678, 2023.

[65] B. Zhang, R. Kannan, and V. Prasanna. BoostGCN: A Framework for Optimizing GCN Inference on FPGA. In *2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 29–39. IEEE, 2021.

[66] H. Zhang, Z. Yu, G. Dai, G. Huang, Y. Ding, Y. Xie, and Y. Wang. Understanding GNN Computational Graph: A Coordinated Computation, IO, and Memory Perspective. *Proceedings of Machine Learning and Systems*, 4:467–484, 2022.

[67] J. Zhang, X. Shi, J. Xie, H. Ma, I. King, and D. Yeung. GaAN: Gated Attention Networks for Learning on Large and Spatiotemporal Graphs. In *Proceedings of the Thirty-Fourth Conference on Uncertainty in Artificial Intelligence*, pages 339–349, 2018.

[68] C. Zheng, H. Chen, Y. Cheng, Z. Song, Y. Wu, C. Li, J. Cheng, H. Yang, and S. Zhang. ByteGNN: Efficient Graph Neural Network Training at Large Scale. *Proc. VLDB Endow.*, 15(6):1228–1242, feb 2022.

[69] D. Zheng, C. Ma, M. Wang, J. Zhou, Q. Su, X. Song, Q. Gan, Z. Zhang, and G. Karypis. DistDGL: Distributed Graph Neural Network Training for Billion-Scale Graphs. In *2020 IEEE/ACM 10th Workshop on Irregular Applications: Architectures and Algorithms (IA3)*, pages 36–44. IEEE, 2020.

A Method to Measure Framework Overhead

In this section, we focus on how to measure framework overhead. Though, it has been plotted in the main paper, the main metric to observe in §5.1 is GPU idle time (100 - GPU-Runtime (%)), where overhead runs exclusively. Hence, this section is only for curious reviewers who wants to go deeper.

For reference, the training time is measured as the wall-clock time spent between training start and end. It is a standard practice to insert a "cudaDeviceSynchronize" (barrier) after the last training iteration to make sure GPU finished execution before ending the wall-clock tick due to asynchronous kernel execution as shown in Fig. 19 (left).

Challenges. Give this definition of training time, we discuss following challenges. **a)** One can remove the inserted barrier, and collect the ending wall-clock tick to postulate that it has collected only CPU runtime. However, a limited job queue length in a GPU makes kernel launch synchronous after a few iteration as many kernels are submitted. This blocking behavior make kernel launches includes an indirect wait time spent by CPU. E.g., the profiler shows 90.27% of the total training time as "kernel launch time" for DGL GCN on Reddit for 200 iterations. This is a problem only for mid-size datasets. In smaller datasets, kernels run much faster and hence, job queue never becomes full. However, this methodology cannot be used due to not giving reliable value of overhead time. **b)** Using PyTorch profiler can also not be used as total CPU time is always printed more than GPU total time as the profiler automatically inserts a barrier after the last training epoch even if we do not explicitly write one. Manually subtracting the timing spent on this barrier, which profiler reports, does not give correct overhead cost as kernel launches include the indirect wait time, discussed above.

```
1 start = datetime.now()
2 for epoch in range(200):
3     # training epoch start
4     # training epoch end
5 # end for loop
6 torch.cuda.synchronize()
7 end = datetime.now()
8 trainingTime = end - start
```

```
1 overhead = 0
2 for epoch in range(200):
3     start = datetime.now()
4     # training epoch start
5     # training epoch end
6     end = datetime.now()
7     ovheread += end - start
8     torch.cuda.synchronize()
9 # end for loop
```

Fig. 19: Methodology for measuring training time (left) and framework runtime overhead (right).

Methodology. We propose an automated approach (Fig. 19 (right)) where the idea is to measure the overhead per iteration by introducing a barrier (Line 8). This barrier limits the number of kernel submissions to ensure kernel launch never becomes synchronous. Hence, time spent between Lines 3 and 6 becomes overhead time (CPU) per iteration. The PyTorch profiler shows that time spent on "cudaLaunchKernel" is much lower. Moreover, this printed similar framework overhead per GNN system per model irrespective of dataset size.

The minor issue with this approach is that total training time might increase due to this barrier. Even though our measurement on different datasets shows hardly any such concerns, we continue to rely on actual definitions to measure training runtime (Fig. 19:Left), and not rely on this changed code. For GPU runtime, we always use PyTorch profiler, which print total gpu runtime.

B Systems Suffering from Framework-Runtime Overhead

The main paper discussed how full-batch GNN for smaller datasets, and sampling+mini-batching GNN for all datasets suffer from framework runtime overhead dominating the training runtime. This section outlines other models that suffer from this issue.

Any other model where a small batch size makes computation lightweight will likely suffer from the same framework runtime overhead. As an example, we evaluated LeNet-300-100 model which is a model with just three linear layers of sizes 768×300, 300×100, and 100×10 with ReLU activations, and classifies hand-written digits, such as MNIST dataset. Measurements show framework-runtime overhead to dominate training time even in large batch-size of 4096.

C Code Analysis For Pitfalls

This section presents our code-study results to show that many prior works have pitfalls. These are supporting materials and are optional to convince that pitfalls exist. Kindly note that we have reached out to the authors of their papers to get confirmation. They all have agreed on the issue and we received positive and encouraging feedback.

C.1 GNNAdvisor

https://github.com/YukeWang96/GNNAdvisor_OSDI21/blob/master/GNNAdvisor/GNNConv/GNNAdvisor_kernel.cu#L542 points to backward propagation which performs the SpMMv first, then the degree-norm part within the same kernel. The forward also performs these operations in the same order(line 403), and hence backward should have been done in reverse order.

https://github.com/YukeWang96/GNNAdvisor_OSDI21/blob/master/GNNAdvisor/gnn_conv.py#L98 shows that the bias operator is not there. https://github.com/dmlc/dgl/blob/master/python/dgl/nn/pytorch/conv/graphconv.py#L206 shows that the bias operator is set as True by default in DGL. But https://github.com/YukeWang96/GNNAdvisor_OSDI21/blob/master/dgl_baseline/gcn.py#L12 shows GNNAdvisor did not specifically disable bias in DGL.

C.2 TC-GNN

https://github.com/YukeWang96/TC-GNN_ATC23/blob/atc23ae/TCGNN_conv/TCGNN_kernel.cu#L336 and https://github.com/YukeWang96/TC-GNN_ATC23/blob/atc23ae/gnn_conv.py shows that TC-GNN's GCN implementation does not include degree-norm and bias operator either in the kernel or in Python. But these two URLs (https://github.com/YukeWang96/TC-GNN_

ATC23/blob/atc23ae/dgl_baseline/gcn.py#L20 https://github.com/dmlc/dgl/blob/master/python/dgl/nn/pytorch/conv/graphconv.py#L197) show that TC-GNN uses the default setting for bias and degree-norm in DGL. This means their DGL baseline enables both the degree-norm and bias operator.

One can also see that its AGNN (https://github.com/YukeWang96/TC-GNN_ATC23/blob/atc23ae/TCGNN_conv/TCGNN_kernel.cu) kernels don't do tranpose at all during backward pass.

C.3 FuseGNN

https://github.com/apuaaChen/gcnLib/issues/3 reported an accuracy issue in Github but got no response. Our measurements also show the same accuracy issue.

C.4 Huang et al

https://github.com/xxcclong/GNN-Computing/blob/master/include/aggr_gat.h There is no transpose SpMM function implemented, the kernels work fine for forward but are not workable for backward pass. Their Python code also does not explicitly call/have transposed API from Pytorch/cuSPARSE.

It uses the same copy of edge-level tensor for all the layers of GAT. Hence, in the real-world, they would not be usable in backward propagation. The correct approach would have been to allocate separate memory for each edge-level tensor.

The kernel does not work when the dim is not a multiple of 32, which is a common case for the last layer of GAT. E.g., in Reddit and OGBN-Products, the feature dimensions are 41 and 47 respectively.

For GCN: https://github.com/xxcclong/GNN-Computing/blob/master/include/aggr_gcn.h shows that all the kernels here require an edge-level tensor, i.e. this paper provides SpMMve and not SpMMv, and is the main reason for slower performance despite having a workload-balanced solution.

The kernel does not work for odd dimensions, which is a common occurrence for GCN, specifically for the last layer. E.g., 6 for Cora.

C.5 TLPGNN

https://github.com/charlifu/TLPGNN/blob/main/gat/kernel.cu#L49 shows the fused kernel for GAT forward where the attention-score (edge-level tensor) is computed and stored in a register and not materialized in GPU global memory.

It also neither has backward code for GAT nor transpose code anywhere.

C.6 Seastar

Seastar does not have its code in Github, but it provides a URL to download it. Upon running its training script, it generates the kernel code (in CUDA), compiles it (all automatically), and runs training. We read the generated GCN sparse kernel, which generates a fused version of SpMMv and degree-norm, and suffered from the same set of issues as discussed for GNNAdvisor.

C.7 MariusGNN

https://github.com/marius-team/marius/blob/2f27ffedfbffd405995e8d16d821849db1fe0535/src/cpp/src/nn/layers/gnn/graph_sage_layer.cpp shows its computation kernel for the SpMM kernel. This kernel consists of two parts. The first part scatters the vertex-level features to edges https://github.com/marius-team/marius/blob/2f27ffedfbffd405995e8d16d821849db1fe0535/src/cpp/src/nn/layers/gnn/graph_sage_layer.cpp#L49.resulting in materializing this value. The second part performs the gather procedure https://github.com/marius-team/marius/blob/2f27ffedfbffd405995e8d16d821849db1fe0535/src/cpp/src/nn/layers/gnn/graph_sage_layer.cpp#L52. We use this method to compare its kernel performance.

C.8 GE-SpMM and FeatGraph

These two systems are integrated to DGL. However, due to code reorganization, we could not run Ge-SpMM. For Featgraph, we also could not run its DGL integrated system despite help from authors. We tried many Github branches that their authors suggested.

C.9 DGL

We have reached out to DGL recently through a long email. We are yet to hear from the person. Before this, we had a face-to-face meeting with this person in a conference/workshop, and the employee indicated that they acknowledge that DGL suffers from some issues that we highlighted in the paper. At that time, the manuscript was not fully ready, but we did discuss that their non-standard integration seems to indicate the memory overhead issue.

Bluetooth Low Energy Security Testing with Combinatorial Methods

Dominik-Philip Schreiber
SBA Research

Manuel Leithner
SBA Research

Jovan Zivanovic
SBA Research

Dimitris E. Simos
SBA Research
Salzburg University of Applied Sciences
Paris Lodron University of Salzburg

Abstract

Wireless protocols such as Bluetooth Low Energy (BLE) play a vital role in ubiquitous computing and Internet of Things (IoT) devices. Numerous vulnerabilities in a variety of devices and components of the BLE stack have been uncovered in recent years, potentially affecting millions of customers. Being notoriously difficult to test due to the level of abstraction commonly enforced by the Host Controller Interface (HCI), a recent work successfully implements a fuzzing framework utilizing a custom firmware for a BLE device. However, fuzzing is inherently probabilistic, which may lead to faults remaining undiscovered. In this work, we enhance the aforementioned method with a Combinatorial Security Testing (CST) approach that provides a guaranteed degree of input space coverage. Through an evaluation targeting 10 BLE devices and a variety of firmware versions, we identify a total of 19 distinct issues, replicating findings of the previous work and uncovering additional faults. We additionally provide a performance overview of our tool and the original fuzzer, comparing their execution time and fault detection capabilities.

1 Introduction

BLE [1] is a wireless transmission technology that was first introduced in Bluetooth core specification 4.0 in 2010 as a means of enabling resource constrained entities such as IoT and health care devices to communicate. Despite the similarity in name, it is a separate protocol from Bluetooth classic. According to Berua et al. [4], BLE is the most widely used protocol for IoT devices with 14.2 billion linked IoT devices in 2019. According to forecasts, by 2027, 7.6 billion Bluetooth enabled devices will be shipped annually (most of them supporting BLE and Bluetooth classic).

Because BLE targets battery powered devices with limited computational capabilities and low memory resources, the complexity of the protocol is reduced compared to classic Bluetooth. At the same time, this modification requires adaptations to implement reliable encryption and privacy measures. Due to the widespread adoption of Bluetooth and the fact that it is used in private homes and industrial applications alike, it is important to test applications and protocol implementations in depth.

A variety of approaches have been developed to test different properties of BLE protocol implementations. One aspect of BLE that is often found to be vulnerable is the pairing procedure, especially the "just works" pairing method, which is in general vulnerable to a Man in the Middle (MITM) attack, but was also found to suffer from more severe issues [4, 33].

Other methods focusing on the underlying link layer of Bluetooth protocols have been successful in showing that the firmware controlling Bluetooth chips may harbor vulnerabilities and must be tested extensively [12, 14]. However, these methods are based on fuzzing, which is inherently probabilistic and does not offer guarantees regarding the coverage of the input space. This may lead to bugs remaining uncovered or testing processes incurring infeasibly high execution times. In order to remediate these flaws, we turn to Combinatorial Testing (CT), a black-box testing method based on a model of the input space of a system that features a guaranteed degree of coverage. We adapt the framework used in SweynTooth [14] to execute a combinatorial test set while incorporating feedback from the target BLE device to guide the selection of test cases and evaluate their success in causing unexpected behavior. Our approach features an individual input parameter model per BLE protocol layer. Test sets based on these models are subsequently combined with a granularity as chosen by the practitioner before being submitted to the System under Test (SUT), which is monitored by two separate oracles aiming to identify abnormal operating conditions.

Our primary **contribution** is a CT approach for security testing of BLE devices that offers a guaranteed degree of input space coverage as well as an implementation based on the GreyHound fuzzer. We further provide an experimental evaluation that confirms that the increased coverage permits testers to identify more flaws in total while pointing towards potential avenues for improvement, particularly in terms of execution time and sequence modifications. Our analysis of

detected issues points towards multiple denial of service vulnerabilities as well as further faults that may ultimately lead to memory corruption and potential arbitrary code execution.

The remainder of this work is structured as follows. Section 2 provides relevant background information on BLE, combinatorial security testing, and the GreyHound fuzzer. Subsequently, Section 3 outlines the most important pieces of related work, both in the areas of BLE security testing and combinatorial testing. In Section 4, we provide a detailed description of our methodology, including the overall testing strategy, input modeling considerations, and oracles. Section 5 describes the testing setup, target devices and their firmware versions as well as practical challenges encountered throughout our experiments, followed by a discussion of evaluation results, identified faults, and the differences between the results of GreyHound and our approach in Section 6. We describe the process and results of our responsible disclosure efforts in Section 7. In Section 8, we summarize our work before moving on to a brief outline of potential avenues for future work in Section 9.

2 Preliminaries

This section provides background information on three particular areas that form the focus of our work: Combinatorial Security Testing, the BLE protocol stack, and the GreyHound fuzzer.

2.1 Combinatorial Security Testing

CT is an established black-box testing method that derives tests based on a model of the input parameters of a system under test (SUT). While the underlying concept was explored by Burr and Young in 1998 [8], it received renewed attention after a large-scale empirical study conducted by the National Institute for Standards and Technology (NIST) indicated that all analyzed software faults were triggered by a combination of up to six parameters [22]. This means that exhaustive testing of all combinations of permissible values for each input parameter is often not required to detect faults.

CT utilizes an Input Parameter Model (IPM) containing all externally supplied parameters of a SUT and a list of their possible values. It may further be enhanced with constraints that restrict which combinations of parameter values are allowed to be contained in a single test case (for example, in a test case that encodes the age and age group of a person, a possible constraint could require that if the age group is set to "child", the age must not be greater than 17). A test case assigns one of the permissible values to each of the input parameters. The coverage guarantees of CT stem from its underlying discrete mathematical structure, a Covering Array (CA). A CA is a $N \times k$ array with each of the N rows representing one test case and each of the k columns referring to a parameter. The defining property of a CA is based on an additional value t, which must be between 2 and k (the latter being equivalent to exhaustive testing) and is often referred to as the *strength* in the literature. For every selection of t distinct parameters (columns), a CA is guaranteed to contain every possible parameter-value combination at least once.

Compared to exhaustive testing, the main advantage of CT is a drastic reduction of test set size and thus execution time. Consider, for example, a SUT that accepts 16 binary input parameters. Exhaustively testing this system would require 2^{16} test cases. In contrast to this, a CA with strength $t = 6$ requires merely 179 test cases, a reduction of around 99.997%.

In practice, applying CT usually entails the following steps:

1. Creating an IPM from a description of the SUT, source code or other type of analysis. In this work, IPMs are constructed based on the BLE specification.

2. Generating one or more CAs from the model according to a selected strength. Increasing the strength leads to a larger subset of the input space being explored while also requiring additional test cases.

3. Translating resulting CAs to concrete test sets, as some values may be encoded symbolically and must be transformed to discrete data types.

4. Executing the test cases and gathering the response of the SUT. This may simply be the output of the SUT or more complex behavioral information such as execution traces.

5. Evaluating the SUT's behavior using an oracle to identify whether any faults occurred.

While traditional CT is a type of conformance testing, i.e. seeks to establish whether the SUT processes a wide range of inputs correctly, combinatorial security testing (CST) aims to identify vulnerabilities. This primarily influences the first and last step listed above: In conformance testing, the IPM captures valid values, while in CST, a mixture of valid values and crafted values meant to trigger vulnerabilities (commonly based on expert knowledge on the particular type of attack) is employed. For example, the latter category might encompass overly long sequences, empty values, or particular strings with a special meaning in languages such as SQL (in the context of SQL injections), shell scripts (for command injections) or JavaScript (in the context of XSS). CST additionally utilizes different types of oracles from traditional CT. While a classic CT oracle would seek to identify whether the SUT's output is equivalent to some expected value, CST oracles aim to identify whether a vulnerability has been triggered. Although the findings of the aforementioned NIST fault analysis studies do not directly apply to CST, previous work discussed in Section 3 indicates that the coverage afforded by combinatorial test sets also leads to efficient and effective test sets for the identification of vulnerabilities.

2.2 Bluetooth Low Energy

Bluetooth Low Energy [1] is one of the most widely used communication protocols for IoT devices [4]. It is managed by the Bluetooth Special Interest Group (SIG), consisting of more than 20,000 members, in version 5.6. It operates in the license free 2.4GHz industrial, scientific and medical (ISM) band and is a nearly completely separate protocol from Bluetooth basic rate (BR) and enhanced data rate (EDR). To avoid congestion and collisions, BLE uses Adaptive Frequency Hopping (AFH) to hop between 40 channels (3 advertisement channels, 37 data channels). A BLE network consists of one central device providing a clock and frequency hopping pattern and one or more peripheral devices.

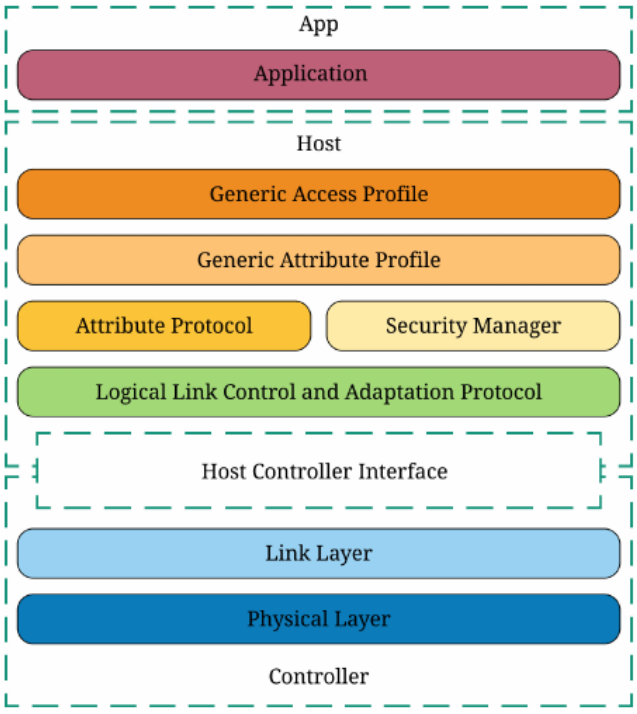


Figure 1: Bluetooth Low Energy stack layers [4]

The BLE stack consists of several protocol layers that interact with each other, each responsible for a different set of functionalities, depicted in Figure 1.

The Host Controller Interface (HCI) represents an isolation layer between lower-layer protocols (Baseband, LMP) and higher-layer protocols (L2CAP, ATT, GATT, GAP, etc.). Higher-layer protocols run on the host operating system, while lower-layer protocols run on a separate hardware called controller, which receives HCI packets from the host that are transformed into appropriate Baseband or LMP packets. The controller also handles timing requirements and checksum calculations. This isolation between components and the inability to access all aspects of the protocol is one of the reasons why the Bluetooth protocol stack is difficult to test.

2.3 GreyHound Fuzzer

Originally, the GreyHound fuzzer framework was developed for testing WiFi protocol implementations [13]. Later, the authors added models for BLE [14] and Bluetooth classic [12], to extend the compatibility of their fuzzer. As our testing approach is built on the GreyHound fuzzer framework, especially its BLE communication functionality, we provide a brief introduction of the hardware and software aspects of this tool in the following paragraphs.

The goal of the GreyHound developers was to find vulnerabilities in BLE protocol implementations [1] and to manipulate and test all their layers. Standards compliant firmware implementations for certified Bluetooth adapters provide only limited access to the inner protocol layers, since they are only accessible through the HCI. The GreyHound fuzzer framework thus utilizes a NRF52840 dongle[1] running a custom firmware implemented in C++ and communicates with it using a USB serial connection. This allows for the use of Python in conjunction with an extension of the scapy [27] framework to craft and dissect arbitrary BLE packets, which are then sent to the dongle. The custom firmware takes care of timing constraints and the CRC calculation and sends the final packet over the air. Contrary to standard HCI implementations, the custom firmware on the dongle provides an unconstrained interface to send raw BLE packets.

The software part of the framework running on the host was implemented using Python 2.7. The framework also includes a state machine, which can initiate the sending of packets and responds according to responses received from BLE peripheral devices. A simplified version of the BLE state machine can be seen in Figure 2. To be compatible with different BLE implementations, the state machine allows for multiple possible paths, since the BLE specification leaves the exact sequence of some interactions open to developers. Because of this, the states depicted in the orange box in Figure 2 can vary in order, depending on the implementation and functionality of the peripheral device.

The state machine starts in the *scanning* state and waits for advertisements from the peripheral. When receiving an advertisement, the central proceeds to the *initiating* state, wherein a connection request is sent to initiate a connection to the peripheral. Now the peripheral can either request GATT services from the central with the central continuing to the *gatt_server* state, or the central and peripheral exchange information about supported features and properties, traversing the states in the orange box in arbitrary order. Subsequently, the central continues on the path after the orange box, negotiating pairing method, exchanging security parameters and keys and finally reading and writing to the GATT services of the peripheral. Finally, if no issues occurred, the central arrives in the *disconnect* state, where it sends a disconnect message to the

[1] https://www.nordicsemi.com/Products/Development-hardware/nRF52840-Dongle

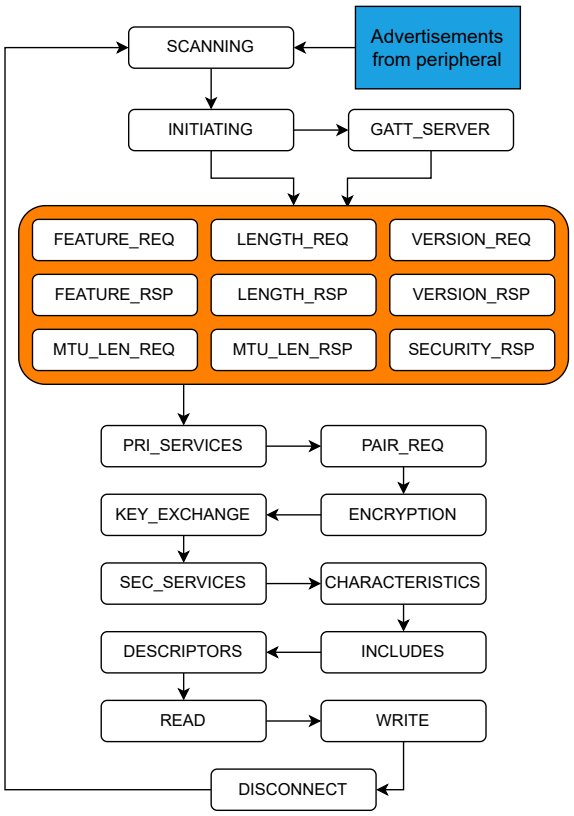


Figure 2: Simplified overview of the BLE state machine [14]. Loops and restrictions regarding state order are omitted.

peripheral and returns to the initial *scanning* state, waiting for advertisements from the peripheral.

Additionally, the GreyHound framework features a fuzzing component that uses the particle swarm optimization (PSO) algorithm of the pyGMO [19] library to guide the probability optimization. The fuzzer module contains two different models, one for WIFI and one for BLE. For every state, the models define what layers a received packet is allowed to contain, which layers of a packet should be fuzzed, what kind of field value mutator should be used for the individual layers and the probability of the individual fields being fuzzed. Initially, all fields and all layers are equally likely to be fuzzed, as long as they are contained in the list of fuzzable layers for that state.

If a field of a layer is selected to be fuzzed, one of the specified mutators is chosen by probability and applied to the field value. The mutator can be either:

- **Random byte:** The field value is replaced by a randomly generated value.
- **Zero filling:** The field value is replaced with zeros.
- **Bit setting:** The most significant bit of every byte is set to zero.

After each iteration through the state machine, the PSO algorithm adjusts the probabilities of the fuzzing model depending on a cost function and which parameters have been fuzzed before. The GreyHound authors wanted to optimize the number of unique anomalies found in a session, which is also chosen as the cost function. For more details of the BLE protocol and its inner workings, we refer the interested reader to the original work [14].

3 Related Work

Due to its popularity, threats to the security and privacy of classic Bluetooth and Bluetooth Low Energy have been studied extensively [4, 9]. Over the years, a variety of vulnerabilities have been discovered [2, 3, 5, 6, 32], often affecting millions of devices. Their impact ranges from simple denial of service attacks to remote code execution or authentication bypasses. Most commonly, finding such vulnerabilities requires significant manual effort such as reverse engineering, code inspection or modification of the source code.

The most closely related work is a more automated BLE fuzz testing approach [14] mentioned in the previous Section, which allows the testing of all layers of the protocol down to the link layer. In addition to custom firmware for the NRF52840 dongle, they constructed a state machine for BLE to utilize as part of a fuzzer guided through probability optimization. This allows for automated testing of peripheral devices with either closed source firmware or BLE development boards, which can be programmed with example applications provided by the vendor of the peripheral chip or other custom implementations. Using this approach, the authors identified 17 previously unknown security vulnerabilities (summarily dubbed *SweynTooth* by the researchers) in the firmware of various BLE controllers. The number of discovered flaws demonstrates the effectiveness of this method, which is further strengthened by the fact that the tool does not need to have access to the source code of the peripheral to test. At the same time, the probabilistic nature of fuzzing may lead to vulnerabilities remaining undiscovered despite extensive testing. This work aims to overcome this restriction through the use of CST methods.

A later work by the same group targets classic Bluetooth devices [12]. This approach is capable of inferring the state machine during execution and features a fuzzer and testing framework similar to their prior work. Instead of using the NRF528400 dongle, which only supports BLE, this research utilizes a ESP32[2] controller.

The use of combinatorial testing in the context of security testing is not new [29]. CST has been successfully used to evaluate web applications with the goal of identifying XSS vulnerabilities [7, 23, 28] and SQL injections [16, 31]. In terms of protocols, prior research attention seems to be focused

[2] https://www.espressif.com/en/products/hardware/esp-wrover-kit/overview

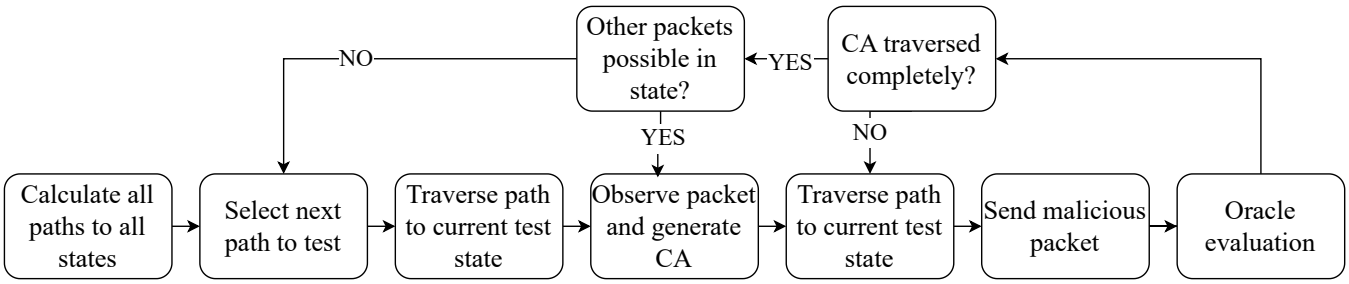


Figure 3: Overview of combinatorial testing strategy.

primarily on TLS. Existing approaches target both X.509 certificate parsers [21, 24] and the flow of protocol messages as well as their contents [15, 30]. While there exist works that aim to identify flaws in IoT devices using CST methods [16, 17], they tend to take a more high-level approach than our contribution, targeting e.g. HTTP REST services instead of lower level protocols.

4 Methodology

This section presents the methodology of our work, including the overall testing strategy, input modeling aspects, CA generation processes and oracles.

Taking into account the considerable success achieved by the authors of the GreyHound fuzzer framework while considering the fundamental drawbacks arising from fuzzing's probabilistic nature, we set out to explore the feasibility of applying CST to generate test cases for BLE implementations. CST has previously been used to perform security testing of TLS implementations [30], but the added complexity of BLE and its interconnected layers requires a more intricate approach. Our work builds upon the aforementioned fuzzing framework, but completely replaces the guided fuzzing component with combinatorial test generation. While this step necessitates the creation of additional models and oracles, our work aims to evaluate whether the guaranteed degree of input space coverage provided by CT leads to the discovery of additional vulnerabilities or bugs.

4.1 Testing Strategy

The original GreyHound fuzzing component traverses the state machine visualized in Figure 2 in an unordered way, guided by probability. In a CST setting, we want to traverse the state machine in a more structured way to ensure that all valid paths (see below) are covered. A simplified overview of our testing strategy is provided in Figure 3, the details of which are discussed in the following paragraphs.

We start by generating a set of all possible paths to every state from the state machine. Subsequently, one path from the set is selected (the exact order is not relevant to our approach, although paths to states closer to the initial state are processed first in our implementation) and we mark all transitions in this path as permissible to traverse by the GreyHound framework. In addition, all transitions that lead to retransmissions and those triggered by timeouts are marked as permitted. The final state of the chosen path is the current state to test. Additionally, we mark all transitions that originate from the current state to test as permitted.

If the current state to test is reachable via this path (which may not always be permitted by the specific SUT), the current path is marked as valid. Invalid paths are currently not tested in our approach, as they are unreachable.

If the path is valid, the next BLE packet that is about to be sent (indicated by the GreyHound framework) is analysed to determine which layers the packet consists of. This packet is the current packet to test. The CST component of our implementation then generates one CA for each of the layers of the current packet to test as well as a meta CA combining these layers, which is used for further testing. Each row in the nested CA represents a test case and contains values for the individual fields of the current packet to test. Details of the CA generation method are given in Section 4.3.

Whenever we reach the current state to test after this point, we select a previously unused row in the CA and change the values of the current packet layers according to the values in the row of the CA in order to create a modified packet which is then transmitted to the SUT. We then continue to send unmodified packets again until an error occurs or we reach the initial state (the scanning state) again, which marks the end of the test. Figure 4 shows an example where the `FEATURE_REQ` state is selected as the current state to test. Unmodified packets (denoted as *Normal Packet*) are transmitted to traverse between all states except for the transition from the state to test to the next state. To traverse further from the state to test, a modified packet is sent, which contains invalid values and is denoted as *Malicious Packet*.

When we reach the end of the CA, we check if there are further outgoing transitions from the current state to itself (i.e. loops). If yes, we generate a new CA for the corresponding packet and proceed as before. Otherwise we continue with another path until all paths to all valid states have been tested.

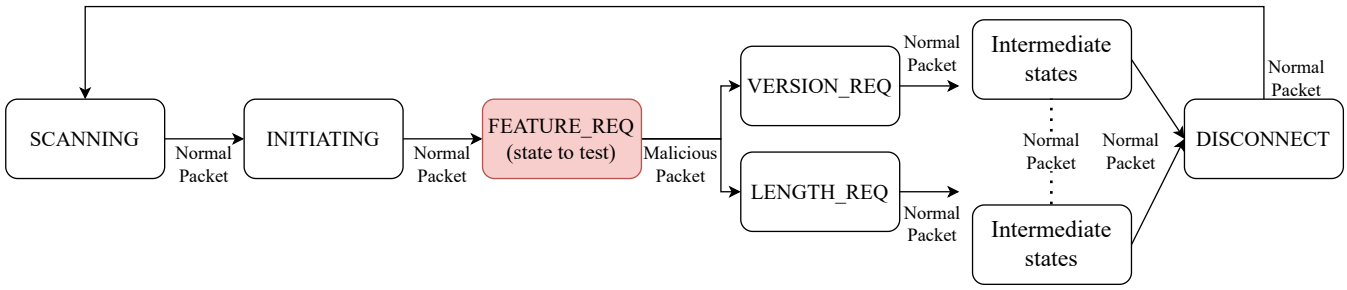


Figure 4: Example testing flow.

4.2 Input Modeling

To generate CAs for testing, we first need to create a model of the input space. This tends to be one of the more challenging tasks in the context of CT, as it commonly requires manual effort to identify all parameters and suitable values.

In this work, we view a BLE packet as a composite system of these layers, with each layer being a component. We utilize the model of layers and their respective fields (which represent our parameters) encoded in the BLE scapy extension contained in the GreyHound framework implementation [14]. For each such field/parameter, we require a set of possible values. To this end, we analyze the source code of the GreyHound framework as well as the BLE specification [1] while considering boundary values [26] of different value ranges. Additionally, we add the symbolic values "LARGER", "SMALLER" or "UNCHANGED" to the parameter values. When applying rows containing these values to a packet, we increment, decrement, or leave unmodified the current value of a field in the original packet. A further symbolic value "RECALCULATE" is added to length parameters and leads to the length value being recalculated based on the current content of the packet.

One individual IPM is constructed for each layer of the BLE protocol. CAs tend to rapidly grow in size when the number of permitted values for a paramater becomes too large. By utilizing multiple smaller models, this effect can be limited, although this slightly reduces overall combinatorial coverage. An example of an IPM for a BLE data layer is shown in Listing 1.

Listing 1: Example of an IPM for a BLE data layer.

```
BTLE_DATA_IPM = {
    'LLID': ["~0", "~1", "~2", "~3", "UNCHANGED"],
    'SN': ["~0", "~1", "UNCHANGED"],
    'NESN': ["~0", "~1", "UNCHANGED"],
    'MD': ["~0", "~1", "UNCHANGED"],
    'RFU': ["~0", "~1", "UNCHANGED"],
    'len': ["~0x00", "RECALCULATE", "~LARGER", "~SMALLER",
            "~0xff"]
}
```

Care must be taken when encoding invalid values for a particular parameter, i.e., values that are not generally permitted to be used for a field. In a process called negative value testing by the authors of PICT [11], it is favorable that only a maximum of one such value be present in a row in the CA. The reasoning behind this choice is that faults or other effects triggered by the presence of an invalid value may mask the effects of additional invalid values. In PICT, it is possible to encode such values by prefixing them with the "∼" symbol.

4.3 Covering Array Generation

The CA generation occurs dynamically during the execution of our CST method. As mentioned in the previous section, we view BLE packets as composite systems, with the layers modeled as subcomponents. To this end, we turn towards a method developed by Kampel et al. [20] that targets composed software systems. Subcomponents are modeled separately, as we have done for the individual packet layers as described in Section 4.2. Using these layer IPMs, so-called seed arrays are constructed (in this work, we selected strength $t = 2$ for all such CAs), the rows of which are subsequently combined based on a meta CA to obtain a resulting combined CA.

To illustrate this approach, we provide an excerpt of three layer IPMs in Listing 2 and the first rows of the resulting seed CAs in comma-separated values (CSV) format in Listing 3. The following paragraphs provide further details on the subsequent steps.

Listing 2: Excerpts of three IPMs used to generate seed CAs

```
BTLE_IPM = {
    'access_addr': ["UNCHANGED", "~0x8e89bed6", "~0xffffffff",
                    "~0x00000000"],
    'crc': ["UNCHANGED"]
}

BTLE_DATA_IPM = {
    'LLID': ["~0", "~1", "~2", "~3", "UNCHANGED"],
    'SN': ["~0", "~1", "UNCHANGED"],
    'NESN': ["~0", "~1", "UNCHANGED"]
}

BTLE_EMPTY_PDU_IPM = {
    'LLID': ["UNCHANGED", "~0", "~1", "~2", "~3"],
    'Length': ["~0x00", "RECALCULATE", "~LARGER",
               "~SMALLER", "~0xff"]
}
```

Listing 3: Excerpts of three seed CAs

```
access_addr,crc
--------
UNCHANGED,UNCHANGED
~0x8e89bed6,UNCHANGED
~0xffffffff,UNCHANGED

LLID,SN,NESN
--------
~0,UNCHANGED,UNCHANGED
UNCHANGED,~0,UNCHANGED
UNCHANGED,UNCHANGED,UNCHANGED

LLID,Length
--------
UNCHANGED,~0x00
UNCHANGED,RECALCULATE
UNCHANGED,~LARGER
~0,RECALCULATE
~2,RECALCULATE
```

At this point, we have obtained an assembly of layer/seed CAs, which provide $t = 2$ coverage for the subcomponents of the composite system. Those seed arrays now have to be combined into a single array. To this end, we first determine the number of rows in each of the seed CAs (in the example above, the first two array excerpts have three rows, while the final array has five rows). Using this information, a meta IPM is constructed wherein each parameter represents a layer CA and takes values between 1 and the number of rows. Indices that refer to rows in the layer CA containing negative values are additionally prefixed with the "∼" symbol in order to prevent masking effects across layers. Listing 4 shows the meta IPM resulting from the sample seed arrays in Listing 3.

Listing 4: Example meta IPM

```
META_IPM = {
    'BTLE_ROWS': ["0", "~1", "~2"],
    'BTLE_DATA_ROWS': ["~0", "~1", "2"],
    'BTLE_EMPTY_PDU_ROWS': ["~0", "1", "~2", "~3", "~4"]
}
```

This meta IPM is subsequently used to generate a meta CA at strength $t = 2$, each row of which references a single row in each of the layer CAs. The meta CA then covers all $t = 2$ combinations of rows from the seed CAs that represent the layers of the current packet. Listing 5 shows an excerpt of a meta CA for our running example.

Listing 5: Excerpt of example meta CA

```
BTLE_ROWS,BTLE_DATA_ROWS,BTLE_EMPTY_PDU_ROWS
--------
0,~1,1
~1,2,1
0,2,~2
0,2,~3
0,2,~4
```

By combining the contents of these referenced rows, we obtain a single combined CA that is suitable to be used in our testing flow as described above. Listing 6 shows the first three rows resulting from combining the seed CAs according to the meta CA given in Listing 5. The resulting CA offers not only guaranteed input space coverage within each layer, but also across layers, to a degree that can be adjusted by increasing the strength (at the cost of larger test sets and thus additional execution time).

Listing 6: Excerpt of resulting combined CA

```
BTLE_access_addr,BTLE_crc,BTLE_DATA_LLID,BTLE_DATA_SN,
    ↪ BTLE_DATA_NESN,BTLE_EMPTY_PDU_LLID,
    ↪ BTLE_EMPTY_PDU_Length
--------
UNCHANGED,UNCHANGED,UNCHANGED,~0,UNCHANGED,
    ↪ UNCHANGED,RECALCULATE
~0x8e89bed6,UNCHANGED,UNCHANGED,UNCHANGED,
    ↪ UNCHANGED,UNCHANGED,RECALCULATE
UNCHANGED,UNCHANGED,UNCHANGED,UNCHANGED,
    ↪ UNCHANGED,UNCHANGED,~LARGER
```

4.4 Oracles

In order to determine whether the crafted inputs submitted by our approach crashed the SUT, we implement two oracles. The first uses a secondary NRF52840 dongle with the custom GreyHound firmware to receive packets sent by the peripheral device SUT. If no packets are detected for over a second, the oracle marks the test as failed; if the initial state is reached without such a timeout, the test is deemed a success. This oracle aims to detect situations in which the peripheral device becomes unreachable for a longer period than usual. In this case, the peripheral might reboot itself, becoming responsive again after a few seconds, but could also end up in a state in which it requires a manual reboot by cycling the power provided via USB (see Section 5).

Since most development boards we tested have a serial port available, we decided to use it for an additional oracle. If the device under test has a serial port, we connect to it and listen for output from the peripheral device. The received serial output is checked for specific keywords such as "trace", "crash" and "dump". If one of those words is detected, the oracle marks the test as failed.

The first oracle is generally usable for every BLE device. However, the generality of the approach is also a potential problem, since it can lead to false positives if the device merely requires more time than usual to enable advertisements again after a disconnect event.

The second oracle is less general, requiring the device under test to have a serial port and additionally assumes that there is debug output containing specific keywords. Most toolchains of the devices we tested provide settings to enable debug output, which lead to the devices giving more detailed information over the serial connection. During our experiments, we only encountered one device that did not produce serial output.

The oracles we developed are capable of identifying crashes that either lead to a recognizable crash dump being

SoC Vendor	SoC Model	SDK Versions	Sample App
Texas Instruments	CC2640R2	3.30.00.20, 5.30.00.03	project zero
Espressif Systems	ESP32	4.1, 5.0	nimble/bleprphrl
Nordic Semiconductors	nRF52	15.3.0, 17.0.1	ble_app_gatts_c
Bouffalolab	bl602	AI-Thinker WB2 beta v1.1.8	ble_slave
WCH	CH582M	MounRiver Studio community v1.50	Peripheral
Hi-Link	W801	a93b517	wm_ble_client_api_multi_conn_demo
Realtek	RTL8720DN	Realtek Ameba Boards 3.1.6	BLEBatteryService
Silicon Labs	BG22	simplicity studio SV5.7.1.1	bluetooth_controlling_led_from_smartphone
Ambiq	Apollo3	sparkfun apollo3 boards v2.2.1	LED_Button
MacroGiga Electronics	MG126	seeed SAMD boards 1.8.4	analog_output

Table 1: Tested devices and software versions.

transmitted via a serial port or to prolonged inactivity of BLE transmissions of the SUT. False negatives may occur if the SUT exhibits incorrect behaviour due to received packets but does not become unresponsive and does not output a crash dump (or outputs it in a manner that is recognized by our oracle). Examples of such cases could be that the peripheral accepts an encryption key that is set to only zeros, which would compromise the privacy of the data exchange, or scenarios in which the peripheral accepts invalid authentication parameters, giving an attacker access to its services.

False positives of our first oracle that listens for transmitted packets of the peripheral could occur if the oracle dongle itself crashes, which is trivial to detect, or the SUT takes significantly longer than expected to transmit packets, which is arguably a denial-of-service condition by itself. False positives of the second oracle, which monitors the SUT's serial output, could occur if the SUT's output contains certain keywords indicating a crash dump without actually having crashed.

To exclude these cases, we constructed automated reproducer scripts for all identified issues, removing cases that did not reliably lead to an unreachable SUT and thus eliminating false positives before reporting them to affected vendors.

5 Setup

Our test setup consists of two NRF52840 dongles running the custom firmware of the GreyHound framework and a laptop running Ubuntu 22.0.4 as the host. The dongles are connected to the host via USB, with one being used for sending and receiving BLE packets from the peripheral device under test, while the other passively listens for packets transmitted by the peripheral device, thus acting as one of the test oracles. Figure 5 depicts a simplified version of our setup.

Submitted test cases and their results are stored in a MongoDB instance. A database entry consists of the CA row used for the test, the packet history recorded during the test, the first received response after the test, the results of both oracles, as well as the current state and path to test. Additionally, we store messages received via the serial port of the BLE peripheral device for post-analysis.

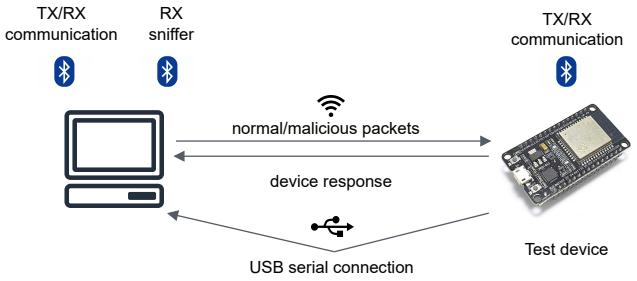


Figure 5: Overview of the test setup.

During testing, we encountered issues with hardware reliability. Some of the evaluated BLE peripheral devices as well as the NRF52840 dongle used for BLE communication crashed or became unresponsive at times, possibly due to race conditions, since we could not reliably reproduce them. We thus implement an automated way of resetting the power to the devices based on relays that interrupt the USB power supply for a few seconds. These relays are controlled via an ESP32 micro-controller that can be communicated with using a REST API. By power cycling the USB devices in this manner, we can reliably reset them to a working and responsive state and repeat the test. Because of the sending dongle occasionally becoming unresponsive, we used a separate receiving dongle as oracle to prevent false positive results in these cases. A test is only logged if a malicious packet was actually sent. Otherwise, the USB devices are reset and the test is repeated.

As the evaluated devices exhibit wide variations in their performance, commonly requiring more than a day to execute a full test set, we parallelize our setup by replicating it four times in total (i.e. using four host computers, each hosting a SUT device and two NRF52840 dongles).

5.1 Target Devices

As we wanted to test the BLE stack implementation of different BLE chips, we needed to source a variety of BLE chips, preferably integrated in development boards. Development boards typically provide easy access to the input and output pins of the chip and ship in a plug-and-play package, integrating the necessary circuitry around the chip such as power supply and serial connections. We further included chips that had previously been tested using the GreyHound fuzzer in order to make a performance comparison. For these devices, we additionally evaluated firmware versions that were newer than those analyzed in the earlier work in order to identify whether the issues that were published as part of the SweynTooth set of vulnerabilities had been successfully remediated. Table 1 shows details of the SDKs and sample applications of the chips we tested.

5.2 Challenges

In addition to the previously mentioned issue of SUTs and dongles becoming unresponsive or crashing, we encountered specific challenges when testing certain devices.

Most notably, the Arduino SDKs of the Apollo3 and MG126 chips were still under development at the time of writing, with their BLE implementations remaining incomplete. This resulted in some unexpected behaviour during testing; devices did not traverse the complete state machine when they reached a state that requires functionality that was not yet implemented, leading to the low number of executed tests shown in Table 2.

For some devices (i.e. ch582m and w801), the onboard USB connectivity was only suitable for flashing new firmware, while capturing the serial output required an additional adapter. In these cases, we utilized an FT232RL USB to serial adapter to enable the use of our second oracle.

In other cases, the testing process was extremely slow due to some states not being reliably reachable via certain paths. This occurred while testing the MG126, CC2640R2 and ESP32 chips. Throughout our experiments, we implemented a routine that identified when paths did not reach the state to test for the last 400 seconds and excluded them from our results, instead proceeding with the next available path.

6 Evaluation

Due to the instability of some of the tested devices, we validated test cases that were marked as failed by either of our oracles using a simple reproducer script that replays the same inputs and records the results. Only those test cases that were deemed reproducible (and therefore not false positives) were selected for further analysis. The following sections present the analyzed results and compare them to those of the previous work based on fuzzing [14].

6.1 Results

While executing the tests, we witnessed crashes and availability issues in most of the devices we tested. Table 2 shows a summary of the reproducible unique faults we encountered in the individual chips and their respective SDK versions. We consider a fault to be unique if the fields/parameters that trigger it have not been identified as part of the trigger of another fault or if the values of overlapping fields/parameters are considered to be semantically different according to the BLE specification or an expert security analyst's opinion (e.g. if a length field's value changes from 1317 to 1318, it would commonly not be considered to be semantically different, while a difference in the packet's `TYPE` code would indicate a separate fault). The column *TO* (short for timeout) lists the number of test cases that led to the SUT becoming unresponsive for at least four seconds, but returning to normal operation after at most two minutes without requiring further intervention. The column *URWF* (Unavailable but Recoverable With Code Fix) details the number of cases where a SUT became unresponsive and required a restart when using the unmodified example code. In these cases, the BLE stack was still properly responding to GAP (General Access Profile) and HCI (Host Controller Interface) commands from the application and despite advertisements not being re-enabled automatically, it was possible to modify the application code to reactivate them. The column *URN* (Unavailable Reset Needed) shows the number of cases in which the SUT became unavailable after receiving a malicious packet and required a reset to function again. When trying to modify the example code to reactivate advertisements in these cases, the BLE functions returned error codes or behaved unexpectedly. Finally, the column *Dump* shows how often a core dump was sent from the SUT to the host PC via serial connection, followed by a restart of the SUT.

Most timeouts and unavailability issues occurred in the *initiating* state, after a connection request with invalid parameters (i.e. a malicious packet) was sent to the peripheral device. An example of such a packet is given in Listing 7, with the invalid value being `0x00000000` for the `access address` (AA) field, a parameter used for handling collisions between peers sharing the same physical radio channel. When sending such a packet to an ESP32 flashed with the `bleprphrl` example application in SDK version 5.0, the BLE stack of the ES32 logs a connection event despite the central not being connected. Because the slave is in a state that indicates that a valid connection has been established, advertisements are not enabled again, which renders the device unusable until the device is reset. Additionally, when trying to re-enable advertisements manually, the following error is logged: "error enabling advertisement; rc=6 (BLE_ERR_PINKEY_MISSING=)". However, a missing PIN error is not consistent with this state in the BLE protocol and can not occur at this point under normal circumstances. This behavior suggests that the BLE

Model	Version	TO	URWF	URN	Dump	Total
CC2640R2	3.30.00.20	1	4	0	0	1,309
	5.30.00.03	0	3	0	0	1,188
ESP32	4.1	0	0	1	1	2,115
	5.0	0	0	2	0	1,910
nRF52	15.3.0	0	0	0	0	1,309
	17.0.1	0	0	0	0	1,309
bl602	AI-Thinker WB2 beta v1.1.8	0	0	1	1	2,985
CH582M	MounRiver Studio community v1.50	0	0	0	0	1,184
W801	a93b517	0	0	1	0	1,132
RTL8720DN	Realtek Ameba Boards 3.1.6	4	0	1	0	2,664
BG22	simplicity studio v5.7.1.1	0	0	0	0	1,282
Apollo3	sparkfun apollo3 boards v2.2.1	0	0	1	0	594
MG126	seeed SAMD boards 1.8.4	2	0	0	0	373

Table 2: Identified unique faults per device and software version. The column TO refers to the number of Timeouts, URWF to Unavailable but Recoverable With Code Fix, URN means Unavailable Reset Needed, and Dump indicates a crash dump output by the SUT. The Total column lists the number of executed test cases. Three issues identified in the old version of the CC2640R2 were still present in the later version. For ESP32, one issue was fixed in version 5.0, but a new URN was introduced. In total, 19 unique faults were identified.

stack is in unknown state and currently malfunctioning. When sending the same packet to an ESP32 flashed with the same application in version 4.1, trying to manually re-enable advertisements leads to HCI desync errors and eventually a core dump with controller restart, which indicates an issue with memory management.

Listing 7: Example of a malicious packet

```
###[ BT4LE ]###
  access_addr= 0x8e89bed6
  crc = None
###[ BTLE advertising header ]###
     RxAdd = public
     TxAdd = public
     RFU = 0
     PDU_type = CONNECT_REQ
     unused = 0
     Length = None
###[ BTLE connect request ]###
        InitA = EB:BC:B4:85:A8:B2
        AdvA = 30:ae:a4:87:9e:de
        AA = 0x00000000
        crc_init = 0x179a9c
        win_size = 0x2
        win_offset= 0x1
        interval = 0x10
        latency = 0x0
        timeout = 0x64
        chM = 0x1fffffffff
        SCA = 0
        hop = 5
```

The core dump issued by the bl602 chip was triggered in the *length request* state, which is part of the initial central peripheral setup procedure and requires the field `max_rx_bytes` to be set to 0. The Apollo3 exclusively became unresponsive in the *pairing request* state, after sending a malicious pairing request to the peripheral device, containing an invalid value for the authentication method.

In addition to the quantitative assessment, we also used the available JTAG ports on the individual SUTs to gather more insight into the identified errors. The results of our analysis indicate that most errors lead to similar behavior. After receiving a malicious connection request, the peripheral devices fail to correctly validate the connection parameters and process the packet as a valid connection request. Since at least one of the parameters is not valid, the BLE central can not communicate with the peripheral device anymore, while the peripheral device is stuck waiting for events from the BLE stack. In this state, the peripheral devices are practically unusable until they are reset, as there is no valid connection established and other devices can not discover them because advertisements can not be re-enabled. The exact impact varies by SUT depending on the field and value of the malicious packet. This is manifested by BLE functions returning different error codes when trying to reactivate advertisements and the sequence of GAP events.

According to the specification, BLE implementations should follow the rule *EffectiveConnectionInterval = ConnectionInterval * (1 + Slave Latency)* [18], and *EffectiveConnectionInterval* must not exceed a maximum of 16 seconds if no packets are received. In our tests, most devices did not drop the connection or were not able to reactivate advertisements after receiving a malicious packet. Simultaneously, applications received a reduced number of GAP events from the BLE stack compared to normal connections. This could indicate an issue in the parameter validation and state transition of the GAP implementation, which could also lead to further corruption of the BLE stack when invalid parameters

Model	Version	Tests		Issues		Anomalies
		GH	CST	GH	CST	GH
CC2640R2	5.30.00.03	2,129	**1,188**	0	**3**	0
ESP32	5.0	5,979	**1,910**	0	**2**	1
nRF52	17.0.1	8,790	**1,309**	**0**	**0**	1
bl602	AI-Thinker WB2 beta v1.1.8	7,233	**2,985**	1	**2**	2
CH582M	MounRiver Studio community v1.50	4,237	**1,184**	**0**	**0**	2
W801	a93b517	**879**	1,132	0	**1**	1
RTL8720DN	Realtek Ameba Boards 3.1.6	10,153	**2,664**	0	**5**	2
BG22	simplicity studio v5.7.1.1	8,016	**1,282**	**0**	**0**	0
Apollo3	sparkfun apollo3 boards v2.2.1	7,935	**594**	**1**	**1**	0
MG126	seeed SAMD boards 1.8.4	9,993	**373**	0	**2**	0

Table 3: Comparison of combinatorial security testing (CST) versus GreyHound (GH) fuzzer on current SDK versions. The smallest number of tests for a particular SUT as well as the greatest number of identified issues is highlighted in bold. As discussed in Section 6.2, anomalies are not assessed by the CST approach.

are processed. The MG126 and RTL8720DN were the only devices that were able to automatically recover in some of these cases; they were, however, often unreachable for over a minute, meaning that these inputs led to Denial of Service (DoS) conditions.

Most of the identified faults manifest themselves as timeouts and may be exploited to perform DoS attacks. They are either caused by the device firmware itself or the example application executed on the SUT. Inputs leading to devices transmitting core dumps indicate memory access violations, which may be exploited to gain control over the SUT or leak cryptographic key material or other secrets.

6.2 Comparison

To evaluate the performance of our combinatorial testing approach, we executed the fuzzer as used in the original work [14] against our suite of SUTs. The effectiveness of fuzzing partially depends on the available execution time. As the maximum time required to test a SUT with our approach was just under one day, we ran the fuzzer for 24 hours and then compared its output to that of our method. A summary of the results is given in Table 3.

The oracle of the GreyHound fuzzer not only checks for timeouts and crashes (listed under *Issues*), but also for deviations from the BLE specification (e.g. BLE devices responding to packets in states in which they should not or accepting invalid packets), which the authors call anomalies. Unfortunately, manual analysis indicates that this oracle produces a significant number of false positives. We thus relied on the oracles described in Section 4.4 and opted to not assess anomalies for our approach, leading to their absence from Table 3. Furthermore, due to the aforementioned reliability issues, we created a script that aims to replay inputs indicated by the GreyHound fuzzer as leading to faults and excluded those that were not reproducible.

The column *Tests* indicates the total number of test cases executed. While the execution rate of the fuzzer is higher than that of our approach (often caused by the time required to identify unreliable paths as noted in Section 5.2), it proved unreliable for some devices such as the W801, leading to a lower number of iterations. Additionally, the NRF52 dongle became unstable at times; while our approach implements a reset functionality (see Section 5), the original work does not include this feature.

Overall, the CST approach presented in this work identified more faults than the GreyHound fuzzer in current firmware versions, but did not find any anomalies.

Re-testing

The original work identified several issues that the authors subsequently communicated to vendors. The aforementioned results indicate that most of these bugs have been addressed in current firmware versions. To provide a comparison that is closer to the circumstances of the earlier work, we re-tested some of the devices using old versions offered by vendors.

Table 4 summarizes the results of our approach compared to the results reported by the original work, which always reported results for 1,000 tests. While our method identifies a greater total number of faults, results vary across different devices and types of triggers; most notably, our approach currently makes no attempt to alter the sequence of protocol messages, while the GreyHound fuzzer does offer this functionality. Accordingly, we were unable to identify some of the faults found by the earlier work.

For the ESP32 SUT, one of the identified issues overlaps with the earlier work (the device crashes when receiving a connection request with the `chM` parameter set to `0`), while the second fault found using GreyHound relies on an alteration of

the sequence. In contrast, the second issue identified by our approach is triggered merely through message parameters.

		Tests		Issues		Anomalies
Model	Version	GH	CST	GH	CST	GH
CC2640R2	3.30.00.20	1,000	1,309	2	**5**	1
ESP32	4.1	1,000	2,115	**2**	**2**	0
nRF52	15.3.0	1,000	1,309	**0**	**0**	1

Table 4: Comparison of CST versus GreyHound (GH) fuzzer applied to old firmware versions.

6.3 Lessons Learned

Applying CST to a complex protocol such as BLE occasionally proved challenging, which led us to adapt our models. Earlier models that did not contain markers for invalid values were sometimes unable to detect errors that required only a single value to be triggered due to masking effects. An interesting observation is that all errors identified through our approach only required one parameter to assume a specific value (although the parameter and value varied across SUTs and errors); in other words, even a CA of strength $t = 1$ would have been able to trigger these issues. Since the GreyHound fuzzer was not able to find these faults, it might indicate that the fuzzer tends to alter too many fields and therefore suffers from masking effects.

Another common issue was that higher strengths and covering of interactions between all layers of a packet tends to lead to very large test sets. It might be beneficial to only cover interactions between layers if they have fields that directly influence each other; for example, this includes length or type fields. This could be achieved through the use of variable-strength CAs [25].

Finally, we performed internal experiments to introduce known combinatorial sequence constructions with the goal of varying the ordering of packets. However, the resulting number of test cases was deemed infeasible for practical use. Most issues discovered using GreyHound that rely on reordering of packets occurred in the pairing or encryption processes (which are also the most complex aspects of the protocol and often rely on information exchanged in previous packets). This indicates that it may prove beneficial to only mutate orderings in these states and leave sequences of packets in later states intact. In contrast to variable-strength CAs, we are not aware of existing work that would result in variable-strength sequence coverings.

7 Responsible Disclosure

We notified the vendors of all affected devices 90 days in advance for the purpose of facilitating a coordinated responsible disclosure process. At the time of writing, Espressif and Realtek have been the only two companies that responded. Espressif acknowledged the issues and released fixes for all of the issues we found in all affected software versions and will assign CVEs in the near future. Espressif was the only contacted vendor running a dedicated bug bounty program for software vulnerabilities.

As the 90 day disclosure period has passed, we have uploaded proof of concept code to reproduce the vulnerabilities we found. The example code, together with MongoDB database exports of the test results, is available for download at https://zenodo.org/records/14647110.

8 Conclusion

This work presented a combinatorial security testing approach for modeling and testing Bluetooth LE communications utilizing a modified GreyHound fuzzing framework. Our method provides combinatorial coverage guarantees regarding the input space of wireless devices while minimizing the required number of test cases, leading to the discovery of 19 distinct issues across 13 device/firmware combinations.

A comparative experimental evaluation against the earlier work points towards increased fault detection capabilities of our approach. This is also confirmed when re-testing older firmware versions that contain the bugs uncovered using GreyHound. However, issues that require a modified order of packets to be triggered are currently out of scope of our work. Additionally, the execution rate of our implementation is hampered by the slow identification of unreliable paths.

9 Future Work

A logical extension of our work is the addition of combinatorial sequence modeling and Sequence Covering Arrays (SCAs) [10], an approach that has successfully been used to test protocols [15] and offers additional coverage guarantees. To avoid the problem of combinatorial explosion when generating test sets, it might be beneficial to additionally add a guiding heuristic with adjustable weights, allowing testers to focus their resources on the most interesting components first.

Acknowledgments

This work has received funding from the European Union under the Horizon Europe programme (grant agreement 101168438) as well as from the Swiss State Secretariat for Education, Research and Innovation and UK Research and Innovation (grant number 10126241). SBA Research (SBA-K1 NGC) is a COMET Center within the COMET - Competence Centers for Excellent Technologies Programme and funded by BMIMI, BMWET, and the federal state of Vienna. The COMET Programme is managed by FFG.

References

[1] Bluetooth specifications and documents. https://www.bluetooth.com/specifications/specs/?keyword=core+specification, 2014. [Online; accessed Jan. 15th, 2024].

[2] Daniele Antonioli, Nils Ole Tippenhauer, and Kasper Rasmussen. Bias: Bluetooth impersonation attacks. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 549–562. IEEE, 2020.

[3] Daniele Antonioli, Nils Ole Tippenhauer, and Kasper Rasmussen. Key negotiation downgrade attacks on Bluetooth and Bluetooth Low Energy. *ACM Trans. Priv. Secur.*, 23(3), July 2020.

[4] Arup Barua, Md Abdullah Al Alamin, Md Shohrab Hossain, and Ekram Hossain. Security and privacy threats for Bluetooth Low Energy in IoT and wearable devices: A comprehensive survey. *IEEE Open Journal of the Communications Society*, 3:251–281, 2022.

[5] Dor Zusman Ben Seri, Gregory Vishnepolsky. Bleedingbit, the hidden attack surface within ble chips. https://media.armis.com/PDFs/wp-blueborne-bluetooth-vulnerabilities-en.pdf, 2023. [Online; accessed Jan. 15th, 2024].

[6] Gregory Vishnepolsky Ben Seri. The dangers of Bluetooth implementations: Unveiling zero day vulnerabilities and security flaws in modern Bluetooth stacks. https://media.armis.com/PDFs/wp-blueborne-bluetooth-vulnerabilities-en.pdf, 2023. [Online; accessed Jan. 15th, 2024].

[7] Josip Bozic, Bernhard Garn, Ioannis Kapsalis, Dimitris Simos, Severin Winkler, and Franz Wotawa. Attack pattern-based combinatorial testing with constraints for web security testing. In *2015 IEEE International Conference on Software Quality, Reliability and Security*, pages 207–212. IEEE, 2015.

[8] Kevin Burr and William Young. Combinatorial test techniques: Table-based automation, test generation and code coverage. In *Proc. of the Intl. Conf. on Software Testing Analysis & Review*, 1998.

[9] Matthias Cäsar, Tobias Pawelke, Jan Steffan, and Gabriel Terhorst. A survey on Bluetooth Low Energy security and privacy. *Computer Networks*, 205:108712, 2022.

[10] Yeow Meng Chee, Charles J Colbourn, Daniel Horsley, and Junling Zhou. Sequence covering arrays. *SIAM Journal on Discrete Mathematics*, 27(4):1844–1861, 2013.

[11] Microsoft Corporation. Pairwise independent combinatorial testing. https://github.com/microsoft/pict, 2022. [Online; accessed April. 17th, 2024].

[12] Matheus E Garbelini, Vaibhav Bedi, Sudipta Chattopadhyay, Sumei Sun, and Ernest Kurniawan. BrakTooth: Causing havoc on Bluetooth link manager via directed fuzzing. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 1025–1042, 2022.

[13] Matheus E Garbelini, Chundong Wang, and Sudipta Chattopadhyay. Greyhound: Directed greybox Wi-Fi fuzzing. *IEEE Transactions on Dependable and Secure Computing*, 19(2):817–834, 2020.

[14] Matheus E. Garbelini, Chundong Wang, Sudipta Chattopadhyay, Sun Sumei, and Ernest Kurniawan. SweynTooth: Unleashing mayhem over Bluetooth Low Energy. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 911–925. USENIX Association, July 2020.

[15] Bernhard Garn, Dimitris E Simos, Feng Duan, Yu Lei, Josip Bozic, and Franz Wotawa. Weighted combinatorial sequence testing for the TLS protocol. In *2019 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 46–51. IEEE, 2019.

[16] Bernhard Garn, Jovan Zivanovic, Manuel Leithner, and Dimitris E Simos. Combinatorial methods for dynamic gray-box SQL injection testing. *Software Testing, Verification and Reliability*, 32(6):e1826, 2022.

[17] Linghuan Hu, W Eric Wong, D Richard Kuhn, Raghu N Kacker, and Shuo Li. CT-IoT: a combinatorial testing-based path selection framework for effective IoT testing. *Empirical Software Engineering*, 27:1–38, 2022.

[18] Texas Instruments. Overview - BLE-Stack user's guide for Bluetooth 4.2 3.01.01.00 documentation. https://software-dl.ti.com/lprf/simplelink_cc2640r2_latest/docs/blestack/ble_user_guide/html/ble-stack-3.x/gap.html, 2016. [Online; accessed April. 17th, 2024].

[19] Dario Izzo. PyGMO and PyKEP: Open source tools for massively parallel optimization in astrodynamics (the case of interplanetary trajectory optimization). In *Proceedings of the Fifth International Conference on Astrodynamics Tools and Techniques, ICATT*. sn, 2012.

[20] Ludwig Kampel, Bernhard Garn, and Dimitris E Simos. Combinatorial methods for modelling composed software systems. In *2017 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 229–238. IEEE, 2017.

[21] Kristoffer Kleine and Dimitris E Simos. Coveringcerts: Combinatorial methods for X.509 certificate testing. In *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 69–79. IEEE, 2017.

[22] D Richard Kuhn, Renee Bryce, Feng Duan, Laleh Sh Ghandehari, Yu Lei, and Raghu N Kacker. Combinatorial testing: Theory and practice. *Advances in Computers*, 99:1–66, 2015.

[23] Manuel Leithner, Bernhard Garn, and Dimitris E Simos. HYDRA: Feedback-driven black-box exploitation of injection vulnerabilities. *Information and Software Technology*, 140:106703, 2021.

[24] Marcel Maehren, Philipp Nieting, Sven Hebrok, Robert Merget, Juraj Somorovsky, and Jörg Schwenk. TLS-Anvil: Adapting combinatorial testing for TLS libraries. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 215–232, 2022.

[25] Sebastian Raaphorst, Lucia Moura, and Brett Stevens. Variable strength covering arrays. *Journal of Combinatorial Designs*, 26(9):417–438, 2018.

[26] Stuart C Reid. An empirical analysis of equivalence partitioning, boundary value analysis and random testing. In *Proceedings Fourth International Software Metrics Symposium*, pages 64–73. IEEE, 1997.

[27] R Rohith, Minal Moharir, G Shobha, et al. SCAPY - a powerful interactive packet manipulation program. In *2018 International Conference on Networking, Embedded and Wireless Systems (ICNEWS)*, pages 1–5. IEEE, 2018.

[28] D. E. Simos, B. Garn, J. Zivanovic, and M. Leithner. Practical combinatorial testing for XSS detection using locally optimized attack models. In *2019 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 122–130, 2019.

[29] D. E. Simos, R. Kuhn, A. G. Voyiatzis, and R. Kacker. Combinatorial methods in security testing. *Computer*, 49(10):80–83, 2016.

[30] Dimitris E. Simos, Josip Bozic, Bernhard Garn, Manuel Leithner, Feng Duan, Kristoffer Kleine, Yu Lei, and Franz Wotawa. Testing TLS using planning-based combinatorial methods and execution framework. *Software Quality Journal*, 27(2):703–729, June 2019.

[31] Dimitris E Simos, Jovan Zivanovic, and Manuel Leithner. Automated combinatorial testing for detecting SQL vulnerabilities in web applications. In *2019 IEEE/ACM 14th International Workshop on Automation of Software Test (AST)*, pages 55–61. IEEE, 2019.

[32] Jianliang Wu, Yuhong Nan, Vireshwar Kumar, Dave Jing Tian, Antonio Bianchi, Mathias Payer, and Dongyan Xu. BLESA: Spoofing attacks against reconnections in Bluetooth Low Energy. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*, 2020.

[33] Yue Zhang, Jian Weng, Rajib Dey, Yier Jin, Zhiqiang Lin, and Xinwen Fu. Breaking secure pairing of Bluetooth Low Energy using downgrade attacks. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 37–54, 2020.

Resource Multiplexing in Tuning and Serving Large Language Models

Yongjun He
ETH Zürich
yongjun.he@inf.ethz.ch

Haofeng Yang
ETH Zürich
yanghao@student.ethz.ch

Yao Lu
National University of Singapore
yao@nus.edu.sg

Ana Klimović
ETH Zürich
aklimovic@ethz.ch

Gustavo Alonso
ETH Zürich
alonso@inf.ethz.ch

Abstract

Large language models (LLMs) have been increasingly adopted in a variety of application scenarios. However, in spite of the high demand for both tuning and inference, GPUs are often underutilized because they are devoted to a single task. A common argument for single-purpose deployments is the need to meet strict service-level objectives (SLOs). As LLM workloads become more complex, there are, indeed, significant challenges in achieving high utilization while still guaranteeing the necessary low latency. In this paper, we present LLMStation, a flexible spatial-temporal multiplexing and scheduling system for concurrent LLM fine-tuning and inference. LLMStation adopts several novel approaches, including a new iteration-level multitasking scheduling mechanism, an Autograd engine that transforms a tuning task into a suspendable pipeline, and an inference engine capable of batching inference and tuning requests. Our evaluation shows that LLMStation delivers 1.38× to 14.77× the throughput of state-of-the-art systems while meeting inference latency SLOs. These performance gains remain under various setups and workloads, proving LLMStation to be an effective tool to increase the efficiency of LLM deployments.

1 Introduction

GPUs are widely used for AI applications to maximize performance per watt. Due to their high cost [63] and power consumption, operating GPUs at high utilization is critical for minimizing the total cost of ownership and for making optimal use of limited power budgets [50, 58, 59].

However, even resource-intensive LLM workloads struggle to keep GPU hardware highly utilized as applications like LLM inference consist of phases with different resource requirements, which often leave some parts of the GPU idle for a significant fraction of time [30, 50, 52, 58, 59]. For example, during the memory-bound decoding phase of serving a Llama3-8B model on A100 GPUs, Microsoft reports less than 10% GPU compute utilization [16, 33].

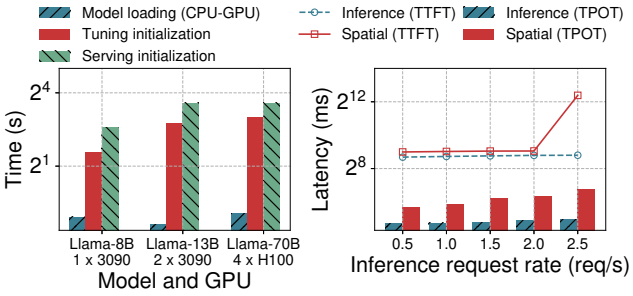


Figure 1: Measured context switch overhead (left). Performance interference between LLM tuning and serving co-located on the same GPU (right).

A natural way to improve resource utilization and cost is to consolidate models and requests on fewer GPUs rather than dedicating a single GPU for each task or model, with time-sharing and/or spatial sharing [23, 43, 52, 58, 59]. Time-sharing the GPU for workloads involving multiple models or tasks can lead to high context-switching overhead, which becomes a critical bottleneck [26, 50]. Figure 1(left) shows the time taken to load models to the GPU memory and initialize the execution engines for inference or fine-tuning (e.g., vLLM [36] and torchtune [54]). As the experiments show, this can take seconds to minutes. In addition to context-switching overhead issues, temporal sharing systems for mixed tasks such as FineInfer [26] may not be able to meet strict latency SLO requirements when models are relatively large, and waste computation resources in the memory-bound decoding phase.

The alternative of spatial-sharing GPUs also faces significant challenges. LLMs require large GPU memory, up to hundreds of GBs, thus preventing strict isolation solutions from being usable, such as the Nvidia virtual GPU [3] which divides the GPU memory into independent small chunks. When relaxing strict isolation, there can be significant interference between processes. Figure 1(right) highlights the performance degradation when tuning and serving tasks on Llama-3.1-8B [22] are co-located on an RTX 3090, with p99

Time to First Token (TTFT) and p99 Time Per Output Token (TPOT) increasing to up to 13.2× and 3.6×, respectively.

To increase the efficiency of LLM systems and reduce resource consumption, in this paper we explore how to share a GPU between LLM parameter-efficient fine-tuning (PEFT) and inference/serving in a spatial-temporal manner, as PEFT creates new opportunities for multiplexing memory and computation resources by sharing base models [26, 29, 44] and co-execution of memory-bound and compute-bound operations [26, 44]. In engineering practice, PEFT is frequently conducted to adapt base models to various tasks and is also commonly performed continuously [10, 13, 15, 31] after deployment to address potential data drift. This scenario is increasingly relevant for applications using on-premise infrastructure, where fine-tuning and model serving compete for limited resources. This is particularly the case when the inference demand fluctuates [21, 50].

To cope with the challenges described above, we propose LLMStation, which employs a novel strategy for spatial-temporal multiplexing. The key insight is that LLM tuning and serving tasks have distinct workload patterns. By spatially batching and temporally reordering tasks, LLMStation simultaneously achieves controllable interference between processes, rapid context switching, maximized LLM fine-tuning throughput, and LLMs serving within the SLO limits.

LLMStation has been implemented in 3k lines of code. It includes a set of novel techniques including an iteration-level multitasking scheduling mechanism, an Autograd engine transforming a tuning task into a pre-emptable pipeline, an inference engine that batches inference requests and tuning requests, and a memory manager that manages base models, adapters, and intermediate states between different tasks. A comprehensive evaluation on Nvidia RTX 3090 and H100 GPUs shows that LLMStation achieves up to 14.77× the fine-tuning throughput of state-of-the-art systems, while maintaining inference latency SLOs. Microbenchmarks confirm that scheduling and context-switching overheads remain minimal. Despite that focusing on the specific but highly relevant use case of LLM tuning and serving, the ideas behind LLMStation can be generalized and the design provides the foundation for future research into GPU virtualization and resource multiplexing to bypass the existing hardware and driver limitations.

2 Background and Motivation

In this section, we provide background on PEFT, LLM inference, and GPU resource multiplexing to motivate our work.

2.1 Parameter-Efficient Fine-Tuning

With the growing adoption of LLMs, fine-tuning becomes more and more frequent in order to adapt base models to different data and downstream tasks [10, 13, 51]. As shown in Figure 2, more than a hundred thousand of adapters were uploaded to Hugging Face in 2024. Even after models are deployed for inference, it is common and important to continue fine-tuning them to adapt to potential data drift [10, 13, 15, 31], as reflected in the continuously changes in model weights for these Hugging Face model repositories.

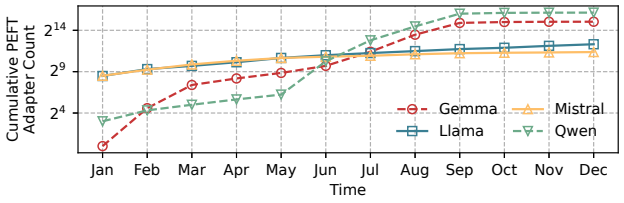


Figure 2: The number of PEFT adapters based on popular open-source model series uploaded to Hugging Face in 2024.

To make fine-tuning cost-effective, various PEFT techniques [29, 37, 39, 41, 42] have been proposed to reduce its memory requirements while retaining comparable statistical performance. The core idea of PEFT technique is to freeze most parameters of the pre-trained model and only update a small part of the model parameters (i.e., adapters), which can greatly reduce the memory footprint of the optimizer state during fine-tuning. A representative PEFT technique is Low-Rank Adaptation (LoRA) [29] that freezes the whole pre-trained model and only fine-tunes extra rank decomposition matrices injected in each transformer layer. The output of a LoRA layer can be formulated as:

$$h(x) = Wx + \Delta Wx = Wx + BAx,$$

where $W \in \mathbb{R}^{d\times k}$ is a pre-trained matrix, and $B \in \mathbb{R}^{d\times r}$ and $A \in \mathbb{R}^{r\times k}$ are trainable matrices. By setting the rank $r \ll min(d,k)$, the number of trainable parameters in LoRA fine-tuning can be reduced by a factor of 1000 compared to full parameter fine-tuning.

2.2 LLM Inference

LLM inference is an auto-regressive process that, given an input sequence $(x_1, \cdots, x_n)$, generates an output sequence $(x_{n+1}, \cdots, x_{n+T})$ according to:

$$p(x_1, \cdots, x_{n+T}) = p(EOS|x_1, \cdots, x_{n+T}) \prod_{t=1}^{n+T} p(x_t|x_1, \cdots, x_{t-1}),$$

where EOS is the end-of-sequence symbol. Mainstream LLM serving systems [36, 62] divide the process into two phases: (1) The prefilling phase generates the first new token x_{n+1} and stores the KV cache tensors representing the model states of the input tokens; (2) The decoding phase takes the latest generated token as input, computes and stores its KV cache tensors. Then it utilizes all KV cache tensors to generate a new token. The decoding phase will execute multiple decoding steps until the generation length limit T is reached or the EOS symbol is encountered.

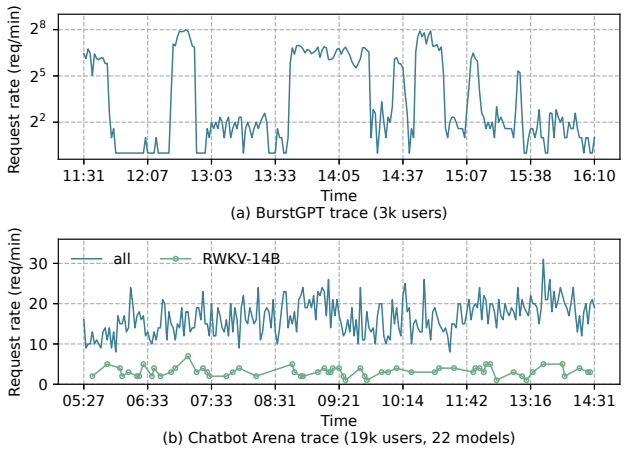


Figure 3: Real-world LLM inference workloads, BurstGPT [57] and Chatbot Arena [65].

2.3 Resource Multiplexing in LLM Workloads

The characteristics of LLM workloads reveal opportunities to improve GPU utilization by colocating LLM tuning and serving tasks as described in Section 2.3.1. However, in the presence of SLOs, present systems struggle to achieve high utilization while still guaranteeing the necessary low latencies as discussed in Section 2.3.2.

2.3.1 Opportunities

The advances of open-source LLMs make self-hosting LLMs in on-premise or cloud servers an attractive alternative to cloud APIs. These deployments usually over-provision GPU resources based on peak load, but the huge minute-to-minute fluctuations [21, 50] in request load lead to inevitable underutilization of resources during certain short and discrete time periods. Figure 3 shows the request loads of two real-world LLM inference workloads from thousands of users (i.e., BurstGPT [57] and Chatbot Arena [65]). The underutilization of GPU resource will be more serious if the request loads to multiple LLMs are unbalanced as shown in Figure 3(b). This observation reveals an opportunity to utilize idle GPU cycles of the highly fluctuating LLM serving workloads.

On the other hand, the Model FLOPs Utilization (MFU) of the LLM decoding process is typically less than 5% [64], because each decoding step loads and evicts the entire model between HBM and SRAM, yet the computation involves only the last generated token rather than the whole sequence. This observation reveals an opportunity to co-execute compute-intensive PEFT and memory-intensive LLM inference to maximize GPU computation utilization.

Tuning base LLMs for downstream tasks often only requires hundreds [51] or thousands [11] of labeled data samples, which can be completed in ten minutes to an hour using one or a few GPUs. The high frequency of computation-intensive LLM fine-tuning, along with its relatively low to moderate time and resource demands, makes it an ideal task for utilizing otherwise idle GPU capacity.

In addition to utilizing spare computing resources in the fluctuating LLM serving workloads, the growing popularity of AI PCs [12, 14] has also made resource multiplexing of PEFT and serving an emerging requirement. AI PCs enable users to locally run inferences on LLMs and regularly finetune them using their personal data [12, 26], highlighting the need for optimized resource multiplexing between these tasks.

2.3.2 Challenges

Previous studies have explored various resource multiplexing techniques to improve GPU utilization. However, existing solutions cannot maximize GPU utilization while meeting the SLO when co-executing PEFT and LLM inference, as illustrated in Figure 4.

Base model multiplexing. The increasing size of LLMs renders out-of-the-box solutions [1, 3, 8]—where each task or fine-tuned LLM must hold an exclusive copy of the base model—less practical, as even a single LLM can exhaust the entire GPU memory capacity, preventing the sharing of compute and memory bandwidth across different tasks or fine-tuned LLMs. Therefore, recent frameworks [18, 26, 31, 44, 48, 61] combine base model multiplexing (i.e., sharing the base model across different tasks and fine-tuned LLMs) with temporal multiplexing and/or spatial multiplexing to create opportunities for GPU resource multiplexing.

Temporal multiplexing. FineInfer [26] proposed *Deferred Continuous Batching* to schedule PEFT and LLM inference tasks in a temporal multiplexing manner [8]. The advantage of temporal multiplexing in co-execution is that it does not interfere with LLM decoding, thus ensuring that the TPOT of LLM inference meets the SLO. However, as the input sequence length of PEFT becomes longer or the model size becomes larger, TTFT will experience a significant delay because LLM inference cannot start until the current step of PEFT ends, as shown in Figure 4(a). Figure 5(left) shows that fine-tuning latency increases with input sequence length and model size, exceeding the commonly used TTFT SLOs.

Spatial multiplexing. An out-of-the-box solution is to use Nvidia MPS [1] with high-performance LLM serving and fine-tuning frameworks to schedule PEFT and LLM inference tasks in a spatial multiplexing manner. By carefully configuring `CUDA_MPS_ACTIVE_THREAD_PERCENTAGE` to partition the computing units (i.e., streaming multiprocessors), spatial multiplexing can maximize GPU utilization while ensuring SLOs when processing static LLM workloads. Besides inheriting the aforementioned drawbacks of out-of-the-box solutions, its main limitation is that the percentages are fixed at framework startup and cannot be adjusted dynamically during task execution. Figure 4(b) shows that, if the percentage of computing units allocated to PEFT is not restricted, the TPOT

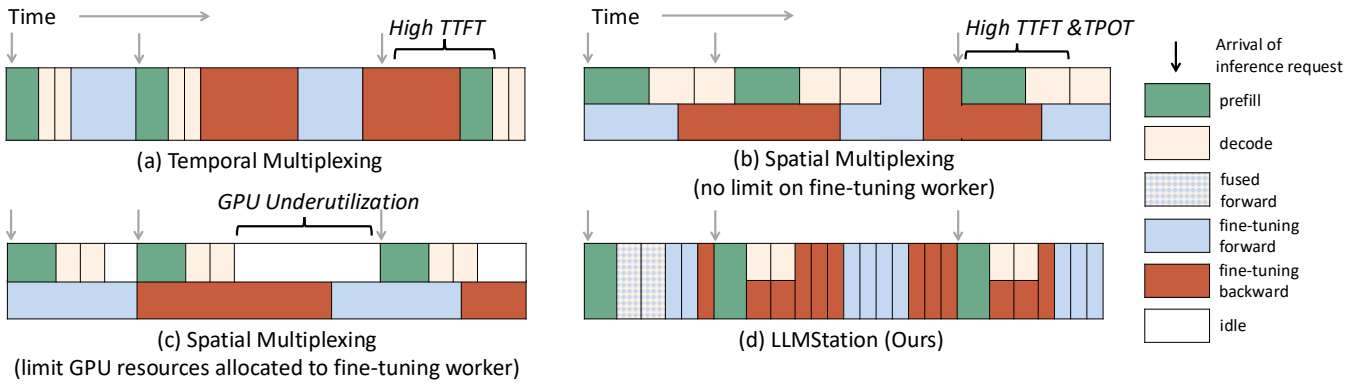


Figure 4: Illustration of different task scheduling strategies.

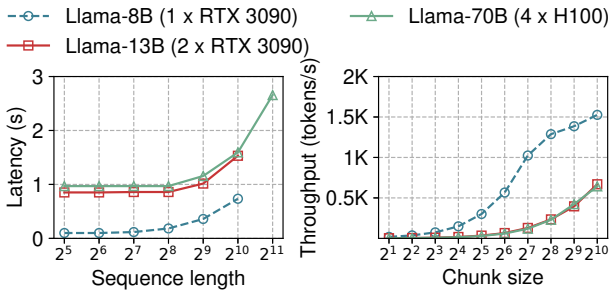


Figure 5: Latency for LoRA fine-tuning per sample for different sequence lengths and model sizes. (left) Throughput of chunked-training for different chunk sizes and model sizes. The complete input sequence length for each sample is fixed to 1024. (right)

of LLM inference becomes very high with many LLM inference requests. If one restricts the percentage of computing units allocated to PEFT, GPU resources will be underutilized when the LLM inference request load is low (Figure 4(c)).

FlexLLM [44] leverages chunked-training [40] to further break down the PEFT tasks. It divides the input sequence into several chunks and performs forward and backward passes on each chunk separately. By storing and reusing the results and states of each chunk required by rest chunks, chunked-training can achieve the same results as regular PEFT. Nevertheless, it is subject to two significant limitations. First, as shown in Figure 5(right), shorter input sequences cannot fully utilize the computing resources of the GPU [19, 34, 35, 38, 56] and it incurs additional data movement overhead. If the input sequence is divided into N chunks, the whole LLM needs to be loaded from slow HBM to fast SRAM $2N$ times. Second, it may degenerate into simple temporal or spatial multiplexing in real-world workloads and suffers from the same problems as the latter. For example, if chunked training does not split the input sequence to avoid incurring additional overhead when no LLM inference requests are running, it cannot ensure the SLO for newly arrived LLM inference requests while processing the current PEFT step.

3 LLMStation Overview

In this section, we give a brief overview of LLMStation, whose architecture is illustrated in Figure 6. LLMStation follows three design principles: (1) *SLO-guarantee*. The co-execution of PEFT and LLM inference should not violate the performance SLOs of LLM serving. (2) *GPU utilization maximiazion*. Our primary objective is to maximize the GPU utilization in each short time slice to improve PEFT throughput under the highly fluctuating real-world LLM serving workload. (3) *Memory efficiency*. The system should be able to reduce memory consumption by sharing common base model weights, adapter weights, and states between PEFT and inference. The principal components are as follows and details in Section 4 and Section 5:

1. SLO-aware task scheduler (Section 4). LLMStation is built around an iteration-level multi-tasking scheduling mechanism that improves PEFT throughput by assigning proper number of PEFT tasklets to be executed with the current inference iteration according to the SLOs. Before the execution of each iteration, the scheduler obtains the latencies of the different kinds of tasks using profiling results, cached runtime records or its own latency predictor, and then uses its planner to adaptively determine the number of PEFT tasklets to execute together.

2. Suspendable automatic differentiation engine (Section 5). Achieving our proposed scheduling mechanism requires system support to decompose the PEFT task into tasklets that can be executed together with the inference iterations without violating the SLO. We do so by taking an existing automatic differentiation engine (PyTorch Autograd [46]) and transforming its backward path to coroutines.

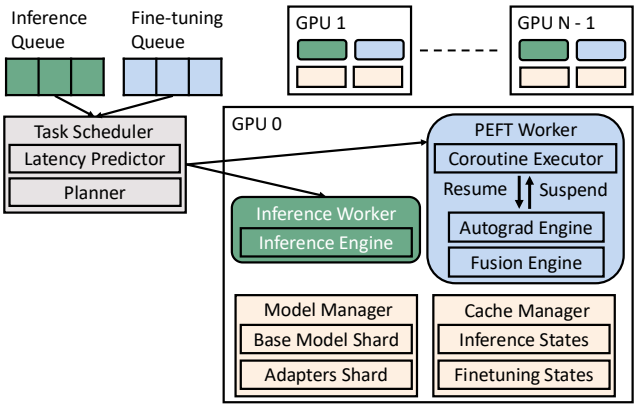


Figure 6: LLMStation overview.

3. Workers. LLMStation divides the workers into PEFT workers and inference workers to handle different parts of the co-execution in different situations: (1) If inference workers are in the prefilling phase, the PEFT workers will be suspended. (2) If the inference workers are in the decoding phase and PEFT workers are in the forward pass, the PEFT workers will fuse their forward pass and the decoding step of the inference workers. (3) If the inference workers are in the decoding phase and PEFT workers are in the backward pass, the PEFT workers will run PEFT tasklets in parallel.

4. Memory manager. Leveraging base model multiplexing, LLMStation's GPU-CPU memory manager only keeps one copy of the base model, adapter, and inference state, which is shared by PEFT workers and inference workers. Only the fine-tuning state is exclusive to PEFT workers. When the available GPU memory is insufficient to hold the new inference states, LLMStation can swap some GPU tensors to CPU memory and swap back when GPU memory is sufficient.

4 LLMStation Scheduler

In this section, we describe the detailed design of LLMStation's scheduler. At its core is a new iteration-level multitasking scheduling mechanism that improves PEFT throughput by co-executing PEFT tasklets with inference requests without violating SLOs, as illustrated in Figure 4(d).

4.1 Scheduling Strategy

The pseudocode in Algorithm 1 shows how LLMStation's scheduling strategy co-executes PEFT and LLM inference. Whenever new inference requests arrive (lines 8 - 11), the scheduler dispatches them to the inference workers to perform the prefilling step (line 18) while suspending the PEFT workers. Before performing the decoding step, the scheduler will send the batch size and length of input sequences, the current phase of PEFT workers, and the hardware and model configurations (e.g., GPU types and numbers, parallelism strategies, model sizes, and adapter types) to the latency predictor. If the PEFT workers are performing the forward pass, the latency predictor estimates the latency of fused execution of LLM decoding and PEFT forward tasklets (line 17). If the PEFT workers are performing the backward pass, the latency predictor estimates the latency of running LLM decoding step and PEFT backward tasklets in parallel (line 21). After obtaining the latencies, the scheduler's planner calculates the number of PEFT tasklets that can be executed together with the LLM decoding step without violating the SLOs (lines 18 and 22). Finally, the scheduler dispatches the tasks to corresponding workers (lines 19, 23 and 24).

When the fine-tuning request queue is empty, our scheduling strategy degenerates to continuous batching [62]. When the inference request queue is empty, our scheduling strategy

Algorithm 1 Iteration-level multitasking scheduling.

Input: hardware and model configurations c, prefilling SLO_p, and decoding SLO_d
1: let Q_i and Q_f denote inference request queue and fine-tuning request queue
2: let W_i and W_f denote inference workers and fine-tuning workers,
3: let $B_i, B_f \leftarrow \emptyset, \emptyset$ denote the current batch of inference requests and the current batch of fine-tuning samples
4: let $B_{new} \leftarrow \emptyset$ denote the batch of new inference requests
5: **while** True **do**
6: $B_{new} \leftarrow \emptyset$
7: **if** $B_f = \emptyset$ **then** $B_f \leftarrow$ get_first_batch(Q_f)
8: **for all** $r \in Q_i$ **do**
9: **if** r cannot fit in the memory **then**
10: Break
11: $B_{new} \leftarrow B_{new} \cup r$
12: **if** $B_{new} \neq \emptyset$ **then**
13: $Q_i \leftarrow Q_i \setminus B_{new}$
14: prefill(W_i, B_{new}, SLO_p)
15: $B_i \leftarrow B_i \cup B_{new}$
16: **if** W_f is in forward pass **then**
17: latencies $\leftarrow$ latency_predictor(forward, c, B_i, B_f)
18: budget $\leftarrow$ planner(latencies, SLO_d)
19: forward_fused(budget, W_f, B_i, B_f)
20: **else**
21: latencies $\leftarrow$ latency_predictor(backward, c, B_i, B_f)
22: budget $\leftarrow$ planner(latencies, SLO_d)
23: async_decode(W_i, B_i)
24: async_backward(budget, W_f, B_f)
25: $B_i \leftarrow B_i \setminus$ finished_requests(B_i)
26: $B_f \leftarrow B_f \setminus$ finished_samples(B_f)

degenerates to normal fine-tuning. In actual implementation, LLMStation also seeks to optimize GPU utilization during the prefilling phase (line 14) by: (1) selectively co-executing PEFT and prefilling steps, as some prompts are short; and (2) scheduling the prefilling step using *Deferred Continuous Batching* from FineInfer [26]. When there is no running inference request, *Deferred Continuous Batching* defers newly arrived requests according to their arrival time and prefilling SLO, thus improving their chance of being able to batch up with other requests.

4.2 Planner and Latency Predictor

Considering that the workloads of LLM inference vary significantly across different short time periods, the scheduler needs to make decisions based on the information of the next iteration adaptively to maximize the PEFT throughput.

Objective. Let SLO_d denote the decoding SLO, l_d and l_{d0} denote the latencies of running a decoding tasklet with and without PEFT tasklets, and l_p denote the latency of running a PEFT tasklet with decoding tasklets. The objective and the SLO constraint can be formulated as

$$\begin{aligned} \max \quad & N_p \\ \text{s.t.} \quad & N_d \cdot l_d + (N_{d,all} - N_d) \cdot l_{d0} \leq SLO_d \\ & N_p \cdot l_p \leq N_d \cdot l_d \end{aligned}$$

where N_p and N_d are the number of PEFT tasklets and decoding tasklets to be executed together, and $N_{d,all}$ is the total number of decoding tasklets that need to be completed in the upcoming iteration. We can further derive the following inequality about N_p:

$$N_p \leq \frac{(SLO_d - N_{d,all} \cdot l_{d0}) \cdot l_d}{(l_d - l_{d0}) \cdot l_p}. \tag{1}$$

Besides the SLO constraint, the planner also calculates the peak memory usage of PEFT and LLM inference, respectively, to check whether the planned tasks can meet the memory constraints before execution.

Latency predictor. In Eq. 1, although SLO_d and $N_{d,all}$ are user-configured constants, the remaining variables (i.e., l_{d0}, l_d and l_p) are highly dependent on the hardware and software environment, the length of the input sequence, the distribution, number, and architecture of adapters and models, etc. Furthermore, l_d and l_p differ in the forward and backward passes and are highly dependent on the tasks executed in parallel with them. Therefore, for co-executed tasklets without cached runtime records or profiling results, the planner adopts a learned latency predictor to estimate the three different types of latency mentioned above. The latency predictor used to estimate the latencies can be defined as:

$$\mathbf{l} = \mathbf{y}(\frac{computation}{FLOP/s_{GPU}}, \frac{communication}{bandwidth_{interconnect}}, \frac{memory_access}{bandwidth_{memory}}, \mathbf{w}),$$

where $\mathbf{l}$ are predicted latencies, $\mathbf{y}$ are prediction models, and $\mathbf{w}$ of each prediction model are parameters that need to be learned. Similar to recent works in LLM performance prediction [32, 40], the current implementation of LLMStation adopts linear models for $\mathbf{y}$ and uses simple feature engineering to obtain computation, communication, and memory access from raw input features (e.g., the batch size and sequence length of inference requests and PEFT samples, and the architecture of models and adapters).

Note that we only use the learned latency predictor when the system does not have any profiling results or cached runtime records. Therefore, after a few initial iterations, the prediction error and runtime latency of the simple linear regression models have little impact on performance, as the actual co-execution latencies are stored in the system's cache, which is organized as a nested index. At the top level, records are grouped and keyed by the (hardware, model, adapter) tuple. Within each group, the records of decoding tasklet latency and PEFT tasklet latency are further keyed by (decode batch size, PEFT input length). Since the required information is available at the start of each iteration, caching and usage of latencies are straightforward. When co-locating PEFT-based LLM inference tasks and PEFT tasks, LLMStation additionally includes the adapters used by PEFT-based LLM inference as a key in the index.

5 LLMStation Engines

Now we describe the design of LLMStation's execution engines. For inference requests, the prefilling step is handled by the inference engine and the decoding step is selectively handled by the inference engine or the fusion engine. For PEFT samples, the backward pass is handled by the Autograd engine and the forward pass is handled by the fusion engine.

5.1 Autograd Engine

In the PyTorch Autograd engine, the backward pass is done synchronously by the PEFT workers and the gradients are calculated from the last layer to the first layer according to the computation graph constructed in the forward pass. As shown in Figure 7(a), once the backward pass starts, the PEFT workers cannot adjust their execution according to the current workload of the inference workers. As a result, it can not realize the iteration-level multitasking scheduling algorithm (See Algorithm 1) to ensure that the SLOs of inference requests are met.

The key to splitting the backward pass into tasklets is transforming nested functions on the backward path into coroutines [2], which are functions that can be suspended voluntarily and be resumed later. In the LLMStation Autograd engine, the functions involved in the function call chains of the backward pass are implemented as coroutines, instead of "normal" functions. We use `co_await` to invoke other coroutines and

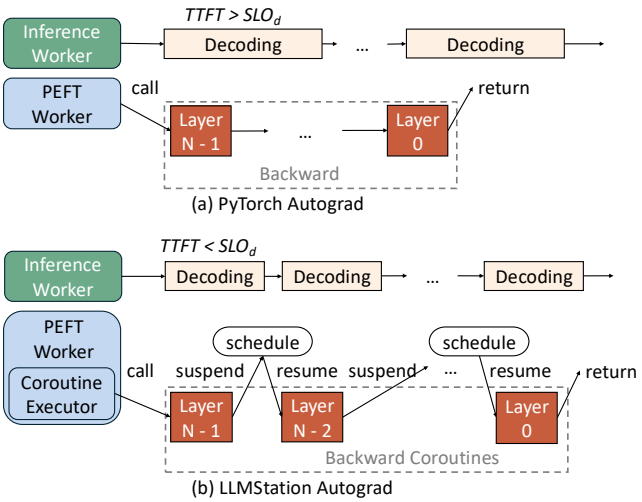


Figure 7: An illustration of Pytorch Autograd engine (a) and LLMStation Autograd engine (b) running backward pass while the inference engine is running the decoding phase.

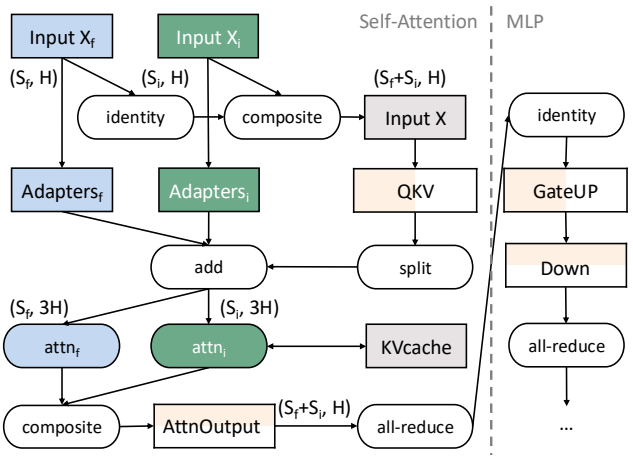


Figure 8: An illustration of LLMStation Fusion engine co-executing inference requests and fine-tuning samples on a Transformer layer. We only depict the QKV projection, the Attention, the Attention Output projection, and the multilayer perceptron (MLP) for simplicity.

replace return with `co_return`. After this transformation, the PEFT worker can use its coroutine executor to voluntarily suspend and resume the backward pass. Figure 7(b) illustrates the end-to-end control flow. When a new backward task is created, LLMStation starts to execute it in a coroutine. According to the workload of the next LLM decoding step, LLMStation scheduler uses its planner to estimate how many layers of the backward pass could be executed together without violating SLOs. After the PEFT tasklets are completed, the LLMStation Autograd Engine returns control to the PEFT worker. The scheduler will then estimate the number of PEFT tasklets to run in the next iteration based on the next LLM decoding step and resume the backward coroutine. Finally, after the backward pass of the current batch is completed, the scheduler starts a new batch and performs a forward pass on it, which we will discuss in the next subsection. The above design enables LLMStation Autograd engine to help users flexibly generate suspendable backward passes for different models and adapters.

5.2 Inference and Fusion Engines

When the PEFT workers are in the backward pass, the current iteration of that inference request will be fully processed by the inference worker. The inference engine in the inference workers is similar to the inference engine in the mainstream LLM serving systems. When an inference request is in the decoding phase and PEFT workers are in the forward pass, LLMStation uses the Fusion engine to amortize the data movement, communication, and kernel launch overheads in the PEFT forward pass. Previous works leverage prompt and generation composition strategies [16, 27] to amortize the overheads of the prefilling phase, but we found that less than 5% of decoding iterations are mixed with prefilling when running real-world workloads, which leaves room for us to composite the decoding phase and the PEFT forward pass.

Figure 8 describes the execution flow of the Fusion engine, where computational operations are represented by ellipses and weight matrices, input sequences, and intermediate states are represented by rectangles. Instead of processing inference requests of shape (S_i, H) and fine-tuning samples of shape (S_f, H) separately, the Fusion engine processes them as the following procedure: (1) composites them to an input tensor of shape $(S_f + S_i, H)$ which is then fed to QKV projection; (2) adds up the results from the base model and the adapters required by inference and PEFT; (3) splits the results and performs self-attention operation separately; (4) composites them again and then performs AttnOutput projection, GateUP projection and Down projection. With the Fusion engine, we can utilize the computing resources in decoding phases without violating the decoding SLO. For distributed co-execution, LLMStation adopts tensor model parallelism [49] and partitions the weight matrices of QKV projection and GateUP projection along columns (output dimension), and the weight matrices of AttnOoutput projection and Down projection along rows (input dimension). All-reduce operations are performed after AttnOutput projection and Down projection during the forward pass and before QKV projection and GateUP projection during the backward pass.

Similar to the co-execution of the decoding phase and the backward pass, the LLMStation scheduler estimates the number of layers that the decoding phase and the forward pass can run together. The remaining layers of the decoding phase are executed in the normal decoding way. Alternatively, the LLMStation Fusion engine can composite the inference requests

and the PEFT samples for every layer and let the LLMStation scheduler estimate the number of tokens that can be batched.

6 Evaluation

We empirically evaluate the performance of LLMStation on both synthetic and real-world workloads. Through experiments, we answer the following questions:

- How does LLMStation compare to other specialized frameworks and out-of-the-box solutions for co-execution of PEFT and LLM inference?
- What are the throughput-latency tradeoffs in LLMStation?
- What are the overheads of LLMStation 's scheduler and Autograd engine?

6.1 Experimental Setup

Here we describe the setup used throughout the experiments.

Hardware. The experiments are conducted on various hardware settings, including a server with four RTX 3090 GPUs and servers with four H100 GPUs. The RAM capacities of these servers range from 256 GB to 512 GB, and all are equipped with intra-node NVLink.

Models and adapters. We use representative open-source LLMs, the Llama series [22], and one of the most popular PEFT methods, LoRA [29]. As listed in Table 1, we considered three different sizes of Llama models, 7B, 13B, and 70B, and three different sizes of LoRA adapters, with ranks of 8, 16, and 32, respectively.

Datasets and traces. For input sequences, we use the ShareGPT [6] datasets for LLM inference and the Alpaca [53] dataset for PEFT, both are real-world datasets. For inference request rates, synthetic workloads send inference requests at a steady rate and real-world workloads send inference requests according to the BurstGPT [57] trace.

Implementation. LLMStation is implemented in about 3k lines of code and combines multiple components from different systems. We modified PyTorch Autograd [46] using C++ stackless coroutines and used it as the Autograd engine for LLMStation. The inference engine, memory manager and cache manager are built atop vLLM [36]. The fusion engine is built atop FineInfer [26], and the implementation of different parallelism strategies refers to Nanotron [4]. Also, we align the fine-tuning results of LLMStation with Hugging Face PEFT [5] to ensure the correctness of our implementation.

Baselines. We compare LLMStation with three baselines. We use the combination of vLLM [36] + torchtune [54] as the out-of-the-box solution. vLLM is a SoTA LLM serving system with the batching mechanism of adapter computation from Punica [18] and an optimized GPU memory allocation mechanism. torchtune is a high-performance fine-tuning system that integrates a range of commonly used performance optimization techniques. In the category of specialized frameworks, we evaluate FineInfer [26] and chunked-training [44], which are discussed in Section 2.

Table 1: Model and GPU configurations.

Model	GPU	# Layers	TP Degree
Llama-3.1-8B	2 RTX 3090	32	2
Llama-2-13B	4 RTX 3090	32	4
Llama-3.1-70B	2×4 H100	80	4

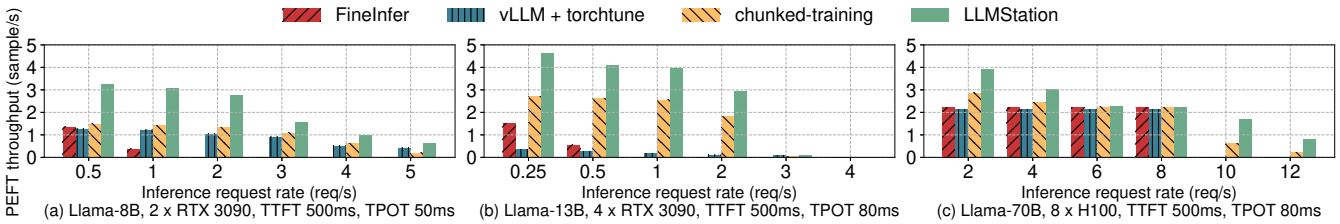
Metrics and SLOs. In end-to-end comparison, we report the throughput (i.e., samples/s) of PEFT that each system can achieve while the given P99 TTFT SLO and P99 TPOT SLO are not violated. Unless otherwise specified, the P99 TTFT SLO is set to 500 ms for all configurations, and the P99 TPOT SLO is set to 50 ms for Llama-3.1-8B and 80 ms for larger models. To compare latencies, we report the latency for different systems to achieve a given PEFT throughput.

6.2 End-to-End Comparison

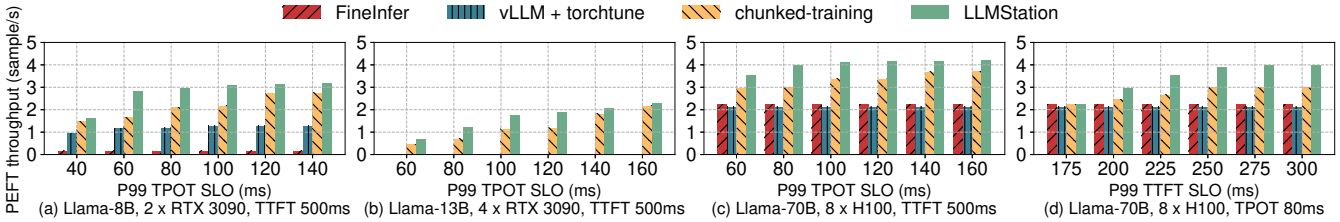
In this section, we evaluate the end-to-end performance using both synthetic and real-world workloads. To get the best performance of baselines, we carefully tune their configurations such as deferral bound for FineInfer, `CUDA_MPS_ACTIVE_THREAD_PERCENTAGE` for vLLM + torchtune, and chunk size for chunked-training.

Synthetic workloads with varying request rates. Our first experiment evaluates the PEFT throughput that different systems can achieve at different inference request rates under commonly used SLO targets. Figure 9i shows the results on the synthetic workloads with different inference request rates. When the inference request rates are relatively low (i.e., $\leq$ 0.5 req/s) in serving Llama-8B and Llama-13B with RTX 3090, LLMStation outperforms FineInfer/vLLM + torchtune/chunked-training by 2.38∼8.17×/2.53∼14.77×/1.57∼2.18×. As a temporal multiplexing solution, FineInfer does not co-execute the decoding phase and PEFT, and therefore cannot fully utilize the computational power of the GPU. As a spatial multiplexing solution, vLLM + torchtune needs to limit the `CUDA_MPS_ACTIVE_THREAD_PERCENTAGE` of the PEFT workers at launch time to ensure SLOs, but this also reduces their PEFT throughput when no inference requests are being processed. Chunked-training does not suffer from the above two problems, and its slowdown is mainly due to the inherent additional data movement overhead. As the inference request rate increases, FineInfer's PEFT throughput quickly drops to zero since it cannot guarantee TTFT. The PEFT throughput of the other systems gradually becomes very close, until it drops to zero when TPOT cannot be guaranteed.

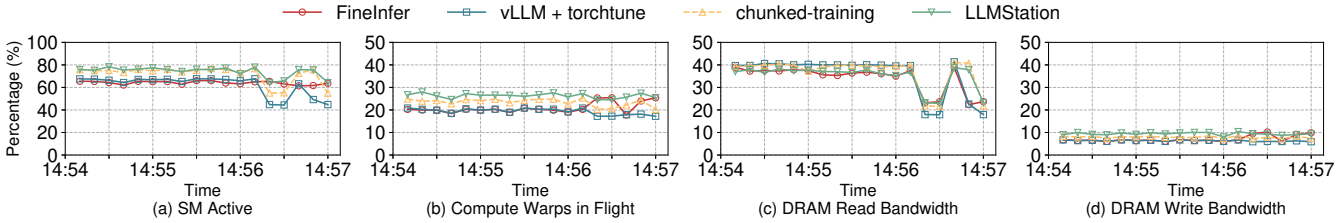
When using two servers equipped with four H100 GPUs



(i) PEFT throughput under synthetic workloads and varying inference request rates.



(ii) PEFT throughput under real-world workload and varying SLO targets.



(iii) Case study of GPU utilization under real-world workload.

Figure 9: End-to-end comparison of LLMStation and baselines.

and built-in node NVLink to serve Llama-70B, the performance difference also comes from the flexibility of deployment and service. Considering the low interconnection bandwidth between servers and the high PEFT latency per sample on Llama-70B, we need to deploy a 4-way tensor-parallel base model for FineInfer on each server instead of partitioning the model to two servers via pipeline parallelism or 8-way tensor parallelism. Considering that the vLLM + torchtune cannot share the base model, and the GPU memory of a server is only enough to place the basic model and state of vLLM or torchtune, we need to deploy vLLM and torchtune on different servers respectively. For a fair comparison, LLMStation and chunked-training are deployed in the same way. Therefore, when inference requests cannot saturate a single server, LLMStation outperforms FineInfer/vLLM + torchtune/chunked-training by up to 1.77×/1.8×/1.38×. This is because FineInfer and vLLM + torchtune can only use the resources of one server for PEFT, while LLMStation can use the total remaining resources of two servers. When executing inference requests requires two servers to guarantee the SLOs, the PEFT throughput of FineInfer and vLLM + torchtune drops to zero. On the contrary, LLMStation can still leverage the remaining resources of one server to perform PEFT until inference requests saturate two servers.

Real-world workloads with varying SLO targets. Our second experiment tests how different approaches perform under varying SLO targets under real-world workloads. The trace was extracted from BurstGPT [57], with an average request rate of 4.15 req/s. Figure 9ii(a - c) show results under varying P99 TPOT SLOs, where the performance differences can be attributed to similar reasons discussed previously. In real-world workloads, more requests arrive at similar times, so the decoding phases of more requests can be batched together, which leaves more room for PEFT. However, vLLM + torchtune and FineInfer may perform worse on real-world workloads with fluctuating request loads than on synthetic workloads with stable request loads. Since more requests arrive at similar times, vLLM + torchtune needs to impose stricter resource constraints on PEFT workers to avoid violating SLOs, while FineInfer may always violate TTFT SLOs in some cases. As the P99 TPOT SLO increases, the PEFT throughput of LLMStation and chunked-training gradually becomes very close, since chunked-training can use a larger chunk size. As shown in Figure 9ii(d), the performance of LLMStation is also limited under strict P99 TTFT SLOs since the prefilling step can only be executed after the current decoding step is finished. Therefore, the number of PEFT tasklets that can be executed together is now limited by the time remaining in the

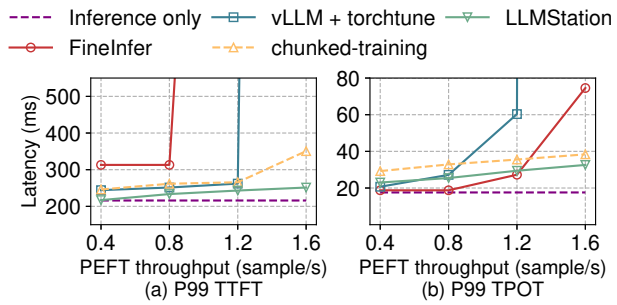


Figure 10: P99 TTFT and P99 TPOT of running Llama-3.1-8B on two RTX 3090 GPUs under real-world workloads and varying PEFT throughput settings.

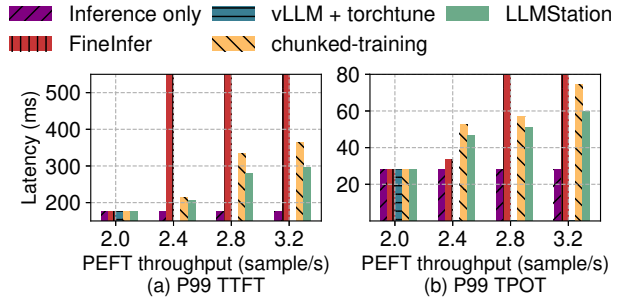


Figure 11: P99 TTFT and P99 TPOT of running Llama-3.1-70B on eight H100 GPUs under real-world workloads and varying PEFT throughput settings.

prefilling phase (rather than the decoding phase).

Case study of GPU utilization. We replayed a 3-minute segment of trace from Figure 9ii(a) using Llama-70B on eight H100 GPUs, with P99 TTFT and TPOT SLOs set to 500 ms and 80 ms, respectively. As shown in Figure 9iii, LLMStation achieves high `SM Active`, `Compute Warps in Flight`, and `DRAM Write Bandwidth` during most of the time. FineInfer only achieves high utilization when inference request loads are low since it cannot execute PEFT tasks when inference tasks are running and vLLM + torchtune can not utilize half of the GPU resources even when inference request loads are low since it cannot share the base model. Chunked-training achieves high `SM Active` and `DRAM Read Bandwidth` but low `Compute Warps in Flight`, reflecting that it suffers from additional data movement overhead.

6.3 Throughput-Latency Tradeoff

In this section, we do not consider SLOs and use real-world traces to explore how inference latency is affected under varying PEFT throughput settings. For the experiment using two RTX 3090s to serve Llama-8B, we extracted another trace from BurstGPT [57] with an average inference request rate of 0.88 req/s. As shown in Figure 10, LLMStation achieves up to 33.13×/52.64×/1.4× lower P99 TTFT and 2.29×/362.49×/1.29× lower P99 TPOT than FineInfer/vLLM + torchtune/chunked-training. FineInfer's P99 TTFT is always higher than other variants because its scheduling algorithm postpones the execution of prefilling phases to obtain higher PEFT throughput. When the PEFT throughput is 1.6, the P99 TTFT of inference requests even exceeds 8 seconds, which is unbearable for users. But its P99 TPOT is up to 36% lower than LLMStation when PEFT throughput is low. Both P99 TTFT and P99 TPOT of vLLM + torchtune are higher than LLMStation because it underutilizes GPU resources when no inference request is being processed and needs to pay more at the expense of latency for inference requests. The trend of latency variation for chunked training is similar to LLMStation, but slightly higher due to its additional data movement overhead. For the experiment using eight H100 GPUs to serve Llama-70B, we use the same real workload as in Section 6.2. As shown in Figure 11, LLMStation achieves up to 180.48×/1.23× lower P99 TTFT and 14.22×/1.24× lower P99 TPOT than FineInfer/chunked-training. Similarly, FineInfer's P99 TPOT is up to 28% lower than LLMStation when PEFT throughput is low but its P99 TTFT exceeds 25 seconds when PEFT throughput is high. In this experiment, if the PEFT throughput needed exceeds what a server with four H100 GPUs can handle, vLLM + torchtune is unable to perform LLM inference since we need to use all eight H100 GPUs to deploy torchtune.

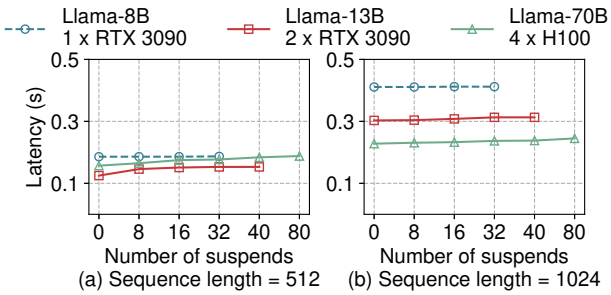


Figure 12: The time of backward pass with varying numbers of suspends.

6.4 LLMStation Autograd Engine and Scheduler Overheads

Now we study the overheads of the scheduler and Autograd.

Autograd engine. Llama-3.1-8B, Llama-2-13B and Llama-70B have 32, 40 and 80 layers respectively, and we selectively suspend and resume after each layer to evaluate the overhead of context switch in this experiment. Figure 12 shows the latency of the backward pass under different sequence lengths, model sizes, and GPUs and varying numbers of suspends. For single-GPU fine-tuning, our Autograd engine incurs less than 0.5% additional latency thanks to the low overhead of

C++ stackless coroutines [2]. For multiple-GPU fine-tuning, our Autograd engine incurs up to 18% additional latency. We attribute this to the extra inter-GPU and inter-process synchronization and a slight amplification in synchronization overhead due to execution variability.

Scheduler. On an AMD EPYC 7313 16-Core processor, the average latency of the planner and the latency predictor of LLMStation scheduler is 18 microseconds per iteration, and the average latency is less than 1 microsecond when profiling results or cached runtime records are available since we need not to call the latency predictor. For the latency predictor, we train the prediction models with our profiling results from RTX 3090 GPU servers and H100 GPU servers and test them on A100 GPU servers. For Llama-3.1-8B and Llama-2-13B with the rank of LoRA adapters set to 8, the overall average Coefficients of Determination [7] (i.e., R^2 score), are 0.73 and 0.69. The R^2 score is the default score function for many regression models in scikit-learn [47]. By definition, the score ranges from $-\infty$ and 1.0, with scores closer to 1 indicating that the model explains a large portion of the variance.

6.5 Impact of Access Distribution and Sizes of Adapters

The final set of experiments analyzes the impact of adapter access distribution and adapter size on PEFT. We run Llama-3.1-8B and the LoRA adapters on two RTX 3090 GPUs and use the same real-world workload as in Section 6.3. For the adapter access distribution experiment, we set the rank of LoRA adapters to 32 and the number of LoRA adapters to 8 and use three different access distributions: (1) None: None of the LoRA adapter is accessed. (2) Skewed: 50% of LoRA adapters are uniformly accessed. (3) Uniform: All LoRA adapters are uniformly accessed. As shown in Figure 13(left), LLMStation outperforms FineInfer/vLLM + torchtune/chunked-training by up to 2.98×/2.41×/1.74×. The performance of all variants decreases slightly when the percentage of accessed LoRA adapters increases from 0% to 50%. The performance of all variants drops to zero in the uniform distribution, since serving inference requests alone violates the P99 TTFT SLO.

For the adapter size experiment, we vary the rank of LoRA adapters and set the number of LoRA adapters to 8 and the percentage of accessed LoRA adapters to 50%. As shown in Figure 13(right), LLMStation outperforms FineInfer/vLLM + torchtune/chunked-training by up to 2.98×/2.41×/1.73×. The performance of all variants decreases slightly when the rank of LoRA adapters increases from 8 to 32. The results of these two experiments match the observations from Punica [18]. They found that as the rank of the LoRA adapter increases, the performance is more sensitive to the access distribution. When the rank equals to 32, the latency in their distinct distribution (i.e., our uniform distribution) is up to 2.5× higher than in skew distribution.

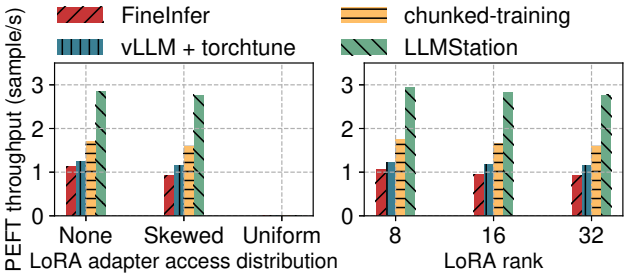


Figure 13: PEFT throughput with varying access distribution to adapters (left) and varying sizes of adapters (right).

7 Discussion and Limitations

Operator-level suspend. LLMStation's Autograd Engine is natively able to suspend and resume between operators (i.e., nodes of the backward computation graph). Considering that the backward computation graphs of LoRA fine-tuning on Llama-3.1-8B and Llama-3.1-70B have about 4k and 10k nodes respectively, the average latency per tasklet can be as low as a few microseconds. Therefore, if the latency of each layer in backpropagation is too high, LLMStation can switch to a combination of iteration-level scheduling and operator-level suspends. For the PEFT of ultra-long context LLMs (i.e., millions of tokens), a potential solution is to integrate LLMStation and chunked-training [40, 44], which we leave as future work.

Decoding SLO, TPOT and disaggregated LLM serving. TTFT and TPOT are common SLOs used in today's LLM serving. In our scheduling mechanism, the definition of prefilling SLO is the same as TTFT, but the definition of decoding SLO is slightly different from TPOT. Decoding SLO is defined as the latency of each decoding iteration while TPOT is defined as $TPOT = (e2e_time - TTFT)/number_of_generated_tokens$, where $e2e_time$ represents the time between the arrival and completion of a request. They are identical in disaggregated LLM serving systems, and only these systems can strictly meet the TTFT and TPOT SLOs in theory, since there is no interference between prefilling and decoding. For common monolithic deployments, we need to set the decoding SLO slightly smaller than the TPOT SLO. To transform the scheduling mechanism into a TPOT-aware algorithm, one needs to continuously update the respective TPOTs of all running inference requests and make decisions based on them.

Parallelization. LLM serving typically parallelizes models across different models via tensor model parallelism (TP) [49] and pipeline model parallelism (PP) [40, 45]. Since tensor model parallelism performs better in servers with NVlink and most open source models can be accommodated by 8 GPUs on a single server, we consider only TP in the paper and plan to incorporate PP into LLMStation in the future. Dy-

namoLLM [50] observes that the most cost-beneficial tensor parallelism strategy is different for workloads with different request rates, request input lengths and output lengths. Therefore, they adjust tensor parallelism every five minutes if it is necessary since per adjustment might take up to several seconds. The current implementation of LLMStation does not consider workload-aware dynamic parallelization. We assume that the parallelization is controlled by the inference worker and the PEFT worker performs fine-tuning accordingly based on the given settings.

Applicable models. Our approach can be seamlessly applied to large auto-regressive vision, time-series, tabular, DNA, and multimodal models that share similar memory-bound characteristics of the decoding phase, as well as other workloads [28] that have both compound-bound and memory-bound phases. For models or workloads that are purely memory-bound or compute-bound, the benefit will only come from the utilization of idle GPU cycles of the fluctuating serving workloads.

8 Related Work

LLMStation builds upon many techniques from LLM serving and tuning, GPU resource multiplexing and scheduling.

System optimizations for LLM serving. A flurry of optimizations has been proposed recently to enhance LLM serving from various aspects. Orca [62] proposed continuous batching that batching requests at the iteration level and interleaving prefill and decoding phases to amortize data movement overhead. vLLM [36] reduces memory fragmentation in KV cache management with PagedAttention to maximize throughput. Punica [18] and SLoRA [48] explored efficient batch processing of requests for multiple PEFT models with Segmented Gather Matrix-Vector Multiplication and Multi-size Batched Gather Matrix-Vector Multiplication, respectively. DeepSpeed-FastGen [27] and Sarathi-Serve [16] proposed chunked prefill to batches prefilling and decoding requests to improve GPU utilization. Our work adopts all the above optimizations to improve serving performance, but further focuses on improving GPU utilization without violating SLOs by multiplexing GPU resources between LLM tuning and serving and fine-grained scheduling.

GPU resource multiplexing. Beyond the works covered in Section 2, a substantial body of research has also explored GPU computation multiplexing [8, 24, 52] to improve GPU utilization, which can also be divided into temporal multiplexing and spatial multiplexing. In addition, some studies leverage software-hardware co-design to enable GPU memory multiplexing [3, 55, 60].

Temporal multiplexing shares exclusive GPU cycles by context-switching between multiple jobs. Time-Sliced NVIDIA vGPU [8] schedules vGPUs to run in series. However, out-of-the-box solutions using temporal multiplexing solutions with existing tuning frameworks and serving frameworks suffer from significant context switching overhead.

Spatial sharing allows multiple processes to run on the same GPU simultaneously, eliminating task switch overheads and enhancing GPU utilization. NVIDIA MIG [3] provides applications with exclusive access to a dedicated memory space and streaming multiprocessor, which prevents fine-tuning systems and serving systems from sharing the same base model. However, this will sacrifice statistical performance because they can only use smaller models due to memory constraints. Reef [24] and Orion [52] enable fine-grained spatial multiplexing by padding kernels that have no performance interference on each other together based on their computation and memory access characteristics and then co-execute them. Given that they can only see kernels that have been submitted to the GPU work queue and do not consider interference from interconnect (i.e., NVlink and PCIe), this is a safe execution strategy that does not violate the SLOs. However, they cannot maximize PEFT throughput due to excessive constraints for co-execution. LLMStation is a user-space solution for spatial-temporal GPU resource multiplexing. The iteration-level multitasking scheduling mechanism allows LLMStation to achieve fine-grained spatial multiplexing and optimal co-execution plan since it has the information of all kernels to be executed in the next iteration as well as the SLO constraints.

Co-locating tuning and serving. Recent studies [17, 31] have pointed out the emergent requirements and benefits of co-locating LLM tuning and inference tasks. Several pioneers [26, 44] proposed approaches that multiplex GPU resources for this specific application scenario and are the most relevant works to LLMStation. Since they have been covered in Section 2, we do not repeat them here.

GPU cluster scheduling. Numerous cluster-level GPU resource allocation and management frameworks [20, 30, 58, 59] have been proposed in recent years to improve resource utilization for GPU clusters. AntMan [59], Gandiva [58], and Lucid [30] observe the fluctuating resource demands of deep learning (DL) training jobs and explore opportunities for sharing GPUs across multiple training jobs. GSLICE [20] co-locates DL inference jobs by apportioning the GPU% among them with MPS according to their throughput and latency at varying GPU%. The resource allocation granularity of these frameworks is coarser than LLMStation, and they do not consider the base model multiplexing and specific optimizations between fine-tuning and serving tasks for three reasons. Firstly, unlike LLM workloads, the input size of the inference workload of general DL models is relatively consistent and only one iteration is performed for each request. Therefore, for this relatively stable workload, coarse-grained resource allocation is sufficient. Secondly, the training of general DL models is less expensive or time-consuming so there is no need to reuse base models for fine-tuning or serving. Thirdly, the inference of general DL models is less memory-bound so the benefits from co-executing tuning and serving of them would be limited. Given the growing requirements

in tuning and serving LLMs, applying the optimizations in LLMStation to cluster-level scheduling would be interesting and viable future work.

Systems using coroutines. Much recent work has devised cooperative scheduling to improve the utilization of computation resources. CoroBase [25] models transactions as coroutines, thus enabling overlapping data fetching and computation within transactions. LuisaRender [66] proposes a GPU coroutine model for flexible splitting and scheduling of sophisticated rendering tasks. NVIDIA [28] leverages Boost [9] coroutines to overlap the processing of multiple molecular dynamics simulations, improving GPU utilization without complex code restructuring.

9 Conclusion

We highlighted the gap between resource multiplexing and its adoption in existing systems for LLM workloads. Prior approaches often underutilize GPU resources due to dedicated service and solutions for colocating LLM serving and other tasks were not ready. To fill this gap, we proposed LLMStation, a flexible spatial-temporal multiplexing and scheduling system concurrent LLM fine-tuning and inference. LLMStation achieves high performance via a new iteration-level multitasking scheduling algorithm, an Autograd engine that enables lightweight context switch via stackless coroutine, and an inference engine that merges memory-bound operation in inference and computation-bound operation in PEFT. Evaluation results show that LLMStation can achieve 1.38–14.77× higher PEFT throughput than highly-optimized baselines while meeting inference latency SLOs.

Acknowledgments

We would like to thank Timothy Roscoe and Shivaram Venkataraman for their valuable discussions. We also thank the anonymous reviewers from the Program Committee and the Artifact Evaluation Committee of USENIX ATC '25, as well as our shepherd, Ashish Panwar, for their insightful reviews and feedback.

References

[1] Nvidia multi-process service. https://docs.nvidia.com/deploy/mps/index.html, 2013.

[2] Technical specification — c++ extensions for coroutines. https://www.iso.org/standard/73008.html, 2017.

[3] Nvidia multi-instance gpu. https://www.nvidia.com/en-us/technologies/multi-instance-gpu/, 2020.

[4] Hugging face nanotron. https://github.com/huggingface/nanotron, 2023.

[5] Hugging face peft. https://github.com/huggingface/peft, 2023.

[6] Sharegpt. https://sharegpt.com/, 2023.

[7] Coefficient of determination. https://en.wikipedia.org/wiki/Coefficient_of_determination, 2024.

[8] Time-sliced nvidia vgpu internal architecture. https://docs.nvidia.com/ai-enterprise/3.3/user-guide/index.html#architecture-internal-grid-vgpu-time-sliced, 2024.

[9] Boost c++ libraries. https://www.boost.org/doc/libs/latest/libs/coroutine/doc/html/index.html, 2025.

[10] Customize a model with fine-tuning. https://learn.microsoft.com/en-us/azure/ai-services/openai/how-to/fine-tuning?tabs=azure-openai&pivots=programming-language-studio, 2025.

[11] Dataset guide. https://docs.unsloth.ai/basics/datasets-guide, 2025.

[12] Nvidia project digits. https://nvidianews.nvidia.com/news/nvidia-puts-grace-blackwell-on-every-desk-and-at-every-ai-developers-fingertips, 2025.

[13] Understanding fine-tuning. https://www.databricks.com/glossary/fine-tuning, 2025.

[14] What is an ai pc? https://www.intel.com/content/www/us/en/products/docs/processors/core-ultra/ai-pc.html, 2025.

[15] Divyanshu Aggarwal, Sankarshan Damle, Navin Goyal, Satya Lokam, and Sunayana Sitaram. Exploring continual fine-tuning for enhancing language ability in large language model. In *NeurIPS 2024 Workshop on Scalable Continual Learning for Lifelong Foundation Models*.

[16] Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav S. Gulavani, Alexey Tumanov, and Ramachandran Ramjee. Taming throughput-latency tradeoff in LLM inference with sarathi-serve. In *18th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2024, Santa Clara, CA, USA, July 10-12, 2024*, pages 117–134. USENIX Association, 2024.

[17] Jiaxuan Chen. Comparative analysis and optimization of lora adapter co-serving for large language models. In *Proceedings of the 25th International Middleware Conference: Demos, Posters and Doctoral Symposium, Middleware 2024, Hong Kong, SAR, China, December 2-6, 2024*, pages 27–28. ACM, 2024.

[18] Lequn Chen, Zihao Ye, Yongji Wu, Danyang Zhuo, Luis Ceze, and Arvind Krishnamurthy. Punica: Multi-tenant lora serving. In *Proceedings of the Seventh Annual Conference on Machine Learning and Systems, MLSys 2024, Santa Clara, CA, USA, May 13-16, 2024*. mlsys.org, 2024.

[19] Mostafa Dehghani, Basil Mustafa, Josip Djolonga, Jonathan Heek, Matthias Minderer, Mathilde Caron, Andreas Steiner, Joan Puigcerver, Robert Geirhos, Ibrahim M. Alabdulmohsin, Avital Oliver, Piotr Padlewski, Alexey A. Gritsenko, Mario Lucic, and Neil Houlsby. Patch n' pack: Navit, a vision transformer for any aspect ratio and resolution. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*, 2023.

[20] Aditya Dhakal, Sameer G. Kulkarni, and K. K. Ramakrishnan. GSLICE: controlled spatial sharing of gpus for a scalable inference platform. In *SoCC '20: ACM Symposium on Cloud Computing, Virtual Event, USA, October 19-21, 2020*, pages 492–506. ACM, 2020.

[21] Jiangfei Duan, Runyu Lu, Haojie Duanmu, Xiuhong Li, Xingcheng Zhang, Dahua Lin, Ion Stoica, and Hao Zhang. Muxserve: Flexible spatial-temporal multiplexing for multiple LLM serving. In *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*. OpenReview.net, 2024.

[22] Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, Anirudh Goyal, Anthony Hartshorn, Aobo Yang, Archi Mitra, Archie Sravankumar, Artem Korenev, Arthur Hinsvark, Arun Rao, Aston Zhang, Aurélien Rodriguez, Austen Gregerson, Ava Spataru, Baptiste Rozière, Bethany Biron, Binh Tang, Bobbie Chern, Charlotte Caucheteux, Chaya Nayak, Chloe Bi, Chris Marra, Chris McConnell, Christian Keller, Christophe Touret, Chunyang Wu, Corinne Wong, Cristian Canton Ferrer, Cyrus Nikolaidis, Damien Allonsius, Daniel Song, Danielle Pintz, Danny Livshits, David Esiobu, Dhruv Choudhary, Dhruv Mahajan, Diego Garcia-Olano, Diego Perino, Dieuwke Hupkes, Egor Lakomkin, Ehab AlBadawy, Elina Lobanova, Emily Dinan, Eric Michael Smith, Filip Radenovic, Frank Zhang, Gabriel Synnaeve, Gabrielle Lee, Georgia Lewis Anderson, Graeme Nail, Grégoire Mialon, Guan Pang, Guillem Cucurell, Hailey Nguyen, Hannah Korevaar, Hu Xu, Hugo Touvron, Iliyan Zarov, Imanol Arrieta Ibarra, Isabel M. Kloumann, Ishan Misra, Ivan Evtimov, Jade Copet, Jaewon Lee, Jan Geffert, Jana Vranes, Jason Park, Jay Mahadeokar, Jeet Shah, Jelmer van der Linde, Jennifer Billock, Jenny Hong, Jenya Lee, Jeremy Fu, Jianfeng Chi, Jianyu Huang, Jiawen Liu, Jie Wang, Jiecao Yu, Joanna Bitton, Joe Spisak, Jongsoo Park, Joseph Rocca, Joshua Johnstun, Joshua Saxe, Junteng Jia, Kalyan Vasuden Alwala, Kartikeya Upasani, Kate Plawiak, Ke Li, Kenneth Heafield, Kevin Stone, and et al. The llama 3 herd of models. *CoRR*, abs/2407.21783, 2024.

[23] Arpan Gujarati, Reza Karimi, Safya Alzayat, Wei Hao, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace. Serving dnns like clockwork: Performance predictability from the bottom up. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*, pages 443–462. USENIX Association, 2020.

[24] Mingcong Han, Hanze Zhang, Rong Chen, and Haibo Chen. Microsecond-scale preemption for concurrent gpu-accelerated DNN inferences. In *16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2022, Carlsbad, CA, USA, July 11-13, 2022*, pages 539–558. USENIX Association, 2022.

[25] Yongjun He, Jiacheng Lu, and Tianzheng Wang. Corobase: Coroutine-oriented main-memory database engine. *Proc. VLDB Endow.*, 14(3):431–444, 2020.

[26] Yongjun He, Yao Lu, and Gustavo Alonso. Deferred continuous batching in resource-efficient large language model serving. In *Proceedings of the 4th Workshop on Machine Learning and Systems, EuroMLSys 2024, Athens, Greece, 22 April 2024*, pages 98–106. ACM, 2024.

[27] Connor Holmes, Masahiro Tanaka, Michael Wyatt, Ammar Ahmad Awan, Jeff Rasley, Samyam Rajbhandari, Reza Yazdani Aminabadi, Heyang Qin, Arash Bakhtiari, Lev Kurilenko, and Yuxiong He. Deepspeed-fastgen: High-throughput text generation for llms via MII and deepspeed-inference. *CoRR*, abs/2401.08671, 2024.

[28] Michelle Horton. Optimizing drug discovery with cuda graphs, coroutines, and gpu workflows. https://developer.nvidia.com/blog/optimizing-drug-discovery-with-cuda-graphs-coroutines-and-gpu-workflows/, 2024.

[29] Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and

Weizhu Chen. Lora: Low-rank adaptation of large language models. In *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022*. OpenReview.net, 2022.

[30] Qinghao Hu, Meng Zhang, Peng Sun, Yonggang Wen, and Tianwei Zhang. Lucid: A non-intrusive, scalable and interpretable scheduler for deep learning training jobs. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2023, Vancouver, BC, Canada, March 25-29, 2023*, pages 457–472. ACM, 2023.

[31] Yuyang Huang, Yuhan Liu, Haryadi S. Gunawi, Beibin Li, and Changho Hwang. Alchemist: Towards the design of efficient online continual learning system. *CoRR*, abs/2503.01066, 2025.

[32] Saki Imai, Rina Nakazawa, Marcelo Amaral, Sunyanan Choochotkaew, and Tatsuhiro Chiba. Predicting llm inference latency: A roofline-driven ml method. In *Workshop on ML for Systems at NeurIPS*, 2024.

[33] Aditya K. Kamath, Ramya Prabhu, Jayashree Mohan, Simon Peter, Ramachandran Ramjee, and Ashish Panwar. Pod-attention: Unlocking full prefill-decode overlap for faster LLM inference. In Lieven Eeckhout, Georgios Smaragdakis, Katai Liang, Adrian Sampson, Martha A. Kim, and Christopher J. Rossbach, editors, *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2025, Rotterdam, Netherlands, 30 March 2025 - 3 April 2025*, pages 897–912. ACM, 2025.

[34] Mario Michael Krell, Matej Kosec, Sergio P Perez, and Andrew Fitzgibbon. Efficient sequence packing without cross-contamination: Accelerating large language models without impacting performance. *CoRR*, abs/2107.02027, 2021.

[35] Achintya Kundu, Rhui Dih Lee, Laura Wynter, Raghu Kiran Ganti, and Mayank Mishra. Enhancing training efficiency using packing with flash attention. *CoRR*, abs/2407.09105, 2024.

[36] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles, SOSP 2023, Koblenz, Germany, October 23-26, 2023*, pages 611–626. ACM, 2023.

[37] Brian Lester, Rami Al-Rfou, and Noah Constant. The power of scale for parameter-efficient prompt tuning. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 7-11 November, 2021*, pages 3045–3059. Association for Computational Linguistics, 2021.

[38] Haoyang Li, Fangcheng Fu, Sheng Lin, Hao Ge, Xuanyu Wang, Jiawen Niu, Jie Jiang, and Bin Cui. Demystifying workload imbalances in large transformer model training over variable-length sequences. *CoRR*, abs/2412.07894, 2024.

[39] Xiang Lisa Li and Percy Liang. Prefix-tuning: Optimizing continuous prompts for generation. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing, ACL/IJCNLP 2021, (Volume 1: Long Papers), Virtual Event, August 1-6, 2021*, pages 4582–4597. Association for Computational Linguistics, 2021.

[40] Zhuohan Li, Siyuan Zhuang, Shiyuan Guo, Danyang Zhuo, Hao Zhang, Dawn Song, and Ion Stoica. Terapipe: Token-level pipeline parallelism for training large-scale language models. In Marina Meila and Tong Zhang, editors, *Proceedings of the 38th International Conference on Machine Learning, ICML 2021, 18-24 July 2021, Virtual Event*, volume 139 of *Proceedings of Machine Learning Research*, pages 6543–6552. PMLR, 2021.

[41] Haokun Liu, Derek Tam, Mohammed Muqeeth, Jay Mohta, Tenghao Huang, Mohit Bansal, and Colin Raffel. Few-shot parameter-efficient fine-tuning is better and cheaper than in-context learning. In *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*, 2022.

[42] Xiao Liu, Kaixuan Ji, Yicheng Fu, Weng Tam, Zhengxiao Du, Zhilin Yang, and Jie Tang. P-tuning: Prompt tuning can be comparable to fine-tuning across scales and tasks. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers), ACL 2022, Dublin, Ireland, May 22-27, 2022*, pages 61–68. Association for Computational Linguistics, 2022.

[43] Kshiteej Mahajan, Arjun Balasubramanian, Arjun Singhvi, Shivaram Venkataraman, Aditya Akella, Amar Phanishayee, and Shuchi Chawla. Themis: Fair and efficient GPU cluster scheduling. In *17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2020, Santa Clara, CA, USA, February 25-27, 2020*, pages 289–304. USENIX Association, 2020.

[44] Xupeng Miao, Gabriele Oliaro, Xinhao Cheng, Mengdi Wu, Colin Unger, and Zhihao Jia. Flexllm: A system for co-serving large language model inference and parameter-efficient finetuning. *CoRR*, abs/2402.18789, 2024.

[45] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, Phillip B. Gibbons, and Matei Zaharia. Pipedream: generalized pipeline parallelism for DNN training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019*, pages 1–15. ACM, 2019.

[46] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Z. Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pages 8024–8035, 2019.

[47] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake VanderPlas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Edouard Duchesnay. Scikit-learn: Machine learning in python. *J. Mach. Learn. Res.*, 12:2825–2830, 2011.

[48] Ying Sheng, Shiyi Cao, Dacheng Li, Coleman Hooper, Nicholas Lee, Shuo Yang, Christopher Chou, Banghua Zhu, Lianmin Zheng, Kurt Keutzer, Joseph Gonzalez, and Ion Stoica. Slora: Scalable serving of thousands of lora adapters. In *Proceedings of the Seventh Annual Conference on Machine Learning and Systems, MLSys 2024, Santa Clara, CA, USA, May 13-16, 2024*. mlsys.org, 2024.

[49] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *CoRR*, abs/1909.08053, 2019.

[50] Jovan Stojkovic, Chaojie Zhang, Íñigo Goiri, Josep Torrellas, and Esha Choukse. Dynamollm: Designing LLM inference clusters for performance and energy efficiency. In *IEEE International Symposium on High Performance Computer Architecture, HPCA 2025, Las Vegas, NV, USA, March 1-5, 2025*, pages 1348–1362. IEEE, 2025.

[51] Charla Stracener, Maroun Touma, and Praneet Adusumilli. Foundation models at the edge. https://www.ibm.com/think/insights/edge-ai-strategy, 2023.

[52] Foteini Strati, Xianzhe Ma, and Ana Klimovic. Orion: Interference-aware, fine-grained GPU sharing for ML applications. In *Proceedings of the Nineteenth European Conference on Computer Systems, EuroSys 2024, Athens, Greece, April 22-25, 2024*, pages 1075–1092. ACM, 2024.

[53] Rohan Taori, Ishaan Gulrajani, Tianyi Zhang, Yann Dubois, Xuechen Li, Carlos Guestrin, Percy Liang, and Tatsunori B. Hashimoto. Stanford alpaca: An instruction-following llama model. https://github.com/tatsu-lab/stanford_alpaca, 2023.

[54] torchtune maintainers and contributors. torchtune: Pytorch's finetuning library, April 2024.

[55] Nandita Vijaykumar, Kevin Hsieh, Gennady Pekhimenko, Samira Manabi Khan, Ashish Shrestha, Saugata Ghose, Adwait Jog, Phillip B. Gibbons, and Onur Mutlu. Zorua: A holistic approach to resource virtualization in gpus. In *49th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2016, Taipei, Taiwan, October 15-19, 2016*, pages 15:1–15:14. IEEE Computer Society, 2016.

[56] Shuhe Wang, Guoyin Wang, Jiwei Li, Eduard H. Hovy, and Chen Guo. Packing analysis: Packing is more appropriate for large models or datasets in supervised fine-tuning. *CoRR*, abs/2410.08081, 2024.

[57] Yuxin Wang, Yuhan Chen, Zeyu Li, Xueze Kang, Yuchu Fang, Yeju Zhou, Yang Zheng, Zhenheng Tang, Xin He, Rui Guo, Xin Wang, Qiang Wang, Amelie Chi Zhou, and Xiaowen Chu. Burstgpt: A real-world workload dataset to optimize llm serving systems, 2024.

[58] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, Fan Yang, and Lidong Zhou. Gandiva: Introspective cluster scheduling for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*, pages 595–610. USENIX Association, 2018.

[59] Wencong Xiao, Shiru Ren, Yong Li, Yang Zhang, Pengyang Hou, Zhi Li, Yihui Feng, Wei Lin, and Yangqing Jia. Antman: Dynamic scaling on GPU clusters for deep learning. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*, pages 533–548. USENIX Association, 2020.

[60] Yi Yang, Ping Xiang, Mike Mantor, Norm Rubin, and Huiyang Zhou. Shared memory multiplexing: a novel way to improve GPGPU throughput. In *International Conference on Parallel Architectures and Compilation Techniques, PACT '12, Minneapolis, MN, USA - September 19 - 23, 2012*, pages 283–292. ACM, 2012.

[61] Xiaozhe Yao, Qinghao Hu, and Ana Klimovic. Deltazip: Efficient serving of multiple full-model-tuned llms. In *Proceedings of the Twentieth European Conference on Computer Systems, EuroSys 2025, Rotterdam, The Netherlands, 30 March 2025 - 3 April 2025*, pages 110–127. ACM, 2025.

[62] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. Orca: A distributed serving system for transformer-based generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2022, Carlsbad, CA, USA, July 11-13, 2022*, pages 521–538. USENIX Association, 2022.

[63] Binhang Yuan, Yongjun He, Jared Davis, Tianyi Zhang, Tri Dao, Beidi Chen, Percy Liang, Christopher Ré, and Ce Zhang. Decentralized training of foundation models in heterogeneous environments. In Sanmi Koyejo, S. Mohamed, A. Agarwal, Danielle Belgrave, K. Cho, and A. Oh, editors, *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*, 2022.

[64] Zijian Zhang. Understanding gpu architecture implications on llm serving workloads. Master's thesis, ETH Zurich, 2024.

[65] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric P. Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. Judging llm-as-a-judge with mt-bench and chatbot arena. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*, 2023.

[66] Shaokun Zheng, Xin Chen, Zhong Shi, Ling-Qi Yan, and Kun Xu. GPU coroutines for flexible splitting and scheduling of rendering tasks. *ACM Trans. Graph.*, 43(6):281:1–281:24, 2024.

A Artifact Appendix

The artifact is publicly available at https://github.com/llm-db/llmstation/tree/atc25-artifact. Details regarding software and hardware requirements, installation, and experiment workflows can be found in the `README.md` file.

Colocating ML Inference and Training with Fast GPU Memory Handover

Jiali Wang, Yankui Wang, Mingcong Han, Rong Chen
Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University

Abstract

This paper presents SIRIUS, an efficient colocation system that enables spatial sharing of GPU resources between machine learning (ML) inference and training tasks. To meet strict latency SLOs, SIRIUS prioritizes inference tasks, allowing them to utilize all GPU resources without restriction and interference. Meanwhile, it concurrently runs training tasks on leftover resources to improve throughput and GPU utilization. SIRIUS is novel in three ways. First, it leverages the characteristics of gradient computation in a batch to adjust the memory consumption of training tasks in a few milliseconds. Second, it explicitly manages memory reclamation for training, ensuring a thorough and safe memory handover process. Third, it employs an SLO-aware memory reallocation strategy to mitigate memory initialization overhead and prevent thrashing when facing frequently fluctuating workloads. Our evaluation shows that SIRIUS outperforms existing state-of-the-art colocation approaches, improving inference SLO compliance by an average of 57.0% (up to 97.0%) and training throughput by 2.2× (up to 13.7×).

1 Introduction

Serving machine learning (ML) inference and training workloads has become one of the most critical workloads in cloud data centers. High-performance GPUs serve as the predominant accelerators in ML-as-a-Service (MLaaS) platforms, catering to the growing computational demands for ML tasks [7, 8, 42, 47, 87]. However, these costly GPUs are often inefficiently utilized when serving inference with dynamic and bursty workloads. Inference serving is latency-sensitive and must adhere to strict Service-Level Objectives (SLOs) for latency (e.g., 100 ms [34]). To meet these SLOs even during sharp bursts—which can surge to 50× the average [56]—MLaaS platforms typically allocate dedicated, over-provisioned GPU resources for inference serving [33, 93]. This strategy, however, results in severe GPU underutilization (e.g., below 15% [44]) for ML inference workloads [32, 37, 49, 87].

Colocating heterogeneous workloads, such as latency-sensitive online services and throughput-intensive batch jobs, is a common practice to enhance resource utilization [60, 80]. Industrial products, such as Google Kubernetes Engine [14] and Tencent Cloud qGPU [15], demonstrate this by colocating high-priority inference tasks with low-priority training tasks to improve GPU utilization, serving as a selling point for MLaaS platforms. Furthermore, colocating ML inference and training workloads has also been explored in recent research work [16, 62, 63, 75, 97]. One approach, known as *temporal sharing*, enables inference and training tasks to share GPU resources in different time slices. For example, Lyra [53] colocates inference and training workloads within the same GPU cluster by dynamically allocating varying numbers of GPU servers for each task. Similarly, PipeSwitch [16] alternates between inference and training execution on a single GPU.

However, both inference and training tasks are memory-intensive, consuming substantial GPU memory to store models and intermediate states [12, 21, 28, 57, 91, 92]. Consequently, temporal sharing inevitably incurs significant overhead due to time-consuming context switching between these tasks. Unlike CPUs, GPUs require memory switching because of their limited capacity (typically tens of GBs). For inference tasks, reinitializing contexts and reloading models into GPU memory can cause significant latency overhead—up to a few seconds [16, 26, 91]—likely violating inference SLOs. For training tasks, preempting execution without saving context for fast switching can result in starvation or even failure. More importantly, temporal sharing cannot improve GPU utilization during inference serving.

Concurrently running ML inference and training tasks on the same GPU, known as *spatial sharing*, is an attractive alternative with the potential to fully utilize GPUs and avoid context switching. Modern GPUs allow dynamic partitioning of GPU compute units (i.e., Streaming Multiprocessors, SMs) between multiple tasks (e.g., ML training and inference) by setting SM masks within 1 millisecond [17]. Furthermore, recent studies have demonstrated the feasibility of preventing interference between inference and training tasks when sharing GPU computing resources [17, 36, 72, 75, 89].

Spatially sharing GPU memory between inference and training tasks remains challenging. *Statically partitioning* GPU memory for these memory-intensive tasks inevitably compromises performance for both, even with carefully tuned proportions. This approach is particularly ineffective for dynamic workloads—a hallmark of inference serving. An alternative is *dynamic swapping*, which transfers data between GPU and host memory as needed. This can be done either proactively through CPU offloading [16, 48, 51, 91] or passively via memory oversubscription mechanism (i.e., Unified Memory, UM [29, 39]). However, data transfer between GPU and host memory can significantly degrade performance due to limited PCIe bandwidth. For example, UM-based colocation suffers 93% degradation in inference SLOs, compared to running inference tasks alone, even under low workloads.

Our approach. We observe that the actual memory requirements of inference tasks fluctuate with the dynamic inference workloads. Without compromising inference SLOs and training throughput, the inherent elasticity of training tasks [58, 61, 66] provides an opportunity for *dynamic GPU memory sharing* between inference and training tasks—reconfiguring training tasks at runtime to meet the varying memory requirements of inference tasks.

This paper presents SIRIUS, an efficient colocation system that enables spatial sharing of GPU resources between ML inference and training tasks. SIRIUS treats inference tasks as first-class citizens to utilize all GPU resources without restriction and interference, satisfying strict latency SLOs. Meanwhile, it concurrently runs training tasks on leftover resources as much as possible to improve throughput and GPU utilization.

Key challenges. The handover of GPU memory from a training task to an inference task involves three steps: (1) waiting for the current training batch to complete, (2) transferring the released memory from the training task to the inference task, and (3) initializing the memory for the inference task. This process usually takes several hundred milliseconds (typically 250 ms, see §3), which hardly meets the inference SLOs (e.g., 100 ms). Therefore, we introduce a millisecond-scale reallocation mechanism for dynamic GPU memory sharing between inference and training tasks, built on three key techniques:

Instant memory adjustment (§4.1). We discover that the execution of a training batch can be divided into two distinct phases: gradient computation (GC) and model updating (MU). GC dominates the training execution time (over 95%) but does not change the training states. In contrast, MU requires atomic execution but commonly takes less than 10 ms. Therefore, SIRIUS can adjust the memory consumption of training tasks in just a few milliseconds (5 ms on average) by either instantly discarding the executing batch during GC phases or briefly waiting for MU phases to complete.

Safe memory handover (§4.2). To reduce memory handover time, SIRIUS directly transfers GPU memory released by training tasks to inference tasks, bypassing the memory cache in training frameworks and costly processes in the GPU runtime (e.g., 34 ms). However, this approach may result in *data pollution* due to synchronization issues between allocation and computation. To remedy this, SIRIUS explicitly manages memory reclamation, ensuring a thorough and safe memory handover process.

SLO-aware memory reallocation (§4.3). To guarantee inference SLOs under fluctuating workloads, SIRIUS reallocates memory between inference and training tasks in an SLO-aware manner. SIRIUS reserves GPU memory for inference tasks to tolerate memory re-initialization overhead caused by memory thrashing. Moreover, SIRIUS reallocates memory in a coarse-grained way to minimize disruptions to training tasks (e.g., 1.4% overhead caused by reallocation).

We implemented SIRIUS as a new inference engine with a plugin for training frameworks [55], aiming to colocate inference and training tasks on GPUs while ensuring inference SLOs and improving training throughput. We evaluate SIRIUS with four generated and two real-world traces [71, 85], comparing it with different GPU memory sharing approaches. Our experimental results show that SIRIUS outperforms these approaches by an average of 57.0% (up to 97.0%) improvement in inference SLO compliance and 2.2× (up to 13.7×) speedup in training throughput. Furthermore, SIRIUS can achieve up to 98% of inference SLO compliance of running inference tasks alone, even under a high-intensity workload.

2 Background and Motivation

Inference and training are two major tasks in machine learning services. To clarify the problem of colocating inference and training in MLaaS platforms, we begin by analyzing the unique characteristics and demands of both workloads.

2.1 ML Inference

Underutilized GPU computing resources. To ensure strict latency SLOs of inference services (i.e., on the order of milliseconds [34, 73]), GPU servers are typically overprovisioned. However, due to the dynamic and fluctuating nature of inference workloads, GPU resources are often underutilized when there are insufficient inference requests to saturate all GPU computing units [23, 94, 95]. Figure 1(a) shows that GPU utilization can often remain low during serving with the MAF trace.[1] This underutilization presents a necessity to colocate inference services with training tasks to effectively saturate GPU resources.

Inefficient GPU memory consumption. To maintain low inference latency, GPU memory is typically dedicated to storing inference models. The growing number of deployed models (e.g., more than one million models in HuggingFace model hub [10]) incurs considerable pressure on GPU memory resources. Additionally, the recent emergence of Large Language Models (LLMs) further intensifies this memory pressure due to their KV caches for requests [51, 67, 76]. However, we observe that the dedicated GPU memory (11.9 GB) often surpasses the actual memory requirements (4.4 GB on average), as shown in Figure 1(a). This overconsumption of GPU memory prevents the opportunity to share GPU resources with training tasks, despite the fact that computing resources may remain underutilized.

Dynamic memory requirements. Memory requirements of inference workloads fluctuate over time (e.g., from 1.8 GB to 10.3 GB). This is presented by the changes in the number of requested models, or the volume of generated KV caches. Statically allocating memory that is less than the peak requirement, e.g., based on the *average* memory consumption, can

[1] Similar to prior work [34, 48, 56], we simulate inference model requests using serverless functions. As the original trace is collected from serverless functions, we set the maximum requests per second as 150 reqs/s.

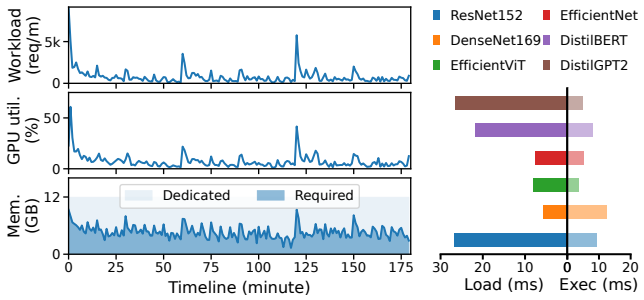


Figure 1: *(a) The GPU utilization and memory consumption during serving 64 different models with MAF [71] trace.* Total *stands for the memory consumption of all deployed inference models, and* Required *stands for the actual memory consumption required to serve inference requests. (b) The model loading (*Load*) and inference execution (*Exec*) time of six different models on NVIDIA V100.*

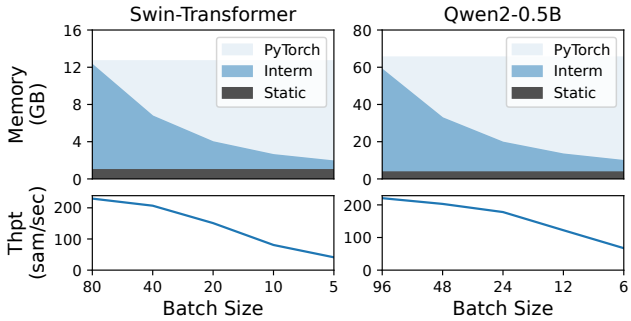


Figure 2: *The memory consumption (GB) and throughput (samples processed per second) for training Swin-Transformer [59] on NVIDIA V100 and Qwen2-0.5 [13] on NVIDIA A100 with different batch sizes.*

severely violate SLOs. Because inference tasks must queue up to use GPU memory when their requirements exceed the allocation. For example, when the allocated GPU memory fails to store all requested models, inference tasks must swap models between GPU memory and host memory, which introduces slow model loading (aka. *cold start*) in the critical path of inference services. Figure 1(b) shows that loading DistilGPT2 [70] (26.6 ms) takes 5.3× longer than its execution (5.0 ms).

2.2 ML Training

To avoid wasted GPU resources of inference serving under low load, colocating inference services with computing-intensive training tasks can ideally saturate GPU resources, which has gained significant attention in recent years [16, 75, 88, 89, 92, 97]. However, due to the large memory consumption of inference services, there is often not enough memory reserved for training tasks, making the colocation inefficient or even infeasible. Fortunately, we observe that the unique characteristics of training tasks bring an opportunity to efficiently share GPU resources with inference tasks.

Memory consumption breakdown. An ML training task iteratively computes gradients (i.e., forward and backward) and applies them to update the model parameters. Figure 2 shows memory consumption during the training of Swin-Transformer (Swin-T) [59] and Qwen2-0.5B models [13].[2] The memory consumption of a training task can be categorized into two types: (1) the static training states, which includes model parameters and optimizer states that store training progress, representing the minimum memory required for training; and (2) the intermediate results, which are consumed during the computation of gradients.

Elastic memory consumption. It can be found that the memory consumption of training is primarily dominated by the intermediate results (91% for Swin-T with batch size of 80). Furthermore, by reducing the batch size during training execution, the memory consumption of intermediate results can be effectively controlled (proportional to the batch size), finally adjusting the training memory consumption. This makes it possible to align the memory consumption of training tasks with the varying memory requirements of inference services by configuring the batch size. To guarantee training convergence, the hyper-parameters of training tasks (i.e., the effective batch size[3]) must remain unchanged. Fortunately, a widely-adopted technique—gradient accumulation—enables gradients to be accumulated over multiple smaller batches, maintaining a constant effective batch size for model parameter updating [31, 78].

However, the memory caching mechanism of ML frameworks (e.g., PyTorch [65]), which reduces the memory allocation overhead to improve training performance, leads to the unchanged memory occupation of training tasks even when the batch size is reduced. This causes the unused memory of training tasks invisible to inference tasks.

Memory demand for high throughput. As shown in Figure 2, having more memory resources enables training tasks to use larger batch sizes, which enhances training throughput by fully utilizing the GPU. However, colocating inference and training tasks can restrict the available memory for training, ultimately impacting training performance.

2.3 Colocation of ML Inference and Training

We review and discuss various approaches to colocating inference and training tasks, as illustrated in Figure 3, which contains two inference models (M_x) and one training task.

Task switching (temporal sharing). This approach enables inference and training tasks to temporally share GPU resources by switching GPU memory between tasks [16]. Therefore, an ML task can utilize the entire GPU memory and avoid memory contention, achieving the optimal performance during execution. However, this approach comes with significant context switching overhead. In Figure 3(a), to handle the first two inference requests (R_1 and R_2), GPU memory must be

[2]The constant memory consumed by the CUDA context is excluded.

[3]The effective batch size, also known as the global batch size in distributed training, represents the total number of samples used to compute gradients before updating model parameters.

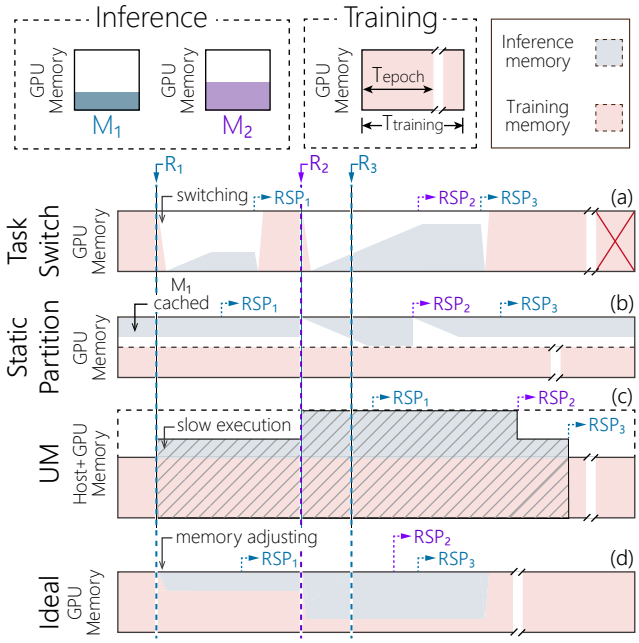


Figure 3: *An example of GPU memory consumption when using different approaches to colocate two inference models (M_x) and one training task. The illustrated time slice contains 3 inference requests (R_x) of two models. RSP_x marks the completion time of requests.*

switched from the training task to the inference tasks, leading to cold starts for inference models (M_1 and M_2). To guarantee inference SLOs, the training task may be preempted to facilitate fast switching. However, this can result in starvation, where training batches are rarely completed due to the frequent context switching when inference requests arrive continuously. Such frequent switching also causes additional cold starts for inference tasks (e.g., M_1 needs to be reloaded to serve R_3). Moreover, during inference serving, unused GPU resources may be wasted.

Static GPU memory partition. This approach statically partitions GPU memory for inference and training tasks to spatially share the GPU. Each task is restricted to using the memory allocated to it. This alleviates the interference between inference and training tasks due to memory contention, and avoids the overhead of task switching. However, the available memory for inference and training tasks is limited and fixed. In Figure 3(b), to serve the requests R_1 and R_2, the inference task must swap models between GPU memory and host memory, increasing the inference latency due to cold starts. Furthermore, training throughput is constrained by the limited memory. On the other hand, such static memory allocation can mismatch with the memory requirements of inference workloads, leading to memory waste under low load or inference SLO violations under high load.

Dynamic GPU memory swapping. This approach provides unlimited memory resources for inference and training tasks by swapping between GPU memory and host memory. Therefore, it not only meets the memory requirements of inference and training tasks but also ensures the optimal GPU utilization. However, executing ML tasks on both GPU memory and host memory significantly degrades performance. For example, as shown in Figure 3(c), the data traffic over PCIe (60× slower than GPU memory on NVIDIA A100) caused by memory swapping of Unified Memory (UM) significantly decelerates both tasks.

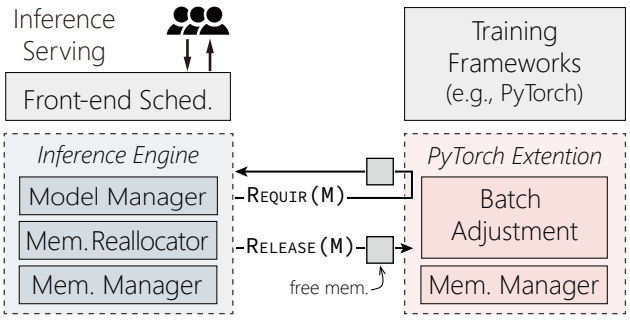


Figure 4: *The architecture of* SIRIUS. *Modules in boxes with dashed borders are the key components of* SIRIUS *for ML inference (blue) and training (red), other modules (gray) can be reused from existing ML frameworks. The interfaces for memory handover are labeled as* REQUIRE*(M) and* RELEASE*(M), where M represents the memory size.*

3 Overview of SIRIUS

Our approach: dynamic memory sharing via training elasticity. SIRIUS enables inference and training tasks to spatially share a GPU by exploiting the elasticity of training tasks. As shown in Figure 3(d), SIRIUS dynamically adjusts training memory consumption by reconfiguring the batch size to accommodate fluctuating inference tasks. As a result, inference tasks can fully utilize all GPU memory during high workloads, ensuring latency SLO. Meanwhile, training tasks can still utilize the remaining GPU memory, executing alongside inference tasks. This approach improves training throughput without compromising inference performance.

For example, to serve the requests R_1 and R_2 in Figure 3(d), the memory consumption of the training task will be swiftly adjusted by reducing the batch size. This allows inference tasks to utilize the released memory to store the model M_1 and M_2 to complete serving. Once all inference requests are finished, the training task can utilize all GPU memory to enhance the throughput by increasing the batch size.

System Overview. SIRIUS is designed as a comprehensive system for existing ML software stacks, which colocates inference and training tasks on multiple GPUs. On each GPU, the inference service process cooperates with the training task process to dynamically share GPU memory. As shown in Figure 4, SIRIUS consists of an inference engine and a plugin to the training framework. SIRIUS introduces the memory handover interface—REQUIRE(M)/RELEASE(M)—to enable

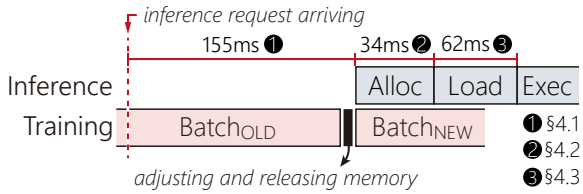


Figure 5: *The breakdown of the time for GPU memory handover from training to inference tasks.*

dynamic GPU memory sharing between ML inference and training tasks. When inference tasks require more memory, the inference engine calls REQUIRE(M) to meet the requirement by adjusting the training memory. Conversely, when less memory is required, the inference engine notifies the training task to utilize idle memory resources by calling RELEASE(M).

Inference Engine. The inference engine uses a standard ML-as-a-Service (MLaaS) interface, the same as existing inference servers (e.g., NVIDIA Triton [12]). To handle dynamic workloads effectively, the inference model manager monitors the memory requirements of online inference services. Memory resources consumed by unrequested models (i.e., idle models) and unused LLM KV caches will be reclaimed for training tasks. Furthermore, the fluctuating memory requirements of inference tasks lead to memory thrashing between inference and training tasks. This incurs cold start delays for inference tasks and frequent interruptions for training tasks. Therefore, SIRIUS introduces an SLO-aware memory reallocator to mitigate memory thrashing.

Extensions on Training Engine. The training plugin extends the popular training frameworks (i.e., PyTorch) to dynamically adjust the training task, while minimizing the training development effort. The main component of training extension is the batch adjustment, which provides a mechanism to adjust training memory consumption by reconfiguring the batch size. When receiving REQUIRE(M) from the inference engine, it swiftly and safely reconfigures the training tasks (i.e., decreasing the batch size) to hand over the memory to inference tasks. When receiving RELEASE(M), it will increase the batch size at the beginning of the next iteration to improve training throughput. To let training tasks utilize multiple GPUs efficiently, the effective batch is distributed across all GPUs with the consideration of load balancing.

Key challenge: fast GPU memory handover between training and inference tasks. When additional inference models are requested, memory must be reallocated from training to inference tasks timely to ensure SLO compliance. As shown in Figure 5, the memory handover process—consisting of adjusting training memory (❶, see §4.1), transferring memory to inference (❷, see §4.2), and initializing inference memory (❸, see §4.3)—takes approximately 250 ms. This delay significantly reduces inference SLO compliance.

Table 1: *Execution time (in milliseconds) breakdown of one training batch for different models on NVIDIA V100.*

Phase	ResNet-152	Swin-T-b	BERT-b	GPT2
Gradient Compute	323.7	344.2	241.5	188.7
Model Update	9.6	9.0	10.0	9.8

4 Fast GPU Memory Handover

In this section, we elaborate on the design of SIRIUS for reallocating GPU memory from ML training to inference.

4.1 Instant Memory Adjustment

To adjust training memory consumption through batch size reconfiguration, inference tasks must wait for the currently running batch to complete. Since training batches take hundreds of milliseconds while inference runs in just milliseconds, this waiting period is unacceptable for latency-sensitive inference tasks.

Based on the characteristics of model training, we propose an approach to instantly adjust memory consumption by discarding the currently running batch. We found that the execution of a training batch can be divided into two phases: gradient computation (i.e., forward and backward) and model parameter updating. During the gradient computation phase, the model parameters typically remain unchanged. However, the model updating phase must be executed atomically to prevent inconsistent training states. Therefore, discarding the running batch to achieve instant memory adjustment is safe as long as it is not in the model updating phase.

As shown in Table 1, gradient computation takes significantly longer than model parameter updating, dominating the training time (e.g., 97% for ResNet-152). This is because model updating involves only a few lightweight addition operations. During memory adjustments, most of the time, the training task is engaged in gradient computation. If an adjustment arises during model updating, waiting for this short phase to complete is acceptable.

We first introduce the execution process of a training batch before discussing how to discard the executing training batch. The execution of the training task involves both the host CPU and the GPU accelerator. The host CPU is responsible for the control plane, and the computation-intensive operators in the training computational graph are dispatched to the GPU. The GPU is responsible for the computation plane, where those operators are executed as asynchronous GPU kernels.

However, asynchronous execution makes it challenging to discard the training batch instantly. Since the CPU operator dispatching is much faster than GPU computation, numerous uncompleted kernels are executing on the GPU when the training task is notified of the memory adjustment. Therefore, simply synchronizing the GPU to halt model training incurs a lengthy wait for these in-flight kernels to complete. Ultimately, this delay causes inference tasks at risk of violating SLOs due to waiting for memory adjustments.

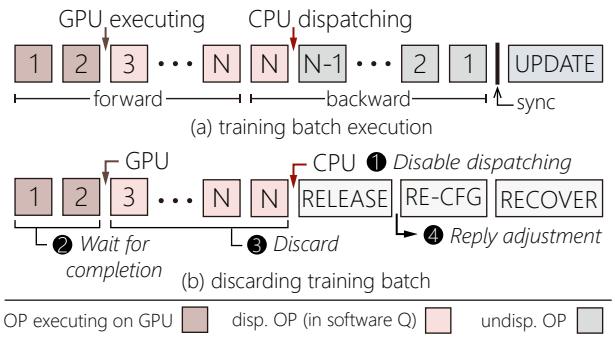


Figure 6: *Instant training memory adjustment.*

To harness the training asynchronous execution, SIRIUS employs the software GPU kernel management technique [36, 72, 75]. As shown in Figure 6(a), instead of directly launching training operators to GPU hardware, SIRIUS adds these operators to a software queue maintained by the host CPU. The operators in the software queue are gradually launched to the GPU for execution. Consequently, only a few in-flight operators are executing on the GPU, whereas most are in the software queue.

Furthermore, SIRIUS leverages the host CPU as the oracle to determine the training stages (i.e., GC or MU), as it manages the training control plane. However, the GPU can lag behind the host CPU and not fully complete gradient computation when the host CPU enters the model updating phase. Therefore, SIRIUS inserts a synchronization between the host CPU and GPU before model updating, ensuring their consistency in the training phase.

Figure 6(b) shows the process of discarding the training batch. When the memory adjustment occurs during the gradient computation, SIRIUS disables operator dispatching by refusing the operators to be added to the software queue (❶). SIRIUS then waits for a few executing GPU kernels to complete (❷). As training kernels are generally short, it only takes a short time. After this, SIRIUS discards the operators in the software queue to fully discard the training batch in the computation plane (❸).

Next, SIRIUS must ensure the training control plane is aware of the adjustment to release the training memory. To achieve this, SIRIUS lets the host CPU actively check for adjustments at each training operator. Because the operator dispatching on the host CPU is very fast, checking for adjustments will not become a bottleneck for the training task. Furthermore, as operators are the basic units of the computational graph, they are fine-grained enough to notify the host CPU of the adjustment in a timely manner.

Once the training memory is released, the memory adjustment will be replied (❹). Before recovering the training task, SIRIUS reconfigures the training task by reducing the batch size. SIRIUS then re-enables the operator dispatching and resets the control plane to the next iteration to recover the training task, where data samples of the discarded batch will be reprocessed.

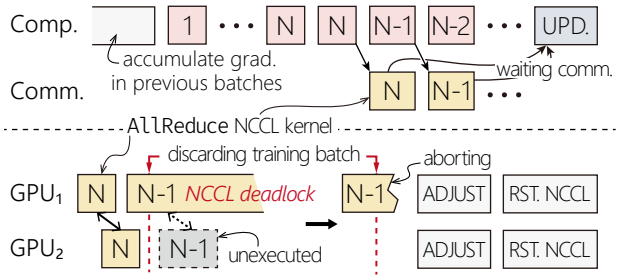


Figure 7: *The memory adjustment with data parallel training.*

To fully release the memory resources occupied by the discarded training batch, SIRIUS must traverse the training computational graph to release the corresponding intermediate results. This is necessary because discarding the training batch disrupts the normal training execution, causing the training task to fail to release these intermediate results during the backward stage.

Support for multi-GPU training. Data parallelism (DP) is a widely used technique to improve training throughput by distributing data samples across multiple GPUs [55]. Unlike single-GPU training, multi-GPU training with DP involves gradient synchronization (i.e., `AllReduce`[4]) across all GPUs in the backward stage, as shown in Figure 7. The training task must wait for gradient synchronization to complete before updating model parameters. When gradient accumulation is incorporated, the gradients are accumulated locally without initiating communication until the effective batch size is reached.

SIRIUS extends its memory adjustment approach from a single GPU to multiple GPUs, supporting training tasks with data parallelism. When memory adjustment is needed, the training task on every GPU receives notifications to initiate the process. Similar to single-GPU training, if a training task is computing gradients, the current batch will be discarded.

However, the execution speed of training varies slightly across different GPUs, leading to inconsistencies in the number of `AllReduce` NCCL kernels launched at any given timestamp. As shown at the bottom of Figure 7, one more `AllReduce` NCCL kernel is launched on GPU_1 when SIRIUS starts to discard the training batch. This presents an NCCL deadlock because the NCCL kernel on GPU_1 cannot receive the required data from GPU_2, which will not launch the corresponding `AllReduce` NCCL kernel after the discarding process starts. This ultimately causes the SIRIUS to fail at waiting for the completion of in-flight GPU kernels to finish discarding the training batch.

To address this issue, SIRIUS must abort the in-flight NCCL kernels. Simply aborting NCCL kernels by terminating NCCL connections between GPUs (i.e., calling `ncclCommAbort`)

[4] `AllReduce` is one of the collective communication primitives provided by NCCL [11]. When the training task initiates `AllReduce`, NCCL submits a GPU kernel to complete the communication.

hurts training throughput. It typically takes hundreds of milliseconds to re-establish NCCL connections to recover the training task.

Contrarily, SIRIUS leverages the NCCL communication mechanism to abort NCCL while retaining NCCL connections. NCCL maintains a counter between GPUs to ensure the order and correctness of collective data transmission. Moreover, to support the basic functionality of collective communication—connection termination, an abort flag is designed to let NCCL kernels exit despite ignoring the correctness of data transmission.

Therefore, SIRIUS sets the abort flag to abort NCCL kernels to fulfill training memory adjustment. To avoid NCCL deadlock or data pollution caused by incorrect NCCL states, SIRIUS resets the NCCL counter on all GPUs before starting the next training iterations, after which the training task will be recovered similarly to the single GPU case.

Batch distributing. The memory requirements for inference tasks can vary across different GPUs, which causes data parallel training to suffer from memory resource imbalances, resulting in a degradation in training throughput due to the straggling GPU. Therefore, SIRIUS must ensure the effective batch processing time is similar across all GPUs. SIRIUS achieves this by dynamically distributing the batch across GPUs to balance the effective batch processing time. SIRIUS uses training batch size to model the training throughput for each GPU through online profiling. SIRIUS redistributes the batch to maintain a balanced training time when the memory adjustment occurs, assigning more samples to faster GPUs. Furthermore, to prevent communication deadlock during training, SIRIUS ensures that all GPUs progress the training tasks (i.e., the training batch size of each GPU is non-zero), or none of GPUs progress the training task. The latter happens when all adjustable memory is allocated for inference tasks.

4.2 Safe Memory Handover

To transfer the released memory from the training task to the inference task, the simplest method is to return the memory cached by the training task to the GPU, then allocate memory from the GPU for the inference task. However, GPU memory allocation (i.e., `cudaMalloc`) is notoriously inefficient [16, 35, 83, 96], which incurs significant overhead to inference latency (see §3). Therefore, SIRIUS maintains a shared memory pool between inference and training tasks to bypass the memory caching mechanism of training frameworks and the inefficient GPU memory management, minimizing the overhead of memory handover. Furthermore, to protect data privacy, the memory transferred from the training task is filled with zeros before being allocated to the inference task, and vice versa. Due to high GPU memory bandwidth, the impact of zero-filling on memory handover latency is negligible. Nevertheless, the shared memory resources must be carefully managed to avoid data pollution.

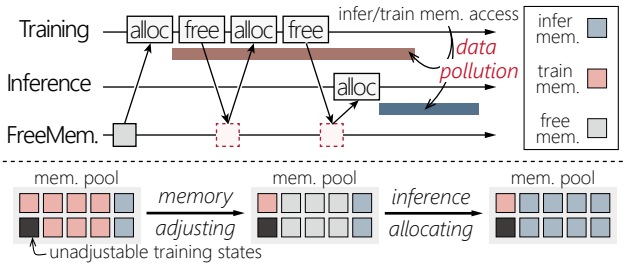


Figure 8: *The illustration of data pollution and memory handover between inference and training tasks.*

Data pollution. In the conventional training, the memory allocation and computation are asynchronous. Typically, the training tasks first allocate memory and then launch computational operators. Rather than waiting for the operators to complete, the memory is promptly released to training internal memory cache after launching. This design reduces GPU synchronization overhead and potentially lowers memory consumption by reusing memory across training operators (e.g., the second training memory allocation at the top of Figure 8).

However, when inference and training tasks allocate memory from the shared memory pool, such behavior can lead to *data pollution* between two tasks. As shown in Figure 8, once the training task releases its memory to the memory pool in the second free operation, this memory immediately appears to be available for the inference task. Then the inference task allocates this memory to serve inference requests. At the same time, however, the training task is still accessing this memory, leading to data pollution. This issue does not occur in solo training execution because no concurrent computation streams using the same memory area.

The root cause of data pollution is that the memory released by the training task is prematurely seen by the inference task, making the inference task mistakenly treat the memory being accessed by the training task as free memory. To address this issue, instead of aggressively polling for the real availability of training released memory, which incurs synchronization overhead to the training task. As shown at the bottom of Figure 8, SIRIUS maintains the memory ownership and only explicitly reclaims the training released memory for inference tasks when the memory adjustment occurs. Because there is no in-flight training operator after training memory adjustment (see §4.1), the safety of memory handover is guaranteed. If the memory adjustment does not occur, the memory released by the training task still belongs to the training task and is not visible to inference tasks.

4.3 SLO-aware Reallocation

When serving multiple inference models under fluctuating workloads, an idle model may soon be requested again. If such a model's memory is reclaimed for the training task, it can lead to memory allocation thrashing. On the one hand, this incurs frequent model loading to initialize memory for inference tasks, degrading the quality of inference services. On the

Algorithm 1: SLO-aware Memory Reallocation

Input: The maximum liveness time of idle models, $\mathcal{T}_{idle}$;
The watermark of memory reallocation, $\mathcal{W}$.
$\mathcal{M}_{rsv}$**:** The volume of reserved memory for inference tasks.

```
1  Function InferRelease(model)
2      if the model has been requested in the last T_idle then
3          return;
4      M_rsv += model_sz
5      if M_rsv < 2 × W then
6          reserve the model
7      else
           /* reallocate memory to training task */
8          release_sz = M_rsv − W
9          release the coldest models
10         M_rsv = W
11 Function InferRequire(model)
12     if M_rsv ≥ model_sz then
           /* use the reserved memory */
13         M_rsv −= model_sz
14     else if training task does not reach the adjustment limit then
           /* reallocate memory to inference task */
15         adjust_sz = (model_sz − M_rsv) + W
16         AdjustTrainingTask(adjust_sz)
17         M_rsv = W
18     else
19         fallback to model swapping
```

other hand, the training task will be frequently interrupted by inference tasks to fulfill memory adjustment, hurting training throughput. To address these issues, we present an SLO-aware reallocation approach, as shown in Algorithm 1.

To reclaim memory resources from inference tasks (i.e., `InferRelease`), SIRIUS maintains a maximum liveness time ($\mathcal{T}_{idle}$) of idle models. An idle model will be reclaimed if it has not been requested for $\mathcal{T}_{idle}$ time. Furthermore, SIRIUS reallocates memory in a coarse-grained way to minimize the disruptions to training tasks. SIRIUS maintains a watermark ($\mathcal{W}$) to control the minimum granularity of memory reallocation. Instead of directly releasing the memory resources of idle models to the training task, SIRIUS reserves these models until the volume of reserved memory ($\mathcal{M}_{rsv}$) reaches the releasing threshold—$2\mathcal{W}$ (`L5`). Therefore, when these models are requested again, SIRIUS can quickly complete the serving. When the releasing threshold is reached, SIRIUS releases the coldest model in the reserved memory until the volume of reserved memory is reduced to $\mathcal{W}$ (`L8-L10`).

When inference tasks require more memory to serve requested models (i.e., `InferRequire`), SIRIUS first uses the inference reserved memory (`L12`). If this reserved memory is exhausted, SIRIUS reallocates memory resources between inference and training tasks (`L15-L17`). Besides meeting the current memory requirements of inference tasks, SIRIUS

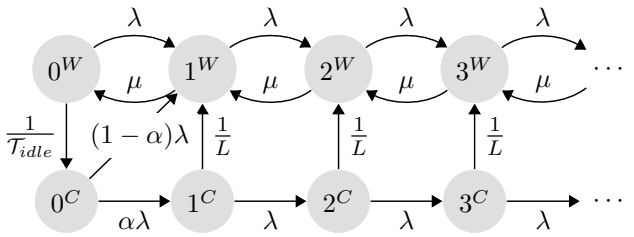


Figure 9: *The M/G/1 model for inference SLO compliance. x^W and x^C represent the state where there are x queueing inference requests without cold start and with cold start, respectively. λ is the arrival rate of inference requests. μ is the execution rate of inference tasks. α is the cold start ratio of state 0^C. L is the re-initialization time of the released model.*

adjusts $\mathcal{W}$ more memory resources to reduce the impact of the memory adjustment on training throughput. However, suppose the training task has reached the adjustment limit (e.g., the batch size has already been adjusted to zero). In that case, SIRIUS will serve inference requests by swapping models between GPU memory and host memory (`L19`). Finally, to mitigate model loading overhead, SIRIUS employs a pipelined execution scheme [16, 26] to overlap the inference execution with model loading.

To effectively mitigate cold start overhead for inference tasks and not hurt training throughput due to insufficient available memory, SIRIUS needs to be configured with the rational values of $\mathcal{T}_{idle}$ and $\mathcal{W}$ (see §6.4). Therefore, we leverage queueing theory to model the inference SLO compliance, finally determining SIRIUS's configuration.

To simplify modeling, we assume inference models are homogeneous (i.e., inference execution time and loading time are the same across models) and independent of each other. Consequently, the system can be modeled as a single-server queueing system, specifically M/G/1 [38]. In this model, inference requests follow a Poisson process with arrival rate λ, and the processing time of inference tasks follows a general distribution, accounting for potential re-initialization.

Figure 9 shows the M/G/1 model for inference SLO compliance. The expiration of the liveness time $\mathcal{T}_{idle}$ is represented by the transition from state 0^W to state 0^C with rate $\frac{1}{\mathcal{T}_{idle}}$. The watermark $\mathcal{W}$ is modeled by the cold start ratio α of state 0^C, which can be approximated as $1 - \frac{\mathcal{W}}{\sum \text{model_sz}}$, assuming the reserved memory follows a uniform distribution. Consequently, when the inference request arrives at state 0^C, the probability of transitioning to state 1^W and state 1^C is $1-\alpha$ and α, respectively. Finally, the re-initialization of the released model triggers the transition from state x^W to state x^C. The re-initialization time L includes the model loading overhead, and potential training adjustment time due to the coarse-grained reallocation, which can be approximated as the training adjustment time multiplied by α.

For every system state s, the inference response time is the sum of the waiting time for queueing inference requests and the processing time of the current inference request,

$$X_s \sim \begin{cases} \sum_{i=0}^{x} \exp(u) & s = x^W \\ \sum_{i=0}^{x} \exp(u) + \exp(\frac{1}{L}) & s = x^C \end{cases} \quad (1)$$

where X_s is the inference response time, following the hypo-exponential distribution. Finally, the inference SLO compliance—the percentage of inference requests that are served within the SLO—can be calculated by conditioning on states of the system,

$$P_{\text{SLO}} = \sum_s \pi_s P(X_s \leq \text{SLO}) \quad (2)$$

where π_s is the probability of state s, which can be calculated using the matrix method [38]. Therefore, given the inference SLO and compliance requirements, we can leverage the above model to search for a suitable configuration of $\mathcal{T}_{idle}$ and $\mathcal{W}$.

5 Implementation

For inference tasks, we built the inference server from scratch with approximately 6,000 lines of C++ code. The inference engine employs TVM [20, 98] as the backend for DNN models and vLLM [50] for LLMs. For training tasks, SIRIUS extends PyTorch with a Python library that contains around 5,000 lines of both Python and C++ code. The GPU memory and computing resource management is implemented as a common component of inference and training tasks, comprising roughly 6,000 lines of C++ code.

Memory management. SIRIUS uses GPU Virtual Memory Management (VMM) [1] to maintain a shared memory pool for both inference and training tasks, offering more flexibility than the native GPU memory management (i.e., `cudaMalloc`/`cudaFree`). SIRIUS establishes a contiguous memory area constituted by all available memory pages to form this shared memory pool. For inference tasks, SIRIUS allocates memory by directly dividing the memory area to avoid costly VMM calls. To prevent the memory fragmentation caused by the interleaving memory allocation of inference and training, SIRIUS allocates memory for the training task by mapping free, potentially non-contiguous memory pages in the memory pool to form a separate contiguous memory area.

When inference tasks request the memory adjustment, SIRIUS unmaps the unused memory pages from the training task to transfer memory resources to inference tasks, leading to the unmapping overhead in the adjustment critical path. To address this, SIRIUS merely updates the ownership of memory pages, postponing the actual unmapping until the next training memory allocation.

Dynamic GPU SM sharing. SIRIUS leverages the GPU hardware mechanism, specifically the SM mask [17], to allocate GPU computing resources between inference and training tasks. This mechanism enables dynamic specification of the Streaming Multiprocessors (SMs) used by GPU kernels, overcoming the limitation of static SM allocation of MPS [4].

Table 2: *Inference trace description.*

Workload	LIGHT	HEAVY	BURST	SKEWED
Request Rate	LOGN_L	LOGN_H	LOGN_{L+H}	LOGN_{L+H}
Model Dist.	Uniform	Uniform	Uniform	Zipfian

SIRIUS offline profiles the number of SMs that each inference model needs without degrading inference performance. To minimize the inference SLO violation during colocation, SIRIUS enables inference tasks to utilize all SMs. To reduce computation interference between inference and training tasks, SIRIUS limits the number of SMs allocated for the training task by subtracting the SMs required by inference tasks from the total number of SMs. When SIRIUS is about to execute inference for a request, SIRIUS decreases the number of SMs allocated to the training task by the amount needed for inference tasks. Upon completion of the inference task, SIRIUS restores these SMs to the training task.

6 Evaluation

Testbed. The experiments are conducted on a GPU server with two Intel Xeon Gold 6138 CPUs (total of 80 cores), 503 GB of DRAM, and four NVIDIA Tesla V100 GPUs with 16 GB of memory. The GPUs are fully connected by NVLinks. The GPU and CPU are connected by PCIe 3.0×16. The software environment of the server is configured with Ubuntu 20.04, Python v3.10, CUDA v11.6, PyTorch v2.1.2, TVM v0.14.

Comparing targets. We compare SIRIUS with the following memory resource sharing approaches in inference and training colocation. **TaskSwitch** represents systems like PipeSwitch [16], where GPU is temporally shared by inference and training tasks. We do not directly compare with vanilla PipeSwitch because it lacks support for multi-GPU training. Moreover, PipeSwitch terminates the entire training task when switching to inference tasks, degrading training performance significantly. Instead, we implement TaskSwitch based on SIRIUS and support switching from the training task to inference tasks by preempting the training batch. **Static Partition** refers to the approach of statically partitioning GPU memory for inference and training tasks. For this approach, we use NVIDIA Triton [12] for serving inference tasks and PyTorch [65] for running training tasks. We compare SIRIUS against two configurations of static partition: **SP-50** (50% GPU memory for inference tasks and 50% for training tasks) and **SP-75** (75% GPU memory for inference tasks and 25% for training tasks). **UM+MPS** represents the dynamic memory swapping approach, which extends GPU memory with the host memory using CUDA Unified Memory [39] and concurrently runs tasks using NVIDIA MPS [4]. We also use NVIDIA Triton and PyTorch to serve inference and training tasks for this approach, respectively.

Workloads. Table 2 shows four generated inference traces

Table 3: *ML model description.*

Inference	Model	Model Size	Exec. Time
DNN	ResNet-152 [40]	319 MB	9.4 ms
	DenseNet-169 [43]	84 MB	12.6 ms
	EfficientNetV2-s [77]	86 MB	5.3 ms
	EfficientViT-b2 [19]	109 MB	3.7 ms
	DistilBERT [70]	254 MB	8.0 ms
	DistilGPT2 [70]	317 MB	5.0 ms
LLM	Llama2-13B [79]	26 GB	170.2 ms (TTFT) 20.7 ms (TBT)
Training	**Model**	**Batch Size**	**Exec. Time**
DNN	Swin-T [59]	72	318 ms (1 batch)
LLM	Qwen2-0.5B [13]	72	346 ms (1 batch)

used for DNN model serving: LIGHT (L), HEAVY (H), BURST (B), and SKEWED (S). In these traces, the inference serving timeline is sliced into 20-second intervals. During each interval, inference requests arrive following a Poisson distribution [68, 69]. LIGHT represents a low-intensity workload where the request rate of each interval is sampled from a LogNormal distribution [64] with μ=1 and σ=1 (LOGN_L), yielding an average request rate of 3.4 requests per second (reqs/s). HEAVY represents a high-intensity workload where the request rate of each interval is sampled from a LogNormal distribution with μ=4.5 and σ=0.3 (LOGN_H), yielding an average request rate of 98.2 reqs/s. BURST represents a bursty inference workload by combining LIGHT and HEAVY workloads [90], with 30% of request rates sampled from LOGN_H and 70% from LOGN_L. The average request rate is 29.9 reqs/s. LIGHT, HEAVY and BURST workloads distribute inference requests uniformly on all models, and SKEWED assigns requests to various models following a Zipfian distribution (α=1.05). Furthermore, we use a real-world trace from Microsoft Azure Functions (MAF) [71] to simulate a realistic arrival distribution in cloud data centers. We compressed the MAF workload by scaling one minute to 5 seconds. All workloads run for a total of 300 seconds.

For inference tasks, we use six widely-deployed DNN models, as shown in Table 3. All models are compiled using TVM before deployment. To simulate different customized variants of standard models, we duplicate these models up to 56 instances in a round-robin manner. For the training task, we use Swin-Transformer (Swin-T) [59]. To evaluate the performance of SIRIUS with LLMs, we employ Llama2-13B for inference tasks and Qwen2-0.5B for the training task using the BurstGPT [85] trace (see §6.6 for details).

Metrics. For inference, we measure the P99 latency and SLO compliance of all requests. The inference SLO is set as four times of inference model standalone execution time (see Table 3). For training, we measure the value of training throughput (i.e., samples per second).

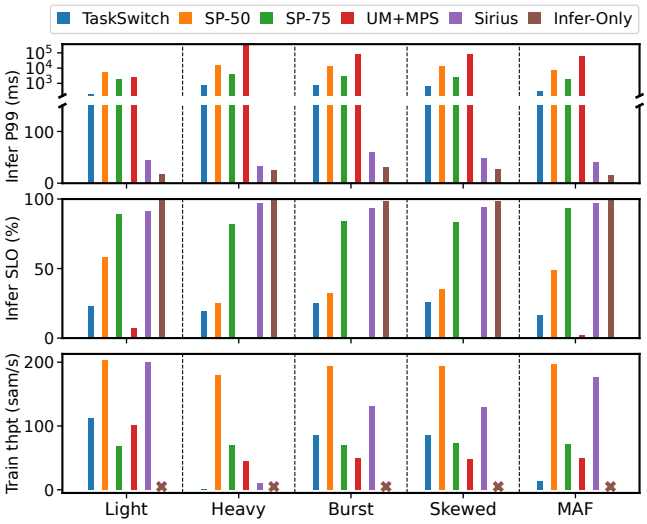


Figure 10: *The comparison of inference P99 latency, SLO compliance, and training throughput for different approaches with various workloads on one GPU.*

6.1 Overall Performance on A Single GPU

We compare the overall performance of SIRIUS with its competitors using various workloads (see Figure 10). To configure SIRIUS with the above SLO requirement, we use the queueing model with averaged inference model attributes to find values of $\mathcal{T}_{idle}$ and $\mathcal{W}$ that achieve an averaged SLO compliance of 90% across all workloads. The result configuration is $\mathcal{T}_{idle}$ = 5 seconds and $\mathcal{W}$ = 1 GB. Overall, SIRIUS improves the inference SLO compliance by an average of 57.0% (up to 97.0%) and improves the training throughput by an average of 2.2× (up to 13.7×) compared with all competitors.

First, compared with TaskSwitch, SIRIUS improves the P99 latency and SLO compliance by averagely 12.6× and 72.6% respectively. The low inference serving quality of TaskSwitch is mainly caused by the cold starts due to the context switch. For example, the inference task suffers a 38% higher cold start ratio than SIRIUS under BURST workload. For training throughput, SIRIUS outperforms TaskSwitch by averagely 4.6×. Due to preempting the training task, TaskSwitch degrades training throughput by 13.7× under MAF workload compared with SIRIUS, despite the GPU is not fully utilized by inference tasks. Contrarily, only 1.4% of computation time is wasted due to the discarded batch in SIRIUS.

Second, compared with SP-50, SIRIUS improves the P99 latency and SLO compliance by averagely 261.9× and 54.6% respectively. Although SP-50 has higher training throughput, it fails to meet the memory requirements of inference tasks, resulting in only 25% SLO compliance under HEAVY workload. Compared with SP-75, SIRIUS improves the P99 latency and SLO compliance by averagely 64.3× and 8.2% respectively. We found that inference model cold start overhead of Triton can be as high as 2 seconds, leading to high inference latency of SP-75 even with LIGHT workload. SP-75 allocates

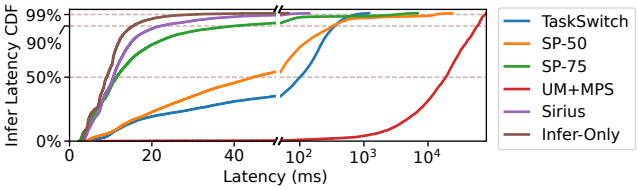


Figure 11: *Inference latency CDF for different approaches with* BURST *workload.*

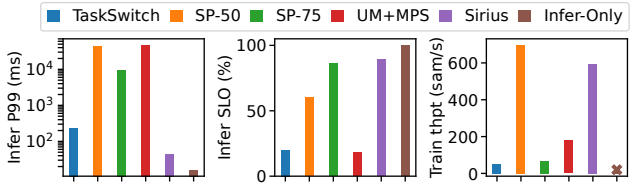


Figure 12: *The comparison of inference P99 latency, SLO compliance, and training throughput with* LIGHT *workload on four GPUs.*

more memory for inference tasks, leading to 3.0× training throughput degradation with LIGHT workload.

Third, compared with UM+MPS, SIRIUS improves the P99 latency and SLO compliance by $3.1k\times$ and 92.6%, respectively. The poor performance of UM+MPS is mainly caused by memory swapping, presenting significant pressure on the PCIe bus. Even under LIGHT workload, UM+MPS can only achieve 7% SLO compliance. The memory swapping also causes 2.2× degradation of training throughput.

Furthermore, we compare SIRIUS with the ideal case, i.e., **Infer-Only**, which is based on SIRIUS but solely executes inference tasks on GPU. The evaluation result shows that SIRIUS achieves averagely 95.3% (up to 98%) of inference SLO compliance of Infer-Only. The decrease in SLO compliance is mainly caused by additional cold start overhead (e.g., 9.1% cold start ratio under BURST workload).

Additionally, Figure 11 shows the CDF of inference latency with BURST workload, illustrating that SIRIUS improves overall inference latency. For more relaxed SLO targets, such as 10× of the standalone execution time, SIRIUS also improves SLO compliance by 48.8% on average compared with all competitors.

Finally, we evaluate the impact of dynamic batch size on training convergence. To evaluate the impact of randomness, we train Swin-T with and without dynamic batch size on CIFAR-100 [9] dataset for five times. We compare the number of epochs required to achieve the same accuracy of 75%. The evaluation result shows that the averaged number of epochs are 210.6 and 206.2 (the standard deviation are 1.5 and 3.7) for each case respectively. This confirms that SIRIUS can ensure training convergence.

6.2 Performance on Multiple GPUs

We next evaluate the performance of SIRIUS with four GPUs. To evaluate the inference and training colocation on multiple GPUs, we scale the request rate and the number of inference

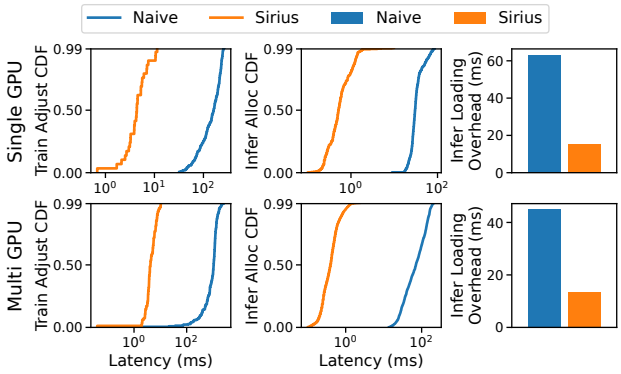


Figure 13: *The comparison of adjustment time for different approaches with* MAF *workload on one and four GPUs.*

models of the LIGHT workload by the number of GPUs. As shown in Figure 12, compared with all competitors, SIRIUS improves the inference P99 latency and SLO compliance by averagely $558.4\times$ and 43.0% respectively, and improves the training throughput by averagely $6.1\times$. The reason for the performance improvement of SIRIUS is similar to the single GPU case. Furthermore, SIRIUS achieves 89.0% of inference SLO compliance of Infer-Only.

6.3 Adjustment Breakdown

We next compare the memory handover performance of SIRIUS with the naive memory handover, which adjusts memory at the end of training batches. Figure 13 shows the breakdown of memory handover, which is evaluated over a mixture of 32 ResNet-152 with MAF workloads on both single and four GPUs. First, for training adjustment, the naive memory handover can take over 250 ms because it must wait for the completion of the training batch. Due to the synchronization across multiple GPUs, the naive adjustment has to wait for the effective batch to complete on all GPUs, resulting in latency exceeding 1 second. In contrast, SIRIUS can adjust training memory in less than 5 ms ($121\times$ faster than the naive on average) for both single and multiple GPUs, with P99 latencies of 11.4 ms and 10.5 ms respectively. Moreover, thanks to the coarse-grained reallocation, the average memory adjustment intervals are 10.3 seconds and 3.6 seconds for single and multiple GPUs, respectively. Second, by bypassing inefficient GPU runtime, SIRIUS accelerates memory allocation to averagely 0.8 ms (including zero-filling overhead), $89.2\times$ faster than the naive. Third, thanks to the SLO-aware reallocation and pipeline loading, SIRIUS reduces the waiting time for model loading by $3.7\times$ than the naive. Finally, compared with the naive adjustment, SIRIUS improves memory reallocation by $148\times$ and inference SLO compliance by 38% on average.

6.4 Ablation Study

We next evaluate the configuration of SIRIUS that affects the memory adjustment.

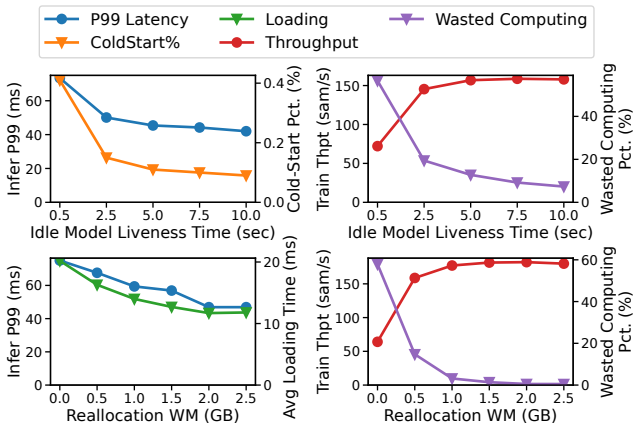


Figure 14: *The effect of maximum model liveness time and reallocation watermark on inference and training performance.*

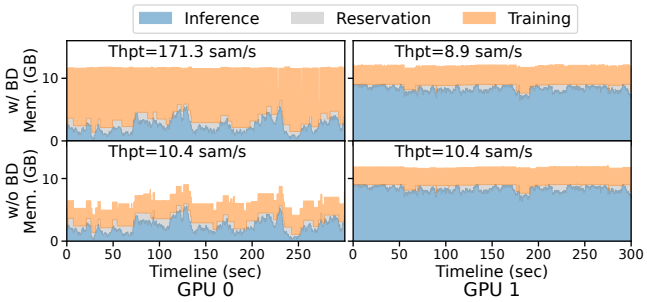


Figure 15: *The training throughput and GPU memory utilization of* SIRIUS *with an unbalanced workload.*

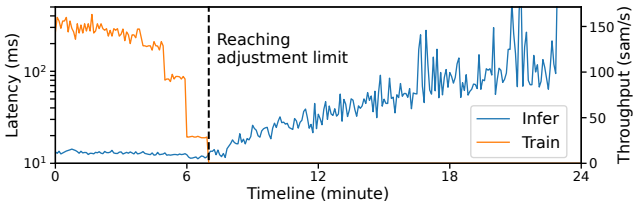


Figure 16: *The inference latency and training throughput of* SIRIUS *with varying the number of requested ResNet-152 models.*

Maximum liveness time ($\mathcal{T}_{idle}$). This configuration determines when idle inference models should be released. We evaluate the cold start ratio and P99 latency for inference tasks, as well as the wasted computation time spent on discarded batches and throughput for training tasks, using different $\mathcal{T}_{idle}$. In this experiment, we use the MAF workload and set the reallocation watermark to zero (i.e., no memory reservation for inference tasks). As shown at the top of Figure 14, with the increase of $\mathcal{T}_{idle}$, both the cold start ratio and P99 inference latency decrease. For training throughput, it first increases due to fewer wasted computing resources from discarded batches. However, for larger $\mathcal{T}_{idle}$, training throughput slightly decreases due to reduced memory resources. This suggests a sweet spot for aggressive levels of dynamic memory sharing.

Memory reallocation watermark ($\mathcal{W}$). The watermark impacts inference model loading and the frequency of adjustments. We evaluate the model loading time and P99 latency for inference tasks, as well as the wasted computation time and throughput for training tasks, using different memory reallocation watermarks. In this experiment, we use the MAF workload and set $\mathcal{T}_{idle}$ to 500 ms. As shown at the bottom of Figure 14, the finding is consistent with that of inference model liveness time. A larger watermark leads to a reduction in model loading time, P99 inference latency, and wasted training computation. However, training throughput may decrease if the watermark is too large.

6.5 Case Study

Memory imbalance. We evaluate the training throughput and GPU memory utilization of SIRIUS with an unbalanced workload on multiple GPUs. To construct the unbalanced workload, we colocate inference and training tasks on two GPUs, where one GPU serves LIGHT inference workload and the other serves HEAVY workload. To demonstrate the issue of memory imbalance, we set the adjustment limit of training batch size to one. Figure 15 shows memory utilization (excluding the CUDA context) without and with the batch distributing (BD) of SIRIUS. It can be found that without batch distributing, the training task must use the same batch size for both GPUs, resulting in memory underutilization on GPU_0 and suboptimal training throughput. In contrast, batch distributing allows the training task to use different configurations to fully utilize GPU memory and achieves 8.7× higher throughput.

Memory pressure. We next evaluate the performance of SIRIUS under memory pressure. We start with 4 ResNet-152 and gradually add 4 ResNet-152 every minute. There are 50 requests per second evenly distributed across these models. Figure 16 shows the evaluation result, where the inference latency and training throughput are averaged over 5 seconds. Before reaching the adjustment limit, inference tasks can get memory from the training task and maintain low latency while training throughput decreases. However, once all training adjustable memory is reallocated for inference tasks, SIRIUS adopts model swapping to serve more models, resulting in increasing inference latency.

6.6 LLM Inference

We further investigate the performance of SIRIUS with Large Language Models (LLMs). The memory consumption of LLMs fluctuates due to their KV caches, which typically dominate their memory requirements [41, 76]. Therefore, we dynamically allocate GPU memory for the LLM's KV cache and the training task to enable memory sharing. The LLM is pinned in GPU memory to minimize inference SLO violations. In our experiments, we use BurstGPT [85] trace and scale the maximum request rate to 10 req/s. We use Llama2-13B as the

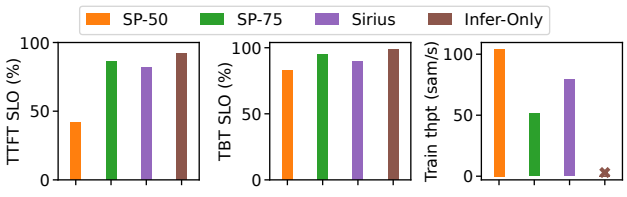


Figure 17: *The comparison of SLO compliance of TTFT (Time To First Token) and TBT (Time Between Tokens) of LLM serving, and training throughput.*

inference model and colocate it with training Qwen2-0.5B. We compare SIRIUS with **SP-50**, **SP-75**, and **Infer-Only**. All systems use vLLM [50] to serve inference requests. For SP-50 and SP-75, we set the KV cache size as 50% and 75% of that of Infer-Only, which configures the KV cache to utilize the GPU memory fully. For LLM inference, we evaluate the SLO compliance of TTFT (Time To First Token) and TBT (Time Between Tokens). Similarly, we set the SLO to be four times the average latency of an inference request standalone execution (see Table 3). Figure 17 shows the evaluation result on an NVIDIA A100 GPU with 80 GB memory. Compared with SP-50, SIRIUS improves the SLO compliance of TTFT and TBT by 40% and 7% respectively. Compared with SP-75, SIRIUS improves training throughput by 1.5×. This performance improvement stems from the same reason discussed in the previous evaluation—dynamic GPU memory sharing. Finally, in comparison to Infer-Only, SIRIUS achieves 89% and 91% of its SLO compliance for TTFT and TBT, respectively.

7 Discussion

Modern inference and training systems are scaled out across multiple GPUs and nodes to enhance their performance. To support memory handover for multi-GPU and multi-node inference tasks, Sirius could allow the inference engine on each GPU to notify the training task of the memory adjustment. The inference task will be processed once the memory adjustment is complete on all GPUs.

For multi-GPU and multi-node training tasks, the memory adjustment can still be completed immediately. However, such training tasks, which typically incorporate data parallelism, pipeline parallelism, and tensor parallelism [18, 74], must be reconfigured and resumed after the memory adjustment. Currently, SIRIUS has explored the elasticity of training tasks with data parallelism to adjust memory consumption, enabling fast GPU memory handover. To support model parallelism (i.e., pipeline parallelism and tensor parallelism), SIRIUS can reshard pipeline stages [30, 45, 81] and reshape tensors [86], respectively. We leave this to future work.

8 Related Work

Improving GPU cluster utilization. To address the underutilization of GPU clusters [27, 87], prior work [33, 69, 93] auto-scales GPU resources based on dynamic workloads. Differently, SIRIUS focuses on colocating ML tasks to improve GPU utilization. While Lyra [53] schedules inference and training tasks within the same GPU cluster, SIRIUS colocates two types of tasks on the same GPU in a fine-grained manner. Observing the dynamic resource utilization of training tasks, AntMan [89] co-designs the GPU cluster scheduler and the ML framework to colocate training tasks, which reconciles the GPU memory resources between training tasks by unifying GPU and host memory. Compared with AntMan, SIRIUS instantly adjusts the memory consumption of training tasks to meet inference SLOs.

Elastic training and dynamic batch. To utilize dynamic GPU resources, elastic training has gained increasing attention recently [2, 3, 6, 46, 54, 61, 66, 82, 84]. Different from these approaches, which mainly focus on improving utilization and accelerating training tasks, SIRIUS exploits the elasticity of training tasks to colocate with latency-sensitive inference tasks under dynamic inference workloads.

GPU sharing for machine learning. Fully utilizing GPU becomes increasingly challenging for more powerful GPUs [87, 88, 92, 97]. NVIDIA GPU offers several techniques to enable GPU sharing, including MPS [4] and MIG [5]. MIG is a hardware mechanism for static partitioning, dividing the GPU into multiple isolated instances. While prior work [22, 24, 25, 36, 52, 72, 75] has primarily focused on computational resource sharing, SIRIUS targets GPU memory sharing. Different from PipeSwitch [16] and Zico [57], SIRIUS dynamically allocates GPU memory between ML tasks to achieve efficient colocation.

9 Conclusion

In this paper, we present SIRIUS, a colocation system that leverages training elasticity to dynamically share GPU memory between ML inference and training tasks. SIRIUS prioritizes inference tasks to meet strict latency SLOs, allowing them access GPU memory without interference. Meanwhile, it concurrently runs training tasks on remaining resources to maximize throughput and GPU utilization. The key innovation behind SIRIUS is a millisecond-scale GPU memory handover mechanism between training and inference tasks. Our experiments demonstrate the efficacy and efficiency of SIRIUS in colocating inference and training tasks. SIRIUS is open-source and publicly available at https://github.com/SiriusInfTra/Sirius.

Acknowledgments

We sincerely thank our anonymous shepherd and reviewers for their insightful comments and suggestions. This work was supported in part by the National Natural Science Foundation of China (No. 62272291, 62432010), the Fundamental Research Funds for the Central Universities, and a research grant from Huawei Technologies. Corresponding author: Rong Chen (rongchen@sjtu.edu.cn).

References

[1] 2023. CUDA C++ Programming Guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

[2] 2023. Elastic Horovod. https://horovod.readthedocs.io/en/latest/elastic_include.html.

[3] 2023. ElasticDL: A Kubernetes-native Deep Learning Framework. https://github.com/sql-machine-learning/elasticdl.

[4] 2023. Multi-Process Service. https://docs.nvidia.com/deploy/mps/index.html.

[5] 2023. NVIDIA Multi-Instance GPU User Guide. https://docs.nvidia.com/datacenter/tesla/mig-user-guide/index.html.

[6] 2023. Torch Distributed Elastic. https://pytorch.org/docs/stable/distributed.elastic.html.

[7] 2024. Amazon SageMaker. https://aws.amazon.com/sagemaker.

[8] 2024. Azure Machine Learning. https://azure.microsoft.com/en-us/products/machine-learning.

[9] 2024. CIFAR-10 and CIFAR-100 datasets. https://www.cs.toronto.edu/~kriz/cifar.html.

[10] 2024. Hugging Face Model Hub. https://huggingface.co/models.

[11] 2024. NVIDIA Collective Communications Library. https://developer.nvidia.com/nccl.

[12] 2024. NVIDIA Triton Inference Server. https://www.nvidia.com/en-us/ai-data-science/products/triton-inference-server/.

[13] 2024. Qwen2 Technical Report. (2024).

[14] 2025. Optimize GKE resource utilization for mixed AI/ML training and inference workloads. https://cloud.google.com/kubernetes-engine/docs/tutorials/mixed-workloads.

[15] 2025. Tencent Cloud qGPU. https://www.tencentcloud.com/document/product/457/52078?lang=en.

[16] Zhihao Bai, Zhen Zhang, Yibo Zhu, and Xin Jin. 2020. PipeSwitch: Fast pipelined context switching for deep learning applications. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 499–514.

[17] Joshua Bakita and James H Anderson. 2023. Hardware compute partitioning on NVIDIA GPUs. In *2023 IEEE 29th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 54–66.

[18] Zhengda Bian, Qifan Xu, Boxiang Wang, and Yang You. 2021. Maximizing parallelism in distributed training for huge neural networks. *arXiv preprint arXiv:2105.14450* (2021).

[19] Han Cai, Junyan Li, Muyan Hu, Chuang Gan, and Song Han. 2022. EfficientViT: Multi-scale linear attention for high-resolution dense prediction. *arXiv preprint arXiv:2205.14756* (2022).

[20] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. 2018. TVM: An automated end-to-end optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 578–594.

[21] Runxiang Cheng, Chris Cai, Selman Yilmaz, Rahul Mitra, Malay Bag, Mrinmoy Ghosh, and Tianyin Xu. 2023. Towards GPU Memory Efficiency for Distributed Training at Scale. In *Proceedings of the 2023 ACM Symposium on Cloud Computing*. 281–297.

[22] Marcus Chow, Ali Jahanshahi, and Daniel Wong. 2023. KRISP: Enabling Kernel-wise RIght-sizing for Spatial Partitioned GPU Inference Servers. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA 23)*. IEEE, 624–637.

[23] Weihao Cui, Han Zhao, Quan Chen, Hao Wei, Zirui Li, Deze Zeng, Chao Li, and Minyi Guo. 2022. DVABatch: Diversity-aware Multi-Entry Multi-Exit Batching for Efficient Processing of DNN Services on GPUs. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. 183–198.

[24] Aditya Dhakal, Sameer G Kulkarni, and KK Ramakrishnan. 2020. GSlice: controlled spatial sharing of GPUs for a scalable inference platform. In *Proceedings of the 11th ACM Symposium on Cloud Computing*. 492–506.

[25] Aditya Dhakal, Sameer G. Kulkarni, and K. K. Ramakrishnan. 2024. D-STACK: High Throughput DNN Inference by Effective Multiplexing and Spatio-Temporal Scheduling of GPUs. *IEEE Transactions on Cloud Computing* 12, 4 (2024), 1344–1358.

[26] Yao Fu, Leyang Xue, Yeqi Huang, Andrei-Octavian Brabete, Dmitrii Ustiugov, Yuvraj Patel, and Luo Mai. 2024. ServerlessLLM: Low-latency Serverless Inference for Large Language Models. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. 135–153.

[27] Wei Gao, Qinghao Hu, Zhisheng Ye, Peng Sun, Xiaolin Wang, Yingwei Luo, Tianwei Zhang, and Yonggang Wen. 2022. Deep learning workload scheduling in GPU datacenters: Taxonomy, challenges and vision. *arXiv preprint arXiv:2205.11913* (2022).

[28] Yanjie Gao, Yu Liu, Hongyu Zhang, Zhengxian Li, Yonghao Zhu, Haoxiang Lin, and Mao Yang. 2020. Estimating GPU memory consumption of deep learning models. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1342–1352.

[29] Chirayu Garg and Nikolay Sakharnykh. 2021. Improving GPU Memory Oversubscription Performance. https://developer.nvidia.com/blog/improving-gpu-memory-oversubscription-performance/.

[30] Hao Ge, Fangcheng Fu, Haoyang Li, Xuanyu Wang, Sheng Lin, Yujie Wang, Xiaonan Nie, Hailin Zhang, Xupeng Miao, and Bin Cui. 2024. Enabling Parallelism Hot Switching for Efficient Training of Large Language Models. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*. 178–194.

[31] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. 2017. Accurate, large minibatch sgd: Training imagenet in 1 hour. *arXiv preprint arXiv:1706.02677* (2017).

[32] Jianfeng Gu, Yichao Zhu, Puxuan Wang, Mohak Chadha, and Michael Gerndt. 2023. FaST-GShare: Enabling efficient spatio-temporal GPU sharing in serverless computing for deep learning inference. In *Proceedings of the 52nd International Conference on Parallel Processing*. 635–644.

[33] Arpan Gujarati, Sameh Elnikety, Yuxiong He, Kathryn S McKinley, and Björn B Brandenburg. 2017. Swayam: distributed autoscaling to meet slas of machine learning inference services with resource efficiency. In *Proceedings of the 18th ACM/IFIP/USENIX middleware conference*. 109–120.

[34] Arpan Gujarati, Reza Karimi, Safya Alzayat, Wei Hao, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace. 2020. Serving DNNs like clockwork: Performance predictability from the bottom up. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 443–462.

[35] Cong Guo, Rui Zhang, Jiale Xu, Jingwen Leng, Zihan Liu, Ziyu Huang, Minyi Guo, Hao Wu, Shouren Zhao, Junping Zhao, and Ke Zhang. 2024. GMLake: Efficient and Transparent GPU Memory Defragmentation for Large-scale DNN Training with Virtual Memory Stitching. https://api.semanticscholar.org/CorpusID:267027901

[36] Mingcong Han, Hanze Zhang, Rong Chen, and Haibo Chen. 2022. Microsecond-scale preemption for concurrent GPU-accelerated DNN inferences. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 539–558.

[37] Ziyi Han, Ruiting Zhou, Chengzhong Xu, Yifan Zeng, and Renli Zhang. 2024. INSS: An intelligent scheduling orchestrator for multi-GPU inference with spatio-temporal sharing. *IEEE Transactions on Parallel and Distributed Systems* (2024).

[38] Mor Harchol-Balter. 2013. *Performance modeling and design of computer systems: queueing theory in action*. Cambridge University Press.

[39] Mark Harris. 2017. Unified Memory for CUDA Beginners. https://developer.nvidia.com/blog/unified-memory-cuda-beginners/.

[40] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.

[41] Coleman Hooper, Sehoon Kim, Hiva Mohammadzadeh, Michael W Mahoney, Yakun Sophia Shao, Kurt Keutzer, and Amir Gholami. 2024. Kvquant: Towards 10 million context length LLM inference with KV cache quantization. *arXiv preprint arXiv:2401.18079* (2024).

[42] Qinghao Hu, Peng Sun, Shengen Yan, Yonggang Wen, and Tianwei Zhang. 2021. Characterization and prediction of deep learning workloads in large-scale GPU datacenters. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–15.

[43] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. 2017. Densely connected convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 4700–4708.

[44] Paras Jain, Xiangxi Mo, Ajay Jain, Harikaran Subbaraj, Rehan Sohail Durrani, Alexey Tumanov, Joseph Gonzalez, and Ion Stoica. 2018. Dynamic space-time scheduling for GPU inference. *arXiv preprint arXiv:1901.00041* (2018), 1–8.

[45] Insu Jang, Zhenning Yang, Zhen Zhang, Xin Jin, and Mosharaf Chowdhury. 2023. Oobleck: Resilient distributed training of large models using pipeline templates. In *Proceedings of the 29th Symposium on Operating Systems Principles*. 382–395.

[46] Suhas Jayaram Subramanya, Daiyaan Arfeen, Shouxu Lin, Aurick Qiao, Zhihao Jia, and Gregory R Ganger. 2023. Sia: Heterogeneity-aware, goodput-optimized ML-cluster scheduling. In *Proceedings of the 29th Symposium on Operating Systems Principles*. 642–657.

[47] Myeongjae Jeon, Shivaram Venkataraman, Amar Phanishayee, Junjie Qian, Wencong Xiao, and Fan Yang. 2019. Analysis of Large-Scale Multi-Tenant GPU clusters for DNN training workloads. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. 947–960.

[48] Jinwoo Jeong, Seungsu Baek, and Jeongseob Ahn. 2023. Fast and Efficient Model Serving Using Multi-GPUs with Direct-Host-Access. In *Proceedings of the Eighteenth European Conference on Computer Systems*. 249–265.

[49] Yunho Jin, Chun-Feng Wu, David Brooks, and Gu-Yeon Wei. 2023. S3: Increasing GPU Utilization during Generative Inference for Higher Throughput. *Advances in Neural Information Processing Systems* 36 (2023), 18015–18027.

[50] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*. 611–626.

[51] Wonbeom Lee, Jungi Lee, Junghwan Seo, and Jaewoong Sim. 2024. InfiniGen: Efficient generative inference of large language models with dynamic KV cache management. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. 155–172.

[52] Baolin Li, Tirthak Patel, Siddharth Samsi, Vijay Gadepally, and Devesh Tiwari. 2022. MISO: exploiting multi-instance GPU capability on multi-tenant GPU clusters. In *Proceedings of the 13th Symposium on Cloud Computing*. 173–189.

[53] Jiamin Li, Hong Xu, Yibo Zhu, Zherui Liu, Chuanxiong Guo, and Cong Wang. 2023. Lyra: Elastic scheduling for deep learning clusters. In *Proceedings of the Eighteenth European Conference on Computer Systems*. 835–850.

[54] Mingzhen Li, Wencong Xiao, Hailong Yang, Biao Sun, Hanyu Zhao, Shiru Ren, Zhongzhi Luan, Xianyan Jia, Yi Liu, Yong Li, et al. 2023. EasyScale: Elastic Training with Consistent Accuracy and Improved Utilization on GPUs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–14.

[55] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, et al. 2020. PyTorch distributed: Experiences on accelerating data parallel training. *arXiv preprint arXiv:2006.15704* (2020).

[56] Zhuohan Li, Lianmin Zheng, Yinmin Zhong, Vincent Liu, Ying Sheng, Xin Jin, Yanping Huang, Zhifeng Chen, Hao Zhang, Joseph E Gonzalez, et al. 2023. AlpaServe: Statistical Multiplexing with Model Parallelism for Deep Learning Serving. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. 663–679.

[57] Gangmuk Lim, Jeongseob Ahn, Wencong Xiao, Youngjin Kwon, and Myeongjae Jeon. 2021. Zico: Efficient GPU memory sharing for concurrent DNN training. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. 161–175.

[58] Haibin Lin, Hang Zhang, Yifei Ma, Tong He, Zhi Zhang, Sheng Zha, and Mu Li. 2019. Dynamic mini-batch sgd for elastic distributed training: Learning in the limbo of resources. *arXiv preprint arXiv:1904.12043* (2019).

[59] Ze Liu, Yutong Lin, Yue Cao, Han Hu, Yixuan Wei, Zheng Zhang, Stephen Lin, and Baining Guo. 2021. Swin transformer: Hierarchical vision transformer using shifted windows. In *Proceedings of the IEEE/CVF international conference on computer vision*. 10012–10022.

[60] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. 2015. Heracles: Improving resource efficiency at scale. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*. 450–462.

[61] Luo Mai, Guo Li, Marcel Wagenländer, Konstantinos Fertakis, Andrei-Octavian Brabete, and Peter Pietzuch. 2020. KungFu: Making training in distributed machine learning adaptive. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 937–954.

[62] Xupeng Miao, Gabriele Oliaro, Xinhao Cheng, Mengdi Wu, Colin Unger, and Zhihao Jia. 2024. FlexLLM: A System for Co-Serving Large Language Model Inference and Parameter-Efficient Finetuning. *arXiv preprint arXiv:2402.18789* (2024).

[63] Jaiaid Mobin, Avinash Maurya, and M Mustafa Rafique. 2023. COLTI: Towards Concurrent and Co-located DNN Training and Inference. In *Proceedings of the 32nd International Symposium on High-Performance Parallel and Distributed Computing*. 309–310.

[64] Ismael Solis Moreno, Peter Garraghan, Paul Townend, and Jie Xu. 2014. Analysis, modeling and simulation of workload patterns in a large-scale utility cloud. *IEEE Transactions on Cloud Computing* 2, 2 (2014), 208–221.

[65] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems* 32 (2019).

[66] Aurick Qiao, Sang Keun Choe, Suhas Jayaram Subramanya, Willie Neiswanger, Qirong Ho, Hao Zhang, Gregory R Ganger, and Eric P Xing. 2021. Pollux: Co-adaptive cluster scheduling for goodput-optimized deep learning. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*.

[67] Ruoyu Qin, Zheming Li, Weiran He, Mingxing Zhang, Yongwei Wu, Weimin Zheng, and Xinran Xu. [n. d.]. Mooncake: A kvcache-centric disaggregated architecture for LLM serving. *URL https://arxiv. org/abs/2407.00079* ([n. d.]).

[68] Vijay Janapa Reddi, Christine Cheng, David Kanter, Peter Mattson, Guenther Schmuelling, Carole-Jean Wu, Brian Anderson, Maximilien Breughe, Mark Charlebois, William Chou, et al. 2020. Mlperf inference benchmark. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 446–459.

[69] Francisco Romero, Qian Li, Neeraja J Yadwadkar, and Christos Kozyrakis. 2021. INFaaS: Automated model-less inference serving. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. 397–411.

[70] Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. 2019. DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter. *arXiv preprint arXiv:1910.01108* (2019).

[71] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. 2020. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *2020 USENIX annual technical conference (USENIX ATC 20)*. 205–218.

[72] Weihang Shen, Mingcong Han, Jialong Liu, Rong Chen, and Haibo Chen. 2025. XSched: Preemptive Scheduling for Diverse XPUs. In *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*.

[73] Sudipta Saha Shubha, Haiying Shen, and Anand Iyer. 2024. USHER: Holistic Interference Avoidance for Resource Optimized ML Inference. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. 947–964.

[74] Jaeyong Song, Jinkyu Yim, Jaewon Jung, Hongsun Jang, Hyung-Jin Kim, Youngsok Kim, and Jinho Lee. 2023. Optimus-CC: Efficient large NLP model training with 3D parallelism aware communication compression. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 560–573.

[75] Foteini Strati, Xianzhe Ma, and Ana Klimovic. 2024. Orion: Interference-aware, Fine-grained GPU Sharing for ML Applications. In *Proceedings of the Eighteenth European Conference on Computer Systems*. 1075–1092.

[76] Biao Sun, Ziming Huang, Hanyu Zhao, Wencong Xiao, Xinyi Zhang, Yong Li, and Wei Lin. 2024. Llumnix: Dynamic Scheduling for Large Language Model Serving. *arXiv preprint arXiv:2406.03243* (2024).

[77] Mingxing Tan and Quoc Le. 2021. Efficientnetv2: Smaller models and faster training. In *International conference on machine learning*. PMLR, 10096–10106.

[78] The Mosaic ML Team. 2021. composer. https://github.com/mosaicml/composer/.

[79] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. 2023. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288* (2023).

[80] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. 2015. Large-scale cluster management at Google with Borg. In *Proceedings of the tenth european conference on computer systems*. 1–17.

[81] Marcel Wagenländer, Guo Li, Bo Zhao, Luo Mai, and Peter Pietzuch. 2024. Tenplex: Dynamic Parallelism for Deep Learning using Parallelizable Tensor Collections. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*. 195–210.

[82] Marcel Wagenländer, Luo Mai, Guo Li, and Peter Pietzuch. 2020. Spotnik: Designing distributed machine learning for transient cloud resources. In *12th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 20)*.

[83] Linnan Wang, Jinmian Ye, Yiyang Zhao, Wei Wu, Ang Li, Shuaiwen Leon Song, Zenglin Xu, and Tim Kraska. 2018. Superneurons: Dynamic GPU memory management for training deep neural networks. In *Proceedings of the 23rd ACM SIGPLAN symposium on principles and practice of parallel programming*. 41–53.

[84] Qinlong Wang, Bo Sang, Haitao Zhang, Mingjie Tang, and Ke Zhang. 2023. DLRover: An Elastic Deep Training Extension with Auto Job Resource Recommendation. *arXiv preprint arXiv:2304.01468* (2023).

[85] Yuxin Wang, Yuhan Chen, Zeyu Li, Xueze Kang, Zhenheng Tang, Xin He, Rui Guo, Xin Wang, Qiang Wang, Amelie Chi Zhou, et al. 2024. BurstGPT: A Real-world Workload Dataset to Optimize LLM Serving Systems. (2024).

[86] Zhigang Wang, Xu Zhang, Ning Wang, Chuanfei Xu, Jie Nie, Zhiqiang Wei, Yu Gu, and Ge Yu. 2024. Accelerating Heterogeneous Tensor Parallelism via Flexible Workload Control. *arXiv preprint arXiv:2401.11469* (2024).

[87] Qizhen Weng, Wencong Xiao, Yinghao Yu, Wei Wang, Cheng Wang, Jian He, Yong Li, Liping Zhang, Wei Lin, and Yu Ding. 2022. MLaaS in the wild: Workload analysis and scheduling in large-scale heterogeneous GPU clusters. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. 945–960.

[88] Bingyang Wu, Zili Zhang, Zhihao Bai, Xuanzhe Liu, and Xin Jin. 2023. Transparent GPU Sharing in Container Clouds for Deep Learning Workloads. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. 69–85.

[89] Wencong Xiao, Shiru Ren, Yong Li, Yang Zhang, Pengyang Hou, Zhi Li, Yihui Feng, Wei Lin, and Yangqing Jia. 2020. AntMan: Dynamic scaling on GPU clusters for deep learning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 533–548.

[90] Jianwei Yin, Xingjian Lu, Xinkui Zhao, Hanwei Chen, and Xue Liu. 2014. BURSE: A bursty and self-similar workload generator for cloud computing. *IEEE Transactions on Parallel and Distributed Systems* 26, 3 (2014), 668–680.

[91] Minchen Yu, Ao Wang, Dong Chen, Haoxuan Yu, Xiaonan Luo, Zhuohao Li, Wei Wang, Ruichuan Chen, Dapeng Nie, and Haoran Yang. 2023. FaaSwap: SLO-Aware, GPU-Efficient Serverless Inference via Model Swapping. *arXiv preprint arXiv:2306.03622* (2023).

[92] Peifeng Yu and Mosharaf Chowdhury. 2019. Salus: Fine-grained GPU sharing primitives for deep learning applications. *arXiv preprint arXiv:1902.04610* (2019).

[93] Chengliang Zhang, Minchen Yu, Wei Wang, and Feng Yan. 2019. MArk: Exploiting Cloud Services for Cost-Effective, SLO-Aware Machine Learning Inference Serving. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. 1049–1062.

[94] Hong Zhang, Yupeng Tang, Anurag Khandelwal, and Ion Stoica. 2023. SHEPHERD: Serving DNNs in the Wild. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. 787–808.

[95] Jeff Zhang, Sameh Elnikety, Shuayb Zarar, Atul Gupta, and Siddharth Garg. 2020. Model-Switching: Dealing with Fluctuating Workloads in Machine-Learning-as-a-Service Systems. In *12th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 20)*.

[96] Junzhe Zhang, Sai Ho Yeung, Yao Shu, Bingsheng He, and Wei Wang. 2019. Efficient memory management for GPU-based deep learning systems. *arXiv preprint arXiv:1903.06631* (2019).

[97] Yihao Zhao, Xin Liu, Shufan Liu, Xiang Li, Yibo Zhu, Gang Huang, Xuanzhe Liu, and Xin Jin. 2023. MuxFlow: Efficient and Safe GPU Sharing in Large-Scale Production Deep Learning Clusters. *arXiv preprint arXiv:2303.13803* (2023).

[98] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, et al. 2020. Ansor: Generating high-performance tensor programs for deep learning. In *14th USENIX symposium on operating systems design and implementation (OSDI 20)*. 863–879.

A Artifact Appendix

This artifact provides the source code of SIRIUS, a detailed readme, and scripts to reproduce the main experimental results from the ATC 2025 paper—"Colocating ML Inference and Training with Fast GPU Memory Handover" by J. Wang, Y. Wang, M. Han, and R. Chen. SIRIUS is a colocation system that leverages training elasticity to dynamically share GPU memory between ML inference and training tasks. We provide instructions to build the software package and run experiments. Our artifact obtained the "Artifacts Available," "Artifacts Functional," and "Results Reproduced" badges from the Artifact Evaluation process of ATC 2025. The DOI of our artifact is https://doi.org/10.5281/zenodo.15581800.

Artifact repository. All project source code, along with comprehensive instructions for building and running the main experiments on SIRIUS, is available in the following git repository: https://github.com/SiriusInfTra/Sirius.git.

AssyLLM: Efficient Federated Fine-tuning of LLMs via Assembling Pre-trained Blocks

Shichen Zhan, Li Li*, Chengzhong Xu
State Key Laboratory of Internet of Things for Smart City, University of Macau

Abstract

Federated Learning (FL) provides a promising way to fine-tune Large Language Models (LLMs) to downstream mobile tasks while preserving data privacy. However, the intensive memory footprint prevents large amount of edge devices from contributing to the fine-tuning process with their own private data.

To this end, we introduce AssyLLM, an innovative framework that conducts fine-tuning in a memory-efficient manner through directly assembling the pre-trained transformer blocks. The core idea of AssyLLM is to decompose a pre-trained LLM into discrete blocks. These blocks are iteratively selected based on the local corpus distributed across various devices, and subsequently assembled to form a novel LLM tailored for downstream tasks. In this way, high fine-tuning efficiency can be achieved through avoiding the backpropagation process adopted in traditional fine-tuning approaches. Specifically, AssyLLM features four core components: 1) Block Comparator, 2) Elastic Adapter, 3) Block Quanter, and 4) Block Swapper. Block Comparator is designed to assess the compatibility between two blocks, facilitating the selection of appropriate blocks for assembling. After that, Elastic Adapter creates customized adapter configurations that address the specific structural differences between the blocks for seamless concatenation between the selected blocks. Meanwhile, Block Quanter is proposed to adjust precision of related weights based on the block output activation in order to reduce the extra memory overhead caused by retaining the candidate blocks while preserving the performance of the assembled model. Moreover, in order to further increase the scalability of the candidate blocks for better fine-tuning performance while guaranteeing fine-tuning progress, Block Swapper is designed to optimize the swapping pipeline by incorporating block correlation metrics. AssyLLM is comprehensively evaluated on multiple benchmark datasets of varying complexity. Compared to traditional methods, AssyLLM improves accuracy by up to 18.26%, achieves up to 30.04× speedup, and significantly reduces memory consumption by up to 92%.

*Corresponding author: Li Li (llili@um.edu.mo)

1 Introduction

Recently, Large Language Models (LLMs) have demonstrated remarkable proficiency in language understanding and generation and are promising to widely support different mobile applications [17, 32]. Fine-tuning pre-trained LLMs using specialized domain-specific corpus has been proven to effectively enhance their applicability to downstream tasks [43]. However, those data are often located across various edge devices, leading to privacy concerns during the collection process. Federated Learning (FL) provides a promising way to fine-tune the LLM while guaranteeing data privacy, e.g. FedLLM [4, 5, 33, 40, 44]. Instead of collecting the data to the central server, FedLLM coordinates the participants to fine-tune the LLM locally on their private data and share only the updated model parameters with the server for aggregation.

Despite its benefits, the high memory footprint of local fine-tuning prevents resource-limited devices from contributing private data in real-world scenarios. For example, full fine-tuning a pre-trained Llama-7B model [39] with a batch size of 16 requires over 40GB of memory, whereas edge devices typically have memory capacities ranging from 4 to 16GB [1]. Even with mainstream parameter-efficient fine-tuning (PEFT) methods like LoRA [14] and adapter-based approaches [28], fine-tuning Llama-7B still requires over 15GB of memory, which still cannot be afforded by most edge devices. The significant disparity between the memory demands of model fine-tuning and the limited memory capacities of edge devices results in a substantial number of low-end devices being excluded from the fine-tuning process. Consequently, these devices are unable to contribute their locally stored data, diminishing the diversity and richness of the fine-tuning dataset. This, in turn, negatively impacts the performance of the LLM.

Limitation of Prior Arts. In order to reduce the memory footprint during the fine-tuning process, several approaches have been proposed which can be mainly divided into the following categories. The first category of work applies quantization to PEFT [8, 14] or adopts adapter tuning with layer freezing [4] to further reduce the resource consumption of

PEFT. Although these methods reduce model size and memory usage, incomplete model updating often degrades performance, and higher compression results in greater accuracy loss. The second category employs zeroth-order optimization [29], or forward gradient [42] to reduce the memory footprint caused by backpropagation (BP-free). However, these methods can lead to unstable model performance due to reliance solely on forward gradient estimation. The third category tackles the memory issue from the system perspective. For instance, gradient recomputation [6, 16, 19] and swapping [15, 30, 31, 38] have been introduced to manage memory usage by dynamically trading off computation and I/O overhead. While both recomputation and swapping effectively reduce memory consumption, they incur higher training time and computational overhead, which ultimately diminishes overall training efficiency. Hence, a new fine-tuning paradigm is urgently required to surmount the memory constraints while guaranteeing the fine-tuning performance and efficiency for the deployment of FedLLM in real-world cases.

Our Design. To tackle these challenges, this paper proposes AssyLLM, an innovative framework that addresses memory constraints in federated fine-tuning by leveraging a pool of pre-trained LLMs instead of relying on a single model. The main idea of AssyLLM is to break down multiple pre-trained LLMs into smaller blocks and select the most appropriate block segments for the downstream tasks. By systematically combining these blocks from various pre-trained LLMs, we create a custom-assembled model that draws on the unique strengths of each block. Specifically, it coordinates the participating clients to select optimal blocks using inference operations, with the help of block compatibility analysis. On the server side, these selected blocks are aggregated to form a global LLM. By dynamically selecting and assembling the most relevant blocks for each task, AssyLLM eliminates the need of repetitive finetuning, thereby accelerating the adaptation of LLMs to diverse applications. Moreover, through avoiding the BP process, this block-wise strategy significantly lowers memory demands during fine-tuning, enabling more devices to participate in the fine-tuning process.

Challenges and Techniques. However, designing such a new learning paradigm is non-trivial and presents several key challenges.

1) How to compare the compatibility between blocks? For effective assembling, it is crucial to select a compatible counterpart from the block pool. A higher compatibility between the selected blocks leads to improved performance of the resulting assembled model. We propose Block Comparator to evaluate the compatibility between blocks using meticulously designed metrics, enabling the selection of the most suitable blocks for concatenation and thereby improving the performance of the assembled model.

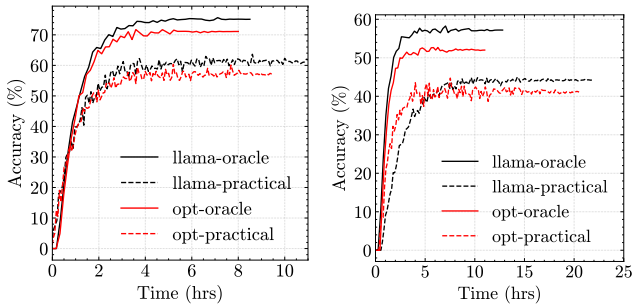
2) How to ensure the appropriate assembly of two blocks while guaranteeing the model performance and fine-tuning efficiency? Blocks from different pre-trained models vary in structure, features, and dimensions, rendering simple concatenation ineffective and potentially harmful. To address this challenge, we introduce Elastic Adapter, an adaptive lightweight fine-tuning approach with adapter that resolves mismatches by aligning the inputs and outputs of blocks from heterogeneous models, ensuring cohesive functionality in the assembled model.

3) How to reduce the excessive memory footprint of the block pool? Compared to a single LLM, constructing a block pool from multiple LLMs incurs substantial memory overhead, adding additional pressure on devices. Hence, we introduce Block Quanter, a mixed-precision quantization method applied at the block level. Since block selection depends on the output activation from each block's inference pass (see Section 3.1), our method applies varying precision levels based on impact of weight inside the block on the output. We prioritize higher precision for critical weights and lower precision for less impactful ones by analyzing the correlation between weights and output activations.

Although Block Quanter effectively reduces the block pool size, it remains too large for deployment on edge devices when handling billion-scale LLMs (e.g., the Llama series). Additionally, incorporating more pre-trained LLMs into a larger block pool improves adaptability and performance across downstream tasks, but also increases related memory costs. Further compression could degrade model performance, making it insufficient for edge deployment. To address these challenges, we introduce Block Swapper, a block-based swapping method that reduces memory usage by dynamically swapping blocks between internal and external storage. Traditional swapping methods face significant I/O latency. In contrast, Block Swapper utilizes a block correlation-aware pipeline strategy, along with pre-loading and pre-swapping mechanisms, which substantially reduce latency during the swapping process.

To the best of our knowledge, we are the first to address fine-tuning memory optimization in FL by leveraging multiple pre-trained LLMs and inference passes. Our approach significantly reduces BP-related overhead, lowering memory cost, energy consumption, and data requirements. This allows resource-constrained clients to participate while achieving performance comparable to state-of-the-art methods. The main contributions are summarized as follows:

- We propose AssyLLM, an FL framework that employs a new LLM generation method, splitting pre-trained models into blocks and assembling them together for downstream tasks.
- We design and implement the four core components, Block Comparator, Elastic Adapter, Block Quanter, and Block Swapper to interact with each other and guide the whole block selection process.
- Extensive experiments evaluate the effectiveness of As-



(a) BoolQ. (b) OBQA.

Figure 1: The memory wall in FedLLM; llama denotes Llama-7B, opt denotes OPT-6.7B.

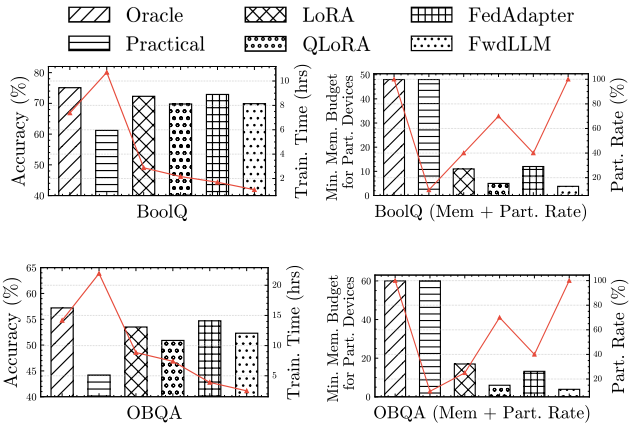
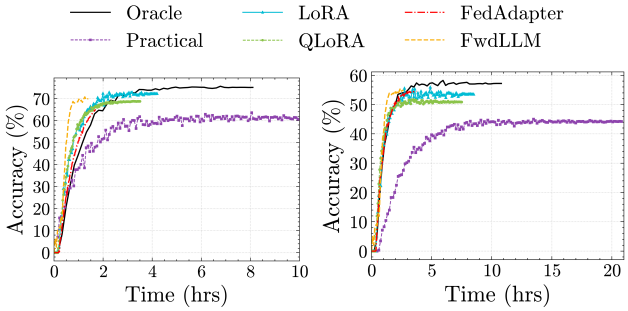


Figure 2: Performance comparison with existing works (left) accuracy and convergence time; (right) minimum memory budget for participation devices and participation rate on BoolQ and OBQA. Model: Llama-7B.



(a) BoolQ. (b) OBQA.

Figure 3: Performance comparison for existing works. Model: Llama-7B.

syLLM on accuracy improvement, generation speedup, and cost reduction for energy and memory. The code of AssyLLM is available at: https://github.com/zhanshichen/AssyLLM.

2 Background and Motivation

2.1 The Memory Wall of FedLLM

One important question required to be explored is: *how does the memory wall impact the federated fine-tuning process of LLM?* In order to make the investigation, we conduct the following experiments. To emulate the federated fine-tuning environment, we set 200 clients and split them into five groups with different memory budgets, as 10% (64GB), 15% (32GB), 15% (16GB), 30% (8GB), and 30% (4GB). We perform fine-tuning for Llama-7B [35] and OPT-6.7B on BoolQ [7] and OpenBookQA (OBQA) [9] datasets. Specifically, BoolQ is a binary question-answering dataset and OBQA focuses on multiple-choice reasoning questions. For data distribution, we follow [41], using a Dirichlet distribution [12] with concentration parameter $\alpha = 1$ to allocate fine-tuning samples for each client. The *oracle* baseline assumes all clients have sufficient memory for full local fine-tuning, while *practical* reflects experiments under memory constraints. As shown in Figure 1, memory limitations cause accuracy drops of up to 14.7% for Llama-7B and 19.1% for OPT-6.7B, as their fine-tuning requires over 45GB and 32GB of memory, respectively. This leaves 85% of Llama-7B clients and 60% of OPT-6.7B clients unable to participate, reducing data diversity and resulting in underperformed fine-tuning results.

2.2 Exploring Existing Techniques in FedLLM

In this section, we explore several techniques that are commonly used in LLM fine-tuning for memory saving.

2.2.1 Parameter-efficient Fine-Tuning.

Parameter-efficient fine-tuning (PEFT) techniques like LoRA [8, 14] and Adapters [4] enable task-specific adaptation of LLMs with minimal retraining. LoRA injects trainable low-rank matrices into attention and feed-forward layers, reducing the number of updated parameters. Adapters introduce task-specific modules between transformer layers, keeping the main model frozen while fine-tuning only these adapter layers.

Using the same experimental settings, we evaluate LoRA, QLoRA [8] (extending LoRA with quantization), FedAdapter [4], and FwdLLM [42] (will be introduced later) in an FL scenario with Llama-7B. As shown in Figures 2 and 3, for example, QLoRA reduces at most 79% memory consumption, decreases the aggregation time by 67%, and increases the participation rate to 70%, but it still comes with a 5.3% accuracy drop compared to oracle scenario. Similarly, other baselines also experience varying degrees of performance drop. This decline is due to two key factors. Firstly, incomplete model updates often lead to performance degradation, with higher compression levels causing greater accuracy loss. Secondly, while PEFT reduces memory usage compared to full fine-tuning, it still requires storing and updating low-rank matrices or adapters, and a portion of resource-constraints clients still remain unable to participate in the fine-tuning

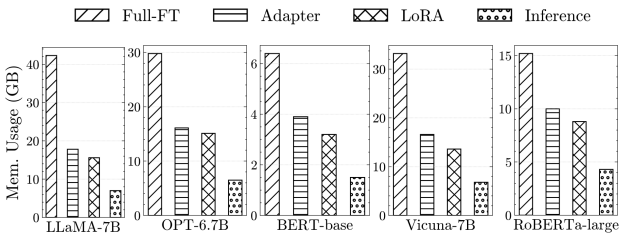


Figure 4: Memory overhead of different fine-tuning technologies and inference.

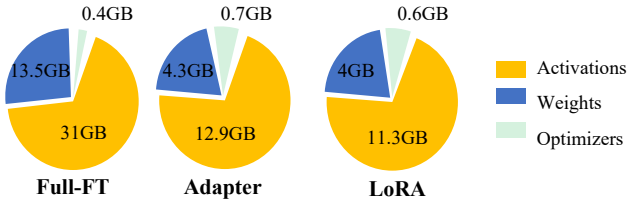


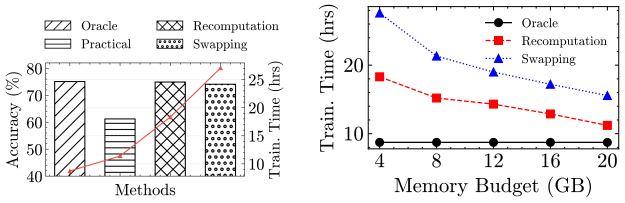
Figure 5: Memory breakdown. Model: Llama-7B. FT denotes fine-tuning.

process, resulting in fewer data contributions to the global model and further impacting overall performance.

2.2.2 Backpropagation-free Fine-tuning.

As shown in Figure 4, LoRA and Adapter effectively reduce the memory overhead of Llama-7B by 62.1% and 53.9%, respectively. However, they still require more than 15GB of memory, which remains insufficient to deploy certain large models on real edge devices. Figure 5 shows the breakdown of memory footprint and analyzes the reason: these activations generated during forward pass account for most of the memory usage, and existing PEFT methods cannot eliminate this overhead. To address this, BP-free fine-tuning methods, such as zero-order optimization [2], have gained attention. BAFFLE [10] and FedZeN [25] apply BP-free training in FL but are not designed for LLMs. FwdLLM [42] is a recent BP-free method tailored for LLMs, requiring only "perturbed inferences" instead of full fine-tuning. By updating parameters directly from gradients computed during the forward pass, FwdLLM reduces memory consumption, making it more suitable for resource-constrained FL environments. From Figure 2, FwdLLM reduces fine-tuning memory requirements from 45~58GB to 3.8GB, enabling 100% client participation. However, this comes with an approximate 5.8% performance drop compared to the oracle baseline.

The decline mainly stems from the inherent limitations of forward gradient estimation, which sacrifices fine-tuning accuracy for computational efficiency. Unlike BP, which provides precise error-correcting feedback through gradients, forward gradient estimation relies on approximate updates, making parameter adjustments less accurate. This imprecision tends to accumulate over time, especially across multiple fine-tuning rounds. Without the precise gradient flow that backpropagation provides, small errors introduced in each update step may compound, resulting in cumulative error that degrades the overall model performance. In non-IID scenarios, this issue becomes more pronounced because non-IID data, which require more precise model adaptation to account for diverse patterns.



(a) Performance comparison with memory budget 4GB.

(b) Fine-tuning time under different memory budget.

Figure 6: System-level memory saving techniques analysis. Model: Llama-7B; Dataset: BoolQ.

2.2.3 System-level Memory Saving Techniques.

System-level memory optimization techniques are widely used to enable model fine-tuning on memory-constrained devices. Recomputation [6, 16] (or gradient checkpointing) reduces memory usage by discarding intermediate activations and recomputing them during BP pass. While effective for memory savings, it incurs additional recomputation overhead, prolonging training time and reducing efficiency. Swapping [15, 30, 31] transfers model components or activations between memory and storage. Though it minimizes in-memory usage, swapping introduces significant I/O overhead, slowing down training due to frequent memory-to-disk transfers. As shown in Figure 6a, both techniques achieve accuracy near the oracle baseline, but the fine-tuning time increases from 8.73 hours to 27.6 hours. This can be attributed to their ability to achieve the same client participation rate as the oracle by efficiently reducing memory overhead, but at the expense of substantial additional I/O operations and computational costs. From Figure 6b, as the memory budget decreases, the required additional time increases from 1.78× to 3.17× due to the need for more recomputation and swapping.

Summary. Although existing techniques effectively reduce the memory overhead of fine-tuning, they either compromise accuracy or introduce additional computational costs, extending the aggregation time. This highlights the necessity of a method that achieves a balanced trade-off across three critical dimensions: reducing memory usage, maintaining model performance, and ensuring process efficiency.

2.3 Opportunity

To address these limitations, we propose a new perspective by leveraging multiple pre-trained LLMs to construct a task-specific model through block-level assembly. Leveraging multiple models with distinct architectures provides better expressive capabilities for different downstream tasks, enabling im-

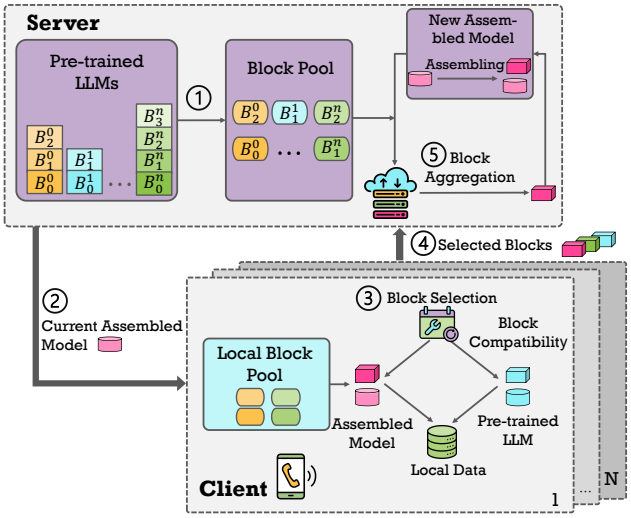


Figure 7: Workflow of assembled model generation.

proved performance for task-specific objectives. First, we divide multiple LLMs into modular transformer blocks, forming a block pool. Then, based on local data characteristics, each client dynamically selects relevant blocks from the shared block pool and assembles them into a task-specific model. This process is accomplished through simple inference operations without the need for BP.

The proposed learning paradigm builds on two key insights: 1) **using pre-trained models as modular building blocks** and 2) **enabling dynamic block assembly for task flexibility**. By selecting blocks based on local data and resources, it tailors model generation to diverse tasks. The block-level optimization reduces memory usage, allowing even low-end devices to participate in FL.

3 AssyLLM Design

3.1 The Fine-tuning Paradigm

Figure 7 represents the overall fine-tuning paradigm of AssyLLM, which can be mainly divided into the following steps. ① **Initialization:** Pre-trained LLMs are manually split into blocks categorized as starting blocks (embedding layer), intermediate blocks (groups of transformer layers), and terminating blocks (output layers). These pre-trained models and candidate block pools are distributed to participating clients. ② **Model Downloading:** In each round, participants receive a candidate assembled model. ③ **Local Block Selection:** Given the current model N_s, each client searches the block pool for compatible blocks. For a candidate block B_{nl}, derived from a pre-trained LLM n up to layer l, clients assemble N_s with B_{nl} and perform two inference passes with the same batch of local data: one on the new assembled model and another on the originating LLM up to layer l. The resulting activations X and Y are compared to compute the compatibility score for B_{nl}. This process is repeated for all candidate blocks. ④ **Block Uploading:** Each client selects the K blocks with the highest compatibility scores and uploads. ⑤ **Server-Side Aggregation:** The server aggregates the uploaded blocks using a weighted voting mechanism (similar to FedAvg). The voting process is based on compatibility scores. A higher score indicates better adaptability to the current assembled model. The K selected blocks with the highest votes, whose parameters are averaged across relevant clients, are sequentially stacked onto the current model N_s to construct the candidate model for the next round. The process continues until a terminating block is selected, the model depth limit is reached, or all paths are explored. The block selection process requires only inference on a small data batch, significantly reducing the data needed.

However, deploying this paradigm in real-world environments presents several challenges that warrant further investigation.

- Q1: *How to compare the compatibility between two blocks?* For a given block, selecting a compatible counterpart from the block pool is essential for effective concatenation in downstream tasks. Higher compatibility between the selected blocks enhances the performance of the resulting assembled model.
- Q2: *How to assemble two different blocks together?* Blocks derived from different pre-trained LLMs exhibit varying features and structures. Directly combining them without careful consideration can negatively impact overall performance, necessitating a more deliberate integration strategy.
- Q3: *How to reduce the additional memory overhead caused by multiple LLMs.* As the block pool comprises multiple LLMs, maintaining it in memory during the block selection process introduces additional memory overhead.

3.2 System Overview

To tackle these challenges, we introduce AssyLLM, an FL system for deploying LLMs that optimizes memory and performance. Furthermore, we design four core techniques: Block Comparator (Section 3.3), Elastic Adapter (Section 3.4), Block Quanter (Section 3.5), and Block Swapper (Section 3.6), as shown in Figure 8. For **Q1**: 1) Block Comparator introduces a well-defined metric to assess block compatibility, facilitating the selection of appropriate blocks for concatenation. For **Q2**: 2) Elastic Adapter addresses model heterogeneity by integrating blocks from different pre-trained LLMs using lightweight adapters, ensuring seamless assembly with minimal information loss. For **Q3**, we design two components to optimize the memory overhead of block pool: 3) Block Quanter applies mixed-precision quantization at the block level to block pool, balancing resource allocation by

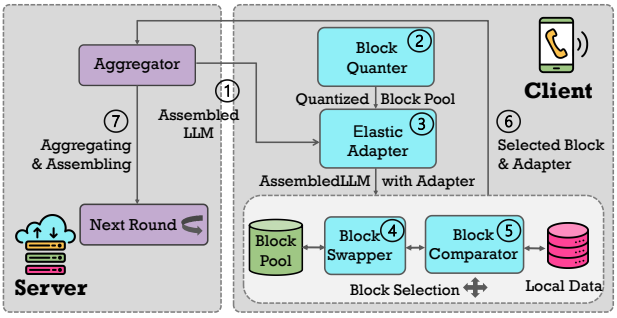
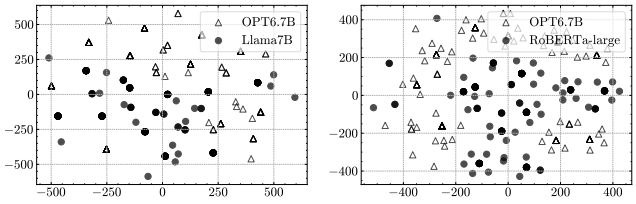


Figure 8: AssyLLM architecture.



(a) OPT-6.7B vs Llama-7B. (b) OPT6.7B vs RoBERT-large.

Figure 9: Feature differences visualization using t-SNE [36].

assigning lower precision to less critical weights and higher precision to key components; 4) Block Swapper further optimizes the swapping pipeline by leveraging the correlation between blocks. Additionally, its pre-loading and pre-swapping mechanisms reduce I/O delays, enabling efficient model assembly on low-resource devices. Together, these techniques enable AssyLLM to overcome memory, fine-tuning efficiency, and computational challenges in FL, enabling real-world FL applications of state-of-the-art billion-scale LLMs.

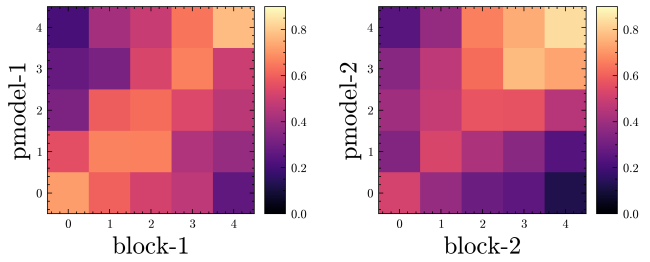
3.3 Block Comparator

Block compatibility is critical when assembling a model from block pool. To guide block selection, robust comparison metrics are necessary. Centered Kernel Alignment (CKA) [18] is a widely used method for measuring the similarity between activation sets, commonly applied to analyze layer representations in neural models. We extend CKA to LLMs by comparing activations from different transformer blocks, providing insights into feature evolution across layers and models. In this work, CKA serves as a key metric to evaluate block compatibility. Specifically, we compute the CKA score using activations K and L from the candidate assembled LLM and corresponding pre-trained LLM (detailed in Section 3.1). A higher CKA score indicates better alignment, aiding in selecting the most compatible blocks to optimize model performance. The CKA score is calculated as follows:

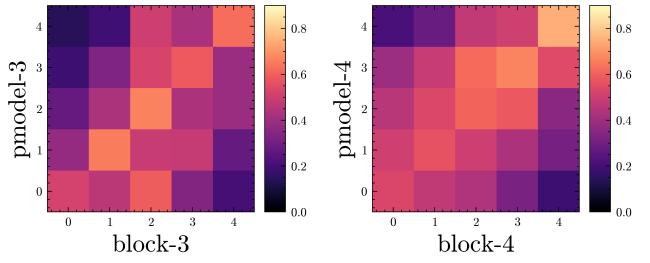
$$CKA(K,L) = \frac{\textbf{HSIC}(K,L)}{\sqrt{\textbf{HSIC}(K,K)\textbf{HSIC}(L,L)}} \quad (1)$$

where **HSIC** is Hilbert-Schmidt Independence Criterion [13]. For the linear kernels, **HSIC** is:

$$\textbf{HSIC}(K,L) = ||cov(X^T X, Y^T Y)||_F^2 \quad (2)$$



(a) cka=0.78, cor=0.61, acc=0.74. (b) cka=0.81, cor=0.49, acc=0.71.



(c) cka=0.63, cor=0.41, acc=0.65. (d) cka=0.72, cor=0.53, acc=0.69.

Figure 10: CKA Heatmaps for four various candidate blocks; *pmodel* denotes pre-trained LLM where the block originates from; *cor* denotes layer-correlation metrics; *acc* denotes final assembled model performance; Dataset: BoolQ.

CKA helps evaluate block compatibility but focuses only on final activations of block, overlooking intermediate layer differences in LLMs, which can lead to suboptimal block selection and reduced model performance. To explore these differences, we conducted an experiment on BoolQ by feeding the same input to various LLMs and extracting their intermediate features. We visualized them using t-SNE [36], as shown in Figure 9. The results highlight that, despite identical inputs, different architectures produce distinct intermediate feature distributions. To address CKA's limitation, we propose layer-correlation (COR), a new metric that compares activation distribution similarities across layers using KL divergence [20]. Given two activation distributions P and Q for a specific layer, KL divergence is defined as:

$$D_{KL}(P \parallel Q) = \sum_i P(i) \log\left(\frac{P(i)}{Q(i)}\right) \quad (3)$$

Where $P(i)$ denotes the probability of activation i in the first distribution, and $Q(i)$ denotes the probability of activation i in the second distribution.

To calculate COR, we sum the KL divergence between corresponding layers within two blocks. This is done once during initialization. To validate the effectiveness of these metrics, we conduct experiments on BoolQ using the LLMs described in Section 2.3. Given an assembled LLM composed of four blocks, we set four different candidate blocks to assemble and record corresponding CKA, COR scores, and accuracy of the generated assembled model. Figure 10 shows the heatmap for CKA distributions, revealing key insights: 1) Blocks with

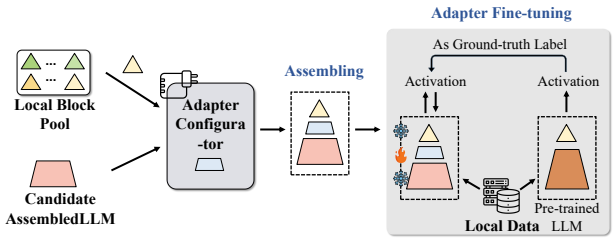


Figure 11: Elastic Adapter fine-tuning.

high CKA and COR achieve better performance (e.g., *Block 1* vs *Block 3*); 2) High CKA alone doesn't ensure optimal selection (e.g., *Block 1* vs *Block 2*); 3) High COR alone is also insufficient (e.g., *Block 2* vs *Block 4*). These findings motivate us to integrate CKA and COR to block compatibility score for reliable block selection in each round to identify the best candidate (e.g., *Block 1*) that maximizes accuracy.

3.4 Elastic Adapter

Assembling blocks from different pre-trained LLMs is non-trivial due to structural and semantic mismatches, which can significantly degrade performance. Specifically, there are three primary types of mismatches in LLMs: dimensional misalignment, semantic inconsistency, and attention mechanism differences. We introduce Elastic Adapter, an adaptive adapter fine-tuning approach that addresses these mismatches by introducing adapters that align the outputs and inputs of blocks from heterogeneous models. 1) *Dimensional Misalignment*: This issue arises when the output of one block does not match the input dimensions of the next. To overcome this, we introduce lightweight linear transformations to project outputs into the correct dimensional space. 2) *Semantic Inconsistency*: Even when block structures align, the features they extract may differ semantically, causing further mismatches. To address this, we implement a cross-attention mechanism that uses one block's output as the query and the next block's output as the key and value. This process aligns semantic content between blocks, reducing feature mismatches and ensuring smoother block integration. The cross-attention alignment process is defined as:

$$\mathbf{O}_{cross} = softmax\left(\frac{\mathbf{Q}\mathbf{K}^{\top}}{\sqrt{d_k}}\right)\mathbf{V}, \tag{4}$$

where $\mathbf{Q} = \mathbf{W}_Q\mathbf{H}_{out1}$ is the query matrix derived from the output of the first block $\mathbf{H}_{out1}$, $\mathbf{K} = \mathbf{W}_K\mathbf{H}_{out2}$ and $\mathbf{V} = \mathbf{W}_V\mathbf{H}_{out2}$ are the key and value matrices derived from the output of the second block $\mathbf{H}_{out2}$. Here, $\mathbf{W}_Q$, $\mathbf{W}_K$, and $\mathbf{W}_V$ are learnable projection matrices, and d_k is the attention dimension used to scale the dot-product for numerical stability.

Optionally, a linear transformation can be applied to ensure dimensional consistency for subsequent processing:

$$\mathbf{H}_{aligned} = \mathbf{W}_{out}\mathbf{O}_{cross}, \tag{5}$$

where $\mathbf{W}_{out}$ is a learnable projection matrix.

Table 1: Assembled model performance and memory overhead of the block pool with different quantization methods comparison. BP denotes the block pool. Dataset: BoolQ. Block pool: Llama-7B, OPT-6.7B, Vicuna-7B, BERT-base, and RoBERT-large.

Precision	BP Overhead	Model Accuracy (%)
FP16	42.2GB	75.8 ± 2.5
INT8	21.1GB	73.1 ± 4.7
INT4	11.5GB	67.3 ± 8.9

3) *Attention Mechanism Differences*: the last challenge is handling different attention mechanisms, such as multi-head versus single-head attention. We address this by designing adapters that pool or expand attention outputs based on block configurations, ensuring compatibility without altering model structures. We apply lightweight fine-tuning by freezing most model parameters and training only adapters with local client data. Notably, not all block assemblies require trainable adapters. In most cases, the mismatch between blocks primarily falls into the first category of challenge (*Dimensional Misalignment*), where a simple projection matrix is often sufficient to address it without introducing additional complexity. Furthermore, trainable adapters are used only for significant mismatches, primarily in the final blocks. As a result, most intermediate activations can be discarded during forward propagation, reducing memory overhead. Figure 11 illustrates the Elastic Adapter workflow.

3.5 Block Quanter

For **Q3**, the block pool size equals the combined size of multiple pre-trained LLMs and need to remain in memory throughout the block selection process, leading to substantial memory overhead. Quantization [11, 22] is a common technique to reduce memory usage, but excessive compression can harm accuracy. As shown in Table 1, FP16 precision results in over 40GB of memory usage, far exceeding edge device limits. INT8 reduces memory to 21GB with minor performance loss, while INT4 significantly degrades accuracy, resulting in a 5.8% accuracy drop with an 8.9% fluctuation. On the other hand, mixed-precision techniques [22, 37] apply varying precision levels at the layer level, requiring layer-wise sensitivity analysis to assess the impact of precision changes. This process involves additional forward passes or statistical measurements, resulting in significant computational overhead during model preparation and fine-tuning.

To address this, we propose Block Quanter, a block-wise mixed-precision quantization method. Unlike prior work [22, 37], which typically applies layer-wise precision adjustments, our approach focuses solely on the impact of internal weights on block output activations, which directly affects block compatibility and selection. By evaluating the sensitivity of weight to the entire block's output rather than individual

Algorithm 1 Block Quanter

Input: Pre-trained LLM M, block pool $\mathcal{B} = \{b_1, b_2, ..., b_n\}$, block activation $\mathbf{A}_b$, sparsity threshold τ_s, sensitivity threshold τ_a.

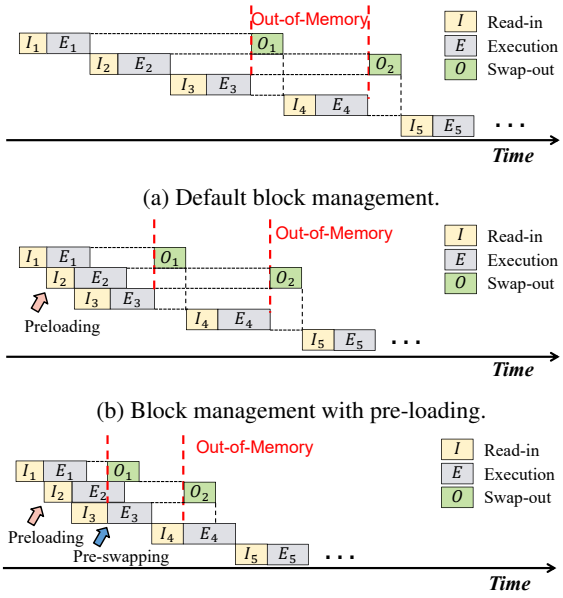
Output: Weight importance scores $\mathcal{S}_w$.

// Weight Sparsity Analysis
for *each block* $b \in \mathcal{B}$ **do**
  for *each weight* $w \in b$ **do**
    $S_s(w) \leftarrow SparsityMetric(w)$
    if $S_s(w) < \tau_s$ **then**
      Mark w as unimportant
    end if
  end for
end for
// Activation Sensitivity Analysis
for *each block* $b \in \mathcal{B}$ **do**
  for *each remaining weight* $w \in b$ **do**
    $\Delta\mathbf{A}_b \leftarrow RandomPerturbation(w)$ or $Masking(w)$
    $S_a(w) \leftarrow ActivationImpact(\Delta\mathbf{A}_b)$
    if $S_a(w) < \tau_a$ **then**
      Mark w as unimportant
    end if
  end for
end for
// Bottom-Up Sensitivity Analysis
for *each block* $b \in \mathcal{B}$ **do**
  $L \leftarrow$ *Number of layers in block b*
  for $l = L$ **down to** 1 **do**
    for *each remaining weight* $w \in layer\ l$ **do**
      $C(w, \mathbf{A}_b) \leftarrow Correlation(w, \mathbf{A}_b)$
      $\mathcal{S}_w(w) \leftarrow C(w, \mathbf{A}_b)$
    end for
  end for
end for
return $\mathcal{S}_w$

layers, Block Quanter avoids the need for layer-level precision tuning across multiple Transformer layers. This block-wise granularity significantly reduces the computational overhead compared to layer-wise methods, as it requires fewer forward passes and sensitivity analyses, making it more efficient and practical for resource-constrained devices. Block Quanter performs offline multi-step quantization by first analyzing weight sparsity in block and then evaluating activation sensitivity through random perturbations and masking to identify critical weights for higher precision allocation. For the random perturbation method:

$$S_a(w) = \frac{\|\mathbf{A}_b(w) - \mathbf{A}_b(w+\Delta w)\|_2}{\|\mathbf{A}_b(w)\|_2}, \quad \Delta w \sim \mathcal{N}(0, \sigma^2), \quad (6)$$

where $\mathbf{A}_b(w)$ represents the block's original output activation,



(a) Default block management.

(b) Block management with pre-loading.

(c) Memory-aware block management with pre-loading and pre-swapping (ours).

Figure 12: Different pipeline strategies for block swapping.

$\mathbf{A}_b(w+\Delta w)$ is the activation after adding random perturbation Δw, and $\|\cdot\|_2$ denotes the L_2 norm.

For the weight masking method:

$$S_a(w) = \|\mathbf{A}_b(w) - \mathbf{A}_b(w=0)\|_2, \quad (7)$$

where $\mathbf{A}_b(w=0)$ is the output activation after masking weight w to zero.

Based on the results, insignificant weights are discarded. A bottom-up sensitivity analysis then re-evaluates the correlation between the remaining weights and the output, starting from the block's output layer to its input layers. This refined analysis ensures that critical weights are retained at high precision, while less important ones are quantized to lower precision. The entire process is performed offline, ensuring minimal additional resource consumption during the block selection phase and significantly reducing memory usage of block pool without compromising performance. The complete process is outlined in Algorithm 1.

3.6 Block Swapper

While BlockQuanter reduces block pool memory usage, it remains insufficient. As shown in Table 1, a block pool composed of five pre-trained LLMs still requires over 11GB of memory, even with INT4 compression, exceeding the typical capacity of edge devices (4~8GB). Furthermore, incorporating additional LLMs improves adaptability across diverse downstream tasks and enhances the assembled model's performance, but it also significantly increases the block pool size, resulting in higher memory costs. To address this, we propose Block Swapper, a system-level solution to further optimize memory usage. The Block Swapper design begins

Algorithm 2 Block Swapper.

Input: Block Pool $\mathcal{B}$, Memory Budget M_{max}, Current assembled LLM $\mathcal{M}_{assembled}$.

```
// Step 1: Initialize Memory Budget
M_used ← 0
for each block b ∈ B do
    // Step 2: Check if block b is in memory
    if b ∈ Memory then
        Continue with assembly
    else
        // Step 3: Perform Swap-In
        if M_used + size(b) ≤ M_max then
            Pre-loading b from external storage
            M_used ← M_used + size(b)
        else
            // Step 4: Perform Swap-Out
            b_out ← BlockCorrelation(b, B)
            Pre-swapping out b_out to external storage
            M_used ← M_used − size(b_out)
            Load b from external storage
            M_used ← M_used + size(b)
        end if
    end if
    // Step 5: Assemble Model
    Assemble block b into M_assembled
    BlockCompatibilityCalculation(b, M_assembled)
end for
```

by setting a memory budget for each device, determining how many blocks can be stored in memory. When a required block is missing, a swap-in operation is initiated. If memory is full, a swap-out unloads less relevant blocks to storage. **Block correlation.** The block swapping decision combines a Least Recently Used (LRU) strategy with block correlation metrics. The relevance of blocks is determined by their feature correlation, previously analyzed in Section 3.3 and directly applied here. Blocks with lower correlation to the current model are prioritized for swapping out.

To mitigate I/O overhead from frequent swaps (Figure 12a), we further introduce a pre-loading mechanism, pre-loading blocks into memory to ensure smoother execution (Figure 12b). Additionally, a memory-aware pre-swapping strategy estimates upcoming memory needs and performs swap-out operations in advance to reduce I/O waiting times (Figure 12c). In summary, Block Swapper optimizes the pipeline by leveraging block correlations and further reduces delays through pre-loading and pre-swapping, enabling efficient memory management. Algorithm 2 outlines the process.

Table 2: Specifications of tested devices.

Device	Mem	GPU	CPU
Nvidia Jetson TX2	8G	256 CUDA Cores (Pascal)	A57 (6C)
Nvidia Jetson Nano	4G	128 CUDA Cores (Maxwell)	A57 (4C)

4 Evaluation

4.1 Experimental Setup

LLMs, Block Pool, and Datasets. We select five pre-trained LLMs: Llama-7B, OPT-6.7B, BERT-base, Vicuna-7B, and RoBERTa-large from Huggingface [39], which are chosen for their widespread adoption and distinct characteristics on edge devices, covering diverse architectures, parameter sizes, and training objectives. Table 3 presents the model sizes of the tested LLMs. To optimize block-based assembly, we use an adaptive segmentation strategy. Shallow and deep layers, which have minimal semantic differences, are grouped into larger blocks (e.g., 6–8 layers per block) to reduce complexity while maintaining functionality. Conversely, middle layers with more significant semantic variation are divided into smaller blocks (e.g., 2–4 layers per block) to improve flexibility and minimize mismatch during assembly. As a result, Llama-7B, OPT-6.7B, BERT-base, Vicuna-7B, and RoBERTa-large consist of 6, 6, 4, 6, and 6 blocks, respectively, forming a block pool with 28 blocks.

We evaluate our approach on three benchmark datasets: BoolQ [7], PIQA [3], and OpenbookQA (OBQA) [9]. BoolQ is a question-answering dataset focused on yes/no questions based on a passage. PIQA involves physical reasoning tasks, challenging the model's ability to predict plausible solutions for real-world scenarios. OBQA tests commonsense knowledge and reasoning, requiring answers based on general science facts. These datasets were selected for their diversity in task types—reading comprehension, physical reasoning, and commonsense knowledge—ensuring a comprehensive evaluation of our method's adaptability.

Baselines. We evaluate our method against two groups of baselines to demonstrate its effectiveness across both algorithmic and system-level aspects. Algorithm-level baselines include **FT-oracle**, representing full-parameter fine-tuning under ideal memory conditions as the upper bound; **FT-practical**, which accounts for memory limitations; **LoRA** [14], a widely used PEFT technique; **QLoRA** [8], which extends LoRA with quantization; **FedAdapter** [4], an FL approach utilizing adapter tuning with layer freezing; and **FwdLLM** [42], the SOTA BP-free method for memory constraints in FL. System-level baselines include **Recomputation** [6, 19], which reduces memory usage by recomputing activations during BP, and **Swapping** [15, 30], which offloads computations or parameters to external storage to save GPU memory. These baselines offer a comprehensive comparison of AssyLLM's performance in terms of both algorithmic flexibility and system-level efficiency.

Implementation Details. We implement AssyLLM on a hy-

Table 3: Model size (FP16).

Model	Size
Llama-7B	14GB
OPT-6.7B	13.4GB
BERT-base	220MB
Vicuna-7B	14GB
RoBERT-large	710MB

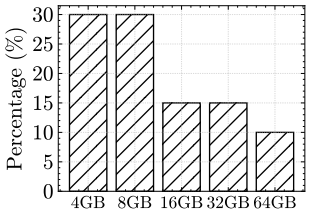
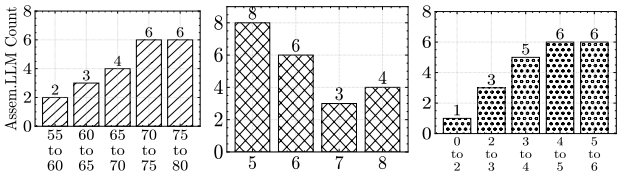


Figure 13: Mem. distr.



(a) Accuracy (%). (b) Block numbers. (c) Size (GB).

Figure 14: Statistics for generated assembled models on BoolQ.

brid platform combining simulation and hardware. Simulation experiments are conducted on two Nvidia A100 GPUs to emulate clients with high memory budgets, while on-device experiments are performed on Nvidia Jetson TX2 and Nano, representing memory-constrained edge devices. Device specifications are provided in Table 2. We use Hugging Face Transformers [39] and PyTorch [27] for on-device fine-tuning and inference, monitoring memory usage with htop. For FL, we follow the setup in Section 2.1, with 200 users split into 4 groups based on memory budgets, as shown in Figure 13. Aggregation uses FedAVG [26]; the maximum sequence length of 512 for OBQA and 256 for others. A minimum 4GB memory budget is allocated for *Swapping*, *Recomputation*, and *AssyLLM* to ensure client participation. Baseline fine-tuning needs 5 local epochs with a batch size of 16 and 500 global epochs. For AssyLLM, the process of selecting the next block for the current candidate model is repeated for 5 rounds, with 3 blocks evaluated per round. It is important to note that block type does not directly influence the aggregation outcome, as all blocks are evaluated using the same compatibility metric. In practice, assembling blocks of different types naturally tends to produce lower compatibility scores due to larger feature distribution mismatches, making them less likely to be selected. We use a learning rate of 0.01, which is higher than the standard (e.g., 5e-5) used for full-model fine-tuning. This is intentional and empirically justified, as only lightweight adapters are trained while the backbone remains frozen. The higher rate accelerates convergence and mitigates underfitting due to limited adapter capacity. Sensitivity analysis confirms this setup achieves efficient training with minimal impact on accuracy. The batch size for CKA computation is 32. The assembly process typically requires fewer than 100 epochs, with each epoch taking less time than baseline methods. For Block Quanter, we use the INT8/INT4 combination for mixed-typed quantization with GPTQ [11].

4.2 Overall Performance

Throughout the assembling process, 21 assembled LLMs are generated, with statistics shown in Figure 14, demonstrating the diversity of models created from various block combinations. All generated models are stored on the server, and each client receives one model per round for block selection, minimizing additional memory overhead for edge devices. After model generation, we select models with high block compatibility, as it typically leads to better accuracy and ensures top-performing models are chosen. We use llama-7B as the LLM for baseline methods, applying INT8 quantization with GPTQ. Table 4 and Figure 15 present the comparison results with algorithm-level baselines and Table 5 and Figure 16 for the system-level baselines.

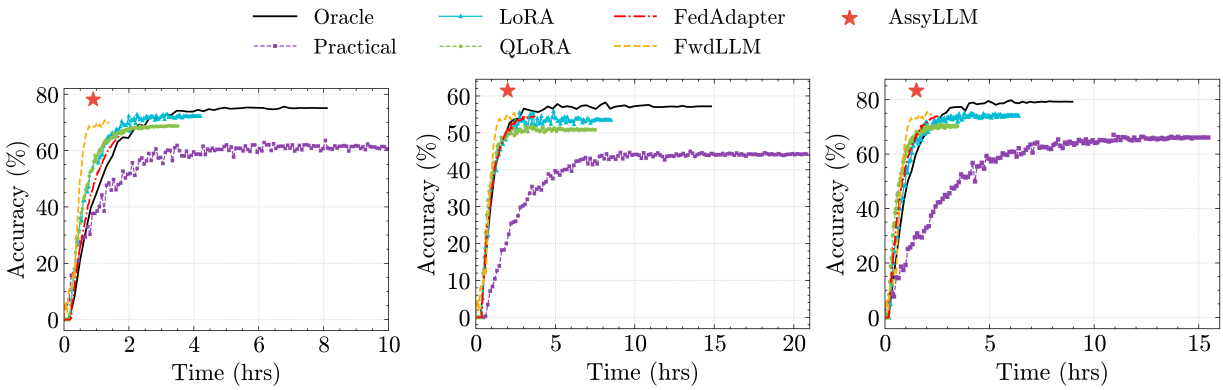
Superior Accuracy. From Table 4 and Figure 15, compared to full fine-tuning, AssyLLM achieves 16.75%, 17.16%, and 18.26% absolute improvements across three datasets due to reducing memory requirements and enabling contributions from users previously constrained by memory limitations. For parameter-efficient fine-tuning baselines like LoRA, QLoRA, and FedAdapter, AssyLLM shows improvements of up to 8.24%, 13.09%, and 11.57%, as these methods reduce weight precision or limit fine-tuning, causing performance degradation. Compared to BP-free methods like FwdLLM, AssyLLM improves by 6.14%, 8.88%, and 7.58%, as FwdLLM's reliance on inference perturbations leads to instability during aggregation. AssyLLM's concatenation method avoids these issues while maintaining superior performance. From Table 5 and Figure 16, AssyLLM outperforms existing memory-saving techniques by up to 3.77%, 5.26%, and 6.18%. This is due to constructing a block pool with multiple LLMs of diverse structures, enriching the model pool, and enhancing the adaptability and expressiveness of the generated models for downstream tasks.

Speedier Convergence. From Table 4, for algorithm baselines, AssyLLM achieves up to 12.92×, 14.67×, and 10.97× speedup across three datasets, thanks to the elimination of BP-related computations, which drastically reduces fine-tuning time. From Table 5, for system-level baselines, the speedup is even more significant, reaching up to 30.04×, 28.71×, and 26.75×. This improvement is due to the substantial additional time introduced by I/O exchanges or activation recomputations, which greatly extend the overall aggregation process.

Robustness to non-IID data. We evaluated performance under various non-IID settings and conducted multiple experiments with different initialization settings. AssyLLM and baselines both utilized the FedAvg aggregation algorithm, with results shown in Figure 17. As depicted, as the local data distribution becomes more skewed, the baseline experiences varying degrees of performance degradation, with accuracy drops of up to 14% and increased bias. In contrast, while AssyLLM also faces some impact (a 4.6% absolute accuracy drop), it still demonstrates up to a 27.2% improvement

Table 4: Test results (top-1 accuracy and speedup) of **algorithm-level memory saving schemes** on BoolQ, PIQA, OBQA in non-IID scenario. *FT-oracle* represents the experiment conducted without memory limitation, denotes as the upper bound.

Dataset	FT-practical		FT-oracle		LoRA		QLoRA		FedAdapter		FwdLLM		AssyLLM	
	Speedup	Acc	Speedup	Acc	Speedup	Acc	Speedup	Acc	Speedup	Acc	Speedup	Acc	Speedup	Acc
BoolQ	1×	61.37	1.45×	75.11	3.72×	71.32	4.83×	69.88	6.32×	72.96	9.83×	71.98	**12.92**×	**78.12**
PIQA	1×	66.23	1.37×	79.22	4.10×	74.14	4.99×	70.3	6.87×	75.81	10.21×	74.51	**14.67**×	**83.39**
OBQA	1×	44.21	1.47×	57.29	3.43×	53.55	4.31×	50.90	6.02×	54.78	9.15×	54.89	**10.97**×	**62.47**



(a) BoolQ. (b) OBQA. (c) PIQA.

Figure 15: Efficiency comparison with **algorithm-level baselines**. Model: Llama-7B.

Table 5: Test results (top-1 accuracy and speedup) with **system-level memory saving baselines** on BoolQ, PIQA, OBQA in non-IID scenario. See Table 4 for further reference.

Dataset	Recomputation		Swapping		AssyLLM	
	Speedup	Acc	Speedup	Acc	Speedup	Acc
BoolQ	0.65×	74.91	0.43×	74.35	**12.92**×	**78.12**
PIQA	0.68×	78.96	0.51×	78.13	**14.67**×	**83.39**
OBQA	0.51×	56.88	0.41×	56.19	**10.97**×	**62.47**

over the baseline, offering more stable performance. This improvement stems from the fact that, in our method, local data only guides block selection and fine-tunes lightweight adapters, without altering the blocks themselves. As a result, the generated LLM is less affected by the local data distribution compared to baselines. Furthermore, our approach is orthogonal to other methods that address non-IID issues, like Harmony [34] and Oort [21], and can be seamlessly integrated with them for further optimization.

4.3 System Cost

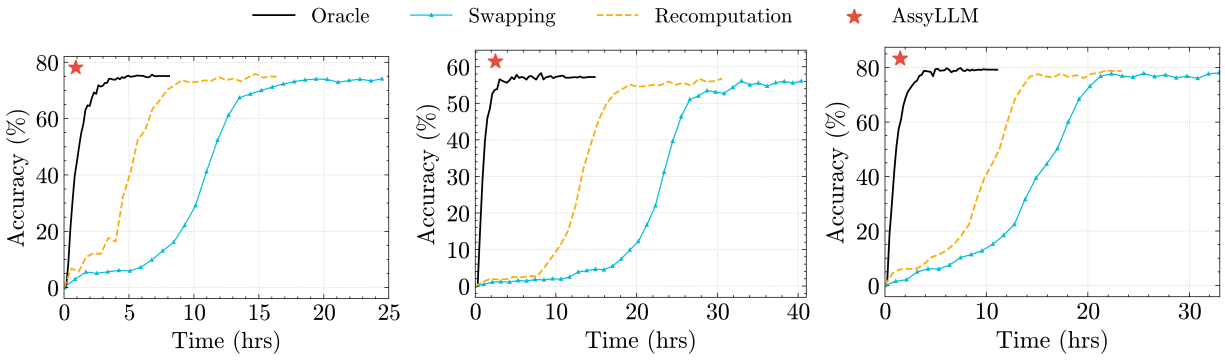
Memory overhead reduction. As shown in Figure 18, AssyLLM reduces memory consumption by 92% compared to full fine-tuning and 63.6% compared to PEFT baselines. This is due to AssyLLM's design, which limits operations to the forward pass, reducing memory cost to only the block pool and lightweight adapter fine-tuning. In contrast, while other system-level and BP-free methods reduce memory usage and also achieve 100% client participation, they often sacrifice accuracy or prolong training time, limiting their real-world efficiency. Moreover, mainstream inference optimization techniques [23, 24] can be integrated with our approach to further reduce memory overhead.

Energy and communication cost reduction. As shown in Figure 19, AssyLLM reduces energy consumption by 95.01% and inter-device communication overhead by 99.1% compared to full fine-tuning. Our method is also highly efficient relative to PEFT or BP-free methods, achieving up to 88.1% energy reduction and 94.2% decrease in communication overhead. This reduction is due to the fact that, unlike traditional training methods, our approach only requires clients to upload block selection results and a lightweight adapter each round. Since the block pool is shared between the server and clients, the block selection results are transmitted as an index, typically requiring only a few bytes. In contrast to traditional methods, which require uploading entire or partial model weights or updates, our method significantly reduces communication overhead.

4.4 Effectiveness of Each Component

Impact of Block Comparator. We compared the impact of using two individual metrics (CKA and COR) for block selection on the final performance. As shown in the Figure 10, neither metric alone can identify the optimal blocks, resulting in a 9.1% and 3.4% performance drop.

Impact of Block Quanter. We compare Block Quanter with different precision schemes. As shown in Figure 20a, compared to FP16, AssyLLM reduces memory consumption by 70.2% with only a 1.1% accuracy loss. For INT4, our method increases memory overhead by 10.1% but improves accuracy by 8.1%. This shows that Block Quanter effectively reduces block pool memory overhead while preserving accuracy through block-specific mixed-precision quantization.



(a) BoolQ. (b) OBQA. (c) PIQA.

Figure 16: Efficiency comparison with **system-level baselines**. Model: Llama-7B.

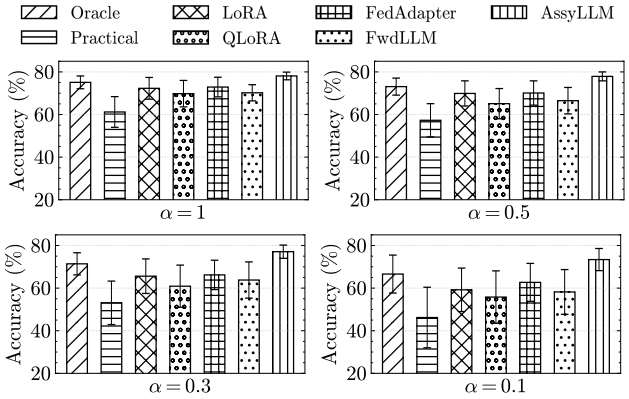


Figure 17: Performance comparison under various non-IID settings in BoolQ. α denotes the concentration parameter in Dirichlet distribution.

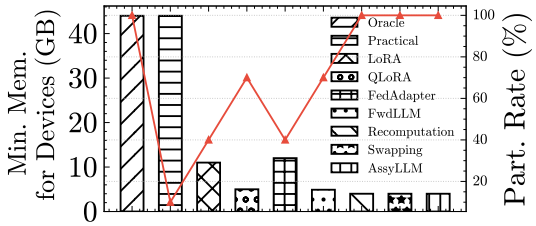
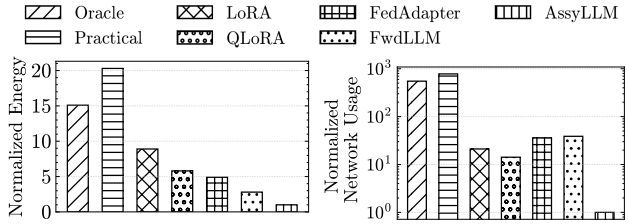


Figure 18: Peak memory footprint and client participation rate comparison on BoolQ. *Min. Mem. for Devices* denotes the minimum memory required for the participant to apply the corresponding methods.

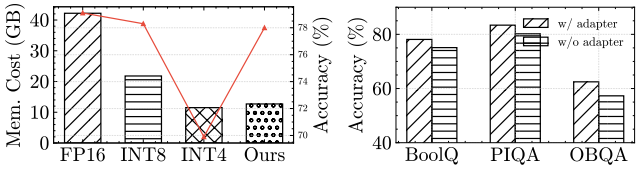
Impact of Elastic Adapter. We compare AssyLLM with a baseline approach that uses a simple dimensional transformation matrix to connect blocks. As shown in Figure 20b, adding the assembling adapter improves performance by 3.02%, 3.19%, and 5.17% on datasets. The improvement demonstrates the effectiveness of our method in facilitating seamless block integration and enhancing performance.

Impact of Block Swapper. To verify the effectiveness of Block Swapper, we evaluated two pipeline management strategies: one without pre-loading and pre-swapping (system default), and another without pre-swapping, as shown in Figure 12. The results in Figure 21 show that as the number of block swaps increases, incorporating pre-loading reduces local time by up to 30.21% compared to the default pipeline. Additionally, adding pre-swapping on top of pre-loading further reduces total time by 68.12%. This significantly minimizes the additional latency from block swapping, improving overall system efficiency.



(a) Total energy cost. (b) Total network cost.

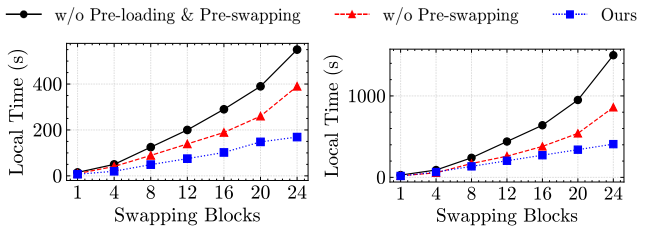
Figure 19: Resource cost of AssyLLM and baselines. Dataset: BoolQ.



(a) Block Quanter on BoolQ. (b) Elastic Adapter.

Figure 20: Ablation study for Block Quanter and Elastic Adapter.

Component-wise Contribution Summary. Each component in AssyLLM contributes to overall performance. Block Comparator selects compatible blocks using CKA and COR scores. Elastic Adapter integrates mismatched blocks by activating adapters only when significant semantic divergence is detected. Block Quanter reduces memory via mixed-precision quantization; INT8/INT4 yields ~70% savings with ~1.1% accuracy loss, while maintaining robust compatibility metrics. Block Swapper improves memory efficiency through correlation-aware prefetching and pre-swapping. Together, these modules enhance accuracy, memory usage, and convergence, as confirmed by our ablation results.



(a) BoolQ. (b) OBQA.

Figure 21: Ablation study for Block Swapper.

5 Discussion

Criteria for selecting pre-trained models. While AssyLLM shows promising results, selecting the appropriate combination of pre-trained LLMs for the block pool remains a non-trivial task. We prioritize architectural diversity to ensure the assembled model captures a broad spectrum of representational patterns, incorporating both decoder-only models (e.g., LLaMA, OPT, Vicuna) and encoder-only models (e.g., BERT, RoBERTa). Empirical results indicate that such heterogeneity improves generalization across downstream tasks. However, expanding the block pool increases memory and computation overhead, and not all combinations yield consistent performance. Our experiments reveal fluctuations across different LLM sets, highlighting the need for a more principled strategy to guide model selection. In future work, we plan to explore adaptive construction methods based on task embeddings, domain-aligned pretraining, or proxy task performance to optimize block pool composition.

System Robustness and Scalability. While AssyLLM consists of four interacting components—Block Comparator, Elastic Adapter, Block Quanter, and Block Swapper—each is implemented as an independent and lightweight module (300~600 lines in Python), coordinated through shared interfaces without tight coupling. For example, the Comparator and Quanter operate purely on forward activations, while the Adapter is only triggered when structural mismatches occur. This modularity enhances robustness against architectural variations and supports easy reuse across different LLM workflows.

In terms of scalability, while the current design targets edge devices, AssyLLM can be extended to larger LLMs (e.g., LLaMA-13B or 30B) with minimal architectural changes. The primary challenges lie in the memory overhead due to increased block pool size and potential I/O pressure during block swapping. These can be addressed via hierarchical block indexing, storage prefetching, or streaming-based execution, which we leave as promising directions for future work.

Why block-level assembly works better? Traditional fine-tuning incurs high memory overhead, restricting participation to high-resource devices and reducing data diversity in FL. In contrast, our method removes most BP-related memory costs, enabling participation from low-resource devices. By assembling blocks from diverse pre-trained LLMs, our framework captures a broader range of representational patterns. Moreover, the inherent redundancy in LLMs allows them to tolerate moderate structural or semantic inconsistencies, so the assembled model can still perform effectively even if individual blocks are not perfectly aligned.

6 Conclusions

In this paper, we propose AssyLLM, a memory-efficient federated fine-tuning framework using pre-trained LLMs block assembly, effectively reducing BP-related overhead. The four key components—Block Comparator, Elastic Adapter, Block Quanter, and Block Swapper—effectively enhance the method's efficiency by addressing block compatibility, mismatches and reducing the memory footprint of block pool. Experiments demonstrate that AssyLLM reduces memory requirements, enhances model accuracy, and accelerates training compared to conventional FedLLM methods.

Acknowledgments

This work was supported by Science and Technology Development Fund (FDCT) of Macau SAR (File no. 0107/2024/RIA2), MYRG-GRG2024-00180-IOTSC, MYRG-GRG2023-00211-IOTSC-UMDF, and EF2024-00180-IOTSC. Please ask Dr. Li Li (llili@um.edu.mo) for correspondence.

References

[1] How much ram does your android phone really need in 2023? https://www.androidauthority.com/how-much-ram-do-i-need-phone-3086661/. Accessed: 2023.

[2] Andrew G Barto and Michael I Jordan. Gradient following without back-propagation in layered networks. In *1st Int. Conference Neural Nets, San Diego*, volume 2, 1987.

[3] Yonatan Bisk, Rowan Zellers, Jianfeng Gao, Yejin Choi, et al. Piqa: Reasoning about physical commonsense in natural language. In *Proceedings of the AAAI conference on artificial intelligence*, volume 34, pages 7432–7439, 2020.

[4] Dongqi Cai, Yaozong Wu, Shangguang Wang, Felix Xiaozhu Lin, and Mengwei Xu. Fedadapter: Efficient federated learning for modern nlp. *arXiv preprint arXiv:2205.10162*, 2022.

[5] Chaochao Chen, Xiaohua Feng, Jun Zhou, Jianwei Yin, and Xiaolin Zheng. Federated large language model: A position paper. *arXiv preprint arXiv:2307.08925*, 2023.

[6] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. Training deep nets with sublinear memory cost. *arXiv preprint arXiv:1604.06174*, 2016.

[7] Christopher Clark, Kenton Lee, Ming-Wei Chang, Tom Kwiatkowski, Michael Collins, and Kristina Toutanova. Boolq: Exploring the surprising difficulty of natural yes/no questions. *arXiv preprint arXiv:1905.10044*, 2019.

[8] Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. Qlora: Efficient finetuning of quantized llms. *Advances in Neural Information Processing Systems*, 36, 2024.

[9] Jesus Escudero-Sahuquillo, Pedro J Garcia, Francisco J Quiles, Jose Flich, and Jose Duato. Obqa: Smart and cost-efficient queue scheme for head-of-line blocking elimination in fat-trees. *Journal of Parallel and Distributed Computing*, 71(11):1460–1472, 2011.

[10] Haozhe Feng, Tianyu Pang, Chao Du, Wei Chen, Shuicheng Yan, and Min Lin. Does federated learning really need backpropagation. *arXiv preprint arXiv:2301.12195*, 2023.

[11] Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. Gptq: Accurate post-training quantization for generative pre-trained transformers. *arXiv preprint arXiv:2210.17323*, 2022.

[12] Bela A Frigyik, Amol Kapila, and Maya R Gupta. Introduction to the dirichlet distribution and related processes. *Department of Electrical Engineering, University of Washignton, UWEETR-2010-0006*, 6:1–27, 2010.

[13] Arthur Gretton, Olivier Bousquet, Alex Smola, and Bernhard Schölkopf. Measuring statistical dependence with hilbert-schmidt norms. In *International conference on algorithmic learning theory*, pages 63–77. Springer, 2005.

[14] Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. Lora: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685*, 2021.

[15] Chien-Chin Huang, Gu Jin, and Jinyang Li. Swapadvisor: Pushing deep learning beyond the gpu memory limit via smart swapping. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1341–1355, 2020.

[16] Paras Jain, Ajay Jain, Aniruddha Nrusimha, Amir Gholami, Pieter Abbeel, Joseph Gonzalez, Kurt Keutzer, and Ion Stoica. Checkmate: Breaking the memory wall with optimal tensor rematerialization. *Proceedings of Machine Learning and Systems*, 2:497–511, 2020.

[17] Liuyi Jin, Tian Liu, Amran Haroon, Radu Stoleru, Michael Middleton, Ziwei Zhu, and Theodora Chaspari. Emsassist: An end-to-end mobile voice assistant at the edge for emergency medical services. In *Proceedings of the 21st Annual International Conference on Mobile Systems, Applications and Services*, pages 275–288, 2023.

[18] Simon Kornblith, Mohammad Norouzi, Honglak Lee, and Geoffrey Hinton. Similarity of neural network representations revisited. In *International conference on machine learning*, pages 3519–3529. PMLR, 2019.

[19] Vijay Anand Korthikanti, Jared Casper, Sangkug Lym, Lawrence McAfee, Michael Andersch, Mohammad Shoeybi, and Bryan Catanzaro. Reducing activation recomputation in large transformer models. *Proceedings of Machine Learning and Systems*, 5:341–353, 2023.

[20] Solomon Kullback and Richard A Leibler. On information and sufficiency. *The annals of mathematical statistics*, 22(1):79–86, 1951.

[21] Fan Lai, Xiangfeng Zhu, Harsha V Madhyastha, and Mosharaf Chowdhury. Oort: Efficient federated learning via guided participant selection. In *15th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 21)*, pages 19–35, 2021.

[22] Ji Lin, Jiaming Tang, Haotian Tang, Shang Yang, Wei-Ming Chen, Wei-Chen Wang, Guangxuan Xiao, Xingyu Dang, Chuang Gan, and Song Han. Awq: Activation-aware weight quantization for on-device llm compression and acceleration. *Proceedings of Machine Learning and Systems*, 6:87–100, 2024.

[23] Weijie Liu, Peng Zhou, Zhe Zhao, Zhiruo Wang, Haotang Deng, and Qi Ju. Fastbert: a self-distilling bert with adaptive inference time. *arXiv preprint arXiv:2004.02178*, 2020.

[24] Xinyu Liu, Houwen Peng, Ningxin Zheng, Yuqing Yang, Han Hu, and Yixuan Yuan. Efficientvit: Memory efficient vision transformer with cascaded group attention. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 14420–14430, 2023.

[25] Alessio Maritan, Subhrakanti Dey, and Luca Schenato. Fedzen: Towards superlinear zeroth-order federated learning via incremental hessian estimation. *arXiv preprint arXiv:2309.17174*, 2023.

[26] Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Aguera y Arcas. Communication-efficient learning of deep networks from decentralized data. In *Artificial intelligence and statistics*, pages 1273–1282. PMLR, 2017.

[27] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

[28] Jonas Pfeiffer, Andreas Rücklé, Clifton Poth, Aishwarya Kamath, Ivan Vulić, Sebastian Ruder, Kyunghyun Cho, and Iryna Gurevych. Adapterhub: A framework for adapting transformers. *arXiv preprint arXiv:2007.07779*, 2020.

[29] Zhen Qin, Daoyuan Chen, Bingchen Qian, Bolin Ding, Yaliang Li, and Shuiguang Deng. Federated full-parameter tuning of billion-sized language models with communication cost under 18 kilobytes. *arXiv preprint arXiv:2312.06353*, 2023.

[30] Jie Ren, Samyam Rajbhandari, Reza Yazdani Aminabadi, Olatunji Ruwase, Shuangyan Yang, Minjia Zhang, Dong Li, and Yuxiong He. {Zero-offload}: Democratizing {billion-scale} model training. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 551–564, 2021.

[31] Xiaoyang Sun, Wei Wang, Shenghao Qiu, Renyu Yang, Songfang Huang, Jie Xu, and Zheng Wang. Stronghold: fast and affordable billion-scale deep learning model training. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–17. IEEE, 2022.

[32] Zhiqing Sun, Hongkun Yu, Xiaodan Song, Renjie Liu, Yiming Yang, and Denny Zhou. Mobilebert: a compact task-agnostic bert for resource-limited devices. *arXiv preprint arXiv:2004.02984*, 2020.

[33] Kahou Tam, Chunlin Tian, Li Li, Haikai Zhao, and ChengZhong Xu. Fedhybrid: Breaking the memory wall of federated learning via hybrid tensor management. In *Proceedings of the 22nd ACM Conference on Embedded Networked Sensor Systems*, pages 394–408, 2024.

[34] Chunlin Tian, Li Li, Zhan Shi, Jun Wang, and ChengZhong Xu. Harmony: Heterogeneity-aware hierarchical management for federated learning system. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 631–645. IEEE, 2022.

[35] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. Llama: Open and efficient foundation language models, 2023.

[36] Laurens Van der Maaten and Geoffrey Hinton. Visualizing data using t-sne. *Journal of machine learning research*, 9(11), 2008.

[37] Kuan Wang, Zhijian Liu, Yujun Lin, Ji Lin, and Song Han. Haq: Hardware-aware automated quantization with mixed precision. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 8612–8620, 2019.

[38] Kun Wang, Jiani Cao, Zimu Zhou, and Zhenjiang Li. Swapnet: Efficient swapping for dnn inference on edge ai devices beyond the memory budget. *IEEE Transactions on Mobile Computing*, 2024.

[39] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, et al. Huggingface's transformers: State-of-the-art natural language processing. *arXiv preprint arXiv:1910.03771*, 2019.

[40] Yebo Wu, Chunlin Tian, Jingguang Li, He Sun, Kahou Tam, Li Li, and Chengzhong Xu. A survey on federated fine-tuning of large language models. *arXiv preprint arXiv:2503.12016*, 2025.

[41] Jingyi Xu, Zihan Chen, Tony QS Quek, and Kai Fong Ernest Chong. Fedcorr: Multi-stage federated learning for label noise correction. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 10184–10193, 2022.

[42] Mengwei Xu, Dongqi Cai, Yaozong Wu, Xiang Li, and Shangguang Wang. Fwdllm: Efficient fedllm using forward gradient. *arXiv. Available at: hjp://arxiv.org/abs/2308.13894 (Accessed: 11 March 2024)*, 2024.

[43] Mengwei Xu, Feng Qian, Qiaozhu Mei, Kang Huang, and Xuanzhe Liu. Deeptype: On-device deep learning for input personalization service with minimal privacy concern. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, 2(4):1–26, 2018.

[44] Jianyi Zhang, Saeed Vahidian, Martin Kuo, Chunyuan Li, Ruiyi Zhang, Tong Yu, Guoyin Wang, and Yiran Chen. Towards building the federatedgpt: Federated instruction tuning. In *ICASSP 2024-2024 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 6915–6919. IEEE, 2024.

Learning-Enhanced High-Throughput Pattern Matching Based on Programmable Data Plane

Guanglin Duan*[1,2], Yucheng Huang*[1,2], Zhengxin Zhang*[1,2,3], Qing Li†[1], Dan Zhao[1], Zili Meng[4], Dirk Kutscher[5], Ruoyu Li[1,6], Yong Jiang[2], and Mingwei Xu[7]

[1]*Peng Cheng Laboratory* [2]*Tsinghua Shenzhen International Graduate School* [3]*Cornell University* [4]*Hong Kong University of Science and Technology* [5]*Hong Kong University of Science and Technology (Guangzhou)* [6]*Shenzhen University* [7]*Tsinghua University*

Abstract

Pattern matching is critical in various network security applications. However, existing pattern matching solutions struggle to maintain high throughput and low cost in the face of growing network traffic and increasingly complex patterns. Besides, managing and updating these systems is labor intensive, requiring expert intervention to adapt to new patterns and threats. In this paper, we propose *Trochilus*, a novel framework that enables high-throughput and accurate pattern matching directly on programmable data planes, making it highly relevant to modern large-scale network systems. Trochilus innovated by combining the learning ability of model inference with the high-throughput and cost-effective advantages of data plane processing. It leverages a *byte-level recurrent neural network (BRNN)* to model complex patterns, preserving expert knowledge while enabling automated updates for sustained accuracy. To address the challenge of limited labeled data, Trochilus proposes a *semi-supervised knowledge distillation (SSKD)* mechanism, converting the BRNN into a lightweight, data-plane-friendly *soft multi-view forest (SMF)*, which can be efficiently deployed as match-action tables. Trochilus minimizes the need for expensive TCAM through a novel entry cluster algorithm, making it scalable to large network environments. Our evaluations show that Trochilus achieves multi-Tbps throughput, supports various pattern sets, and maintains high accuracy through automatic updates.

1 Introduction

Pattern matching is essential to many network applications, such as network intrusion and prevention systems (NIDS/NIPS) [41, 45, 61], web application firewalls [48], network censorship systems [19], and application identification systems [37]. These applications rely on pattern-matching systems to scan packet headers and payloads to check if they match a given set of rules (i.e., patterns or signatures) containing a series of strings and regular expressions.

A recurring theme in pattern-matching systems literature is the gap between the high bandwidth workloads they need to handle and the scalability or cost of existing hardware/-software implementations. Prior pattern-matching systems have attempted to alleviate this gap via algorithm optimization [5, 17, 42, 51, 54, 58, 59, 62] and hardware (GPU/FPGA/NPU) acceleration [22, 24, 35, 44, 49, 50, 64]. For algorithm optimization which is mainly deployed on CPUs, a server is unable to reach more than 70Gbps [52, 53, 58]. While servers optimized with FPGA/GPU/NPU may achieve a higher throughput (i.e., at most 100Gbps under state of the art [64]) thanks to their inherent parallel computation, they significantly increase the complexity in system management and make the updates extremely challenging [11, 32].

The emerging programmable switching ASICs in the network community provide an unprecedented opportunity to bridge this gap. A programmable switch can easily handle multi-Tbps traffic at a line rate with lower cost than GPU/FPGA/NPU/CPU, offering a new possibility for addressing the performance bottlenecks of pattern-matching systems.

Existing works have integrated programmable switches but they still suffer from the issues of accuracy. Multi-string matching [25, 30, 52] employs Nondeterministic Finite-state Automata (NFA) relevant matching algorithms and is constrained by the limited memory available on the programmable data plane, and thus can only support a limited number of string patterns. Besides, multi-string matching can only detect exact and simple patterns (e.g., digital sequence), which consists of a small subset of pattern matching. Regular expression matching is a more powerful tool for matching complex patterns, attributing to regular expression's representative syntaxes (e.g., Kleene star and counting constraint). However, the aforementioned methods can not extend to deploy complete pattern matching on programmable switches. This is because the serialized state transitions of automaton utilized in [25, 30, 52] cause the problem of resource explosion when deploying the complex syntaxes in regular expressions (e.g., range matching "[*a*-*f*]"). Today, we are still faced with the need to build a pattern-matching system that

*Co-first authors. †Corresponding author: liq@pcl.ac.cn.

Table 1: Trochilus v.s. prior arts. Hardware includes GPU, FPGA, and NPU. Switch represents programmable switch.

Platform	Method	Solution	High throughput	Accuracy	Manag ability
Software	Automaton-based (Rule-based)	[51,53,54]	×	✓	×
Hardware		[13,49,64]	✓	✓	×
Switch		[25,52]	✓	×	×
	Model inference-based	Trochilus	✓	✓	✓

can support line rates on the order of 100Gbps [64]. This paper answers this challenge with the Trochilus Programmable Data Plane-based complete pattern matching system.

Moreover, current pattern-matching systems are focused on enhancing performance while managing cost and power consumption, but they do not allow for the efficient updating of pattern sets to ensure sustained accuracy. In practical deployment, pattern-matching systems require not only an initially high-accuracy pattern set but also regular updates to maintain high accuracy, in order to address new traffic patterns (e.g., new cyber attacks and new applications). However, current rules are typically obtained through offline techniques (e.g., crafted by experts or obtained from proprietary vendor algorithms) [64], which is labor-intensive and induces high maintenance costs. Trochilus addresses this challenge by replacing traditional pattern matching with model inference.

In designing Trochilus, we argue that an ideal system for pattern matching should meet three requirements: R1: High Throughput (Capable of achieving high throughput at a low cost). R2: Accuracy (consistently able to accurately identify traffic patterns). R3: Manageability (efficiently updating the system to accommodate new network traffic patterns). Unfortunately, prior pattern matching schemes [13,25,49,51–54,64] fall short on at least one requirement, as shown in Table 1. Tochilus fulfills all three requirements by incorporating the learning abilities of model inference with the low-cost and high-throughput advantages of programmable switches.

To meet the requirements of accuracy and manageability, Trochilus designs a pattern modelization method that can losslessly transform given patterns to a byte-level recurrent neural network (BRNN). BRNN preserves expert knowledge from the patterns, thereby achieving high initial accuracy. Compared with traditional pattern-matching systems, the BRNN enables automatic updating with labeled data for higher accuracy and supports easy manageability over ever-changing traffic patterns. Further, concerning limited or even no labeled data, Trochilus introduces a semi-supervised knowledge distillation (SSKD) mechanism to transform the BRNN into a lightweight yet accurate model, called the soft multi-view forest (SMF), whose inference process only requires simple operations supported by the switches' match-action paradigm. By training multiple biased decision trees, the SMF can better represent complex traffic patterns and achieve higher accuracy compared to other tree-based models [55,65,66].

To achieve high throughput with programmable switches, Trochilus develops a model representation to efficiently deploy SMFs on the data plane. Trochilus converts SMFs to match-action tables through tree encoding. Specifically, we formulate an optimization problem that aims at finding a partition of table entries to minimize the TCAM requirement. To this end, we propose an entry cluster algorithm, which heuristically aggregates table entries into clusters based on the similarity among features represented by table entries so that each cluster only requires a more compact table. Moreover, a sliding window mechanism is introduced to effectively inspect the incoming data packet payload.

We evaluate Trochilus extensively with various pattern sets (Snort [41] and Suricata [45]) and workloads (captured in one of the largest public cloud providers). Evaluations show that Trochilus effectively preserves the accuracy of original patterns without labeled data and maintains even higher accuracy through training with labeled data. Moreover, Trochilus achieves multi-Tbps throughput and supports various pattern sets with limited resources.

Contributions. 1) We design a novel framework that integrates model inference with low-cost high-throughput data plane processing to address the pressing challenges of scalability, accuracy, and manageability faced by pattern matching systems. 2) We design pattern modelization, SSKD, and entry cluster optimization to convert patterns into accurate and lightweight models and deploy them on resource-constrained programmable data planes. 3) We implement a prototype of Trochilus using the Tofino switch as the programmable data plane and make its source code available [3]. We conduct testbed experiments to show its performance superiority.

2 Background and Motivation

In this section, we introduce the background of pattern matching, propose the design requirements of pattern matching, and highlight the limitations of the prior arts in this field.

2.1 Pattern Matching

The key goal of a pattern-matching system is to identify when network traffic triggers any of up to tens of thousands of signatures, also known as rules. In network applications (e.g., network intrusion and prevention systems), patterns typically include strings for exact matching and regular expressions for complex patterns. Formally, given an alphabet σ, an input string $x = x_0, \cdots, x_n$ and a set of patterns $P = \{p_1, \cdots, p_k\}$, where each element $p_i = r_1 \cdots r_m$ is a predefined pattern and each x_i or r_i is a character belonging to σ, pattern matching can check whether each p_i is a substring of x. Since Finite-state Automata (FA) provides a formal and flexible framework for representing and processing regular languages, converting patterns to Nondeterministic Finite-state Automata (NFA)

and deterministic finite-state automata (DFA) is a common method for practical pattern matching [52].

2.2 Design Requirements

Requirement 1 (R1): High Throughput. Recently, the network bandwidths at traffic aggregation points in regional ISPs have already surpassed multi-hundred Gbps. Many network device providers [46] and standard organizations [2] are moving towards the era of 800 Gbps bandwidth. An ideal pattern-matching system should keep up with the current network bandwidth requirements. Moreover, the financial cost of the system must be low enough to be affordable for the majority of companies.

Requirement 2 (R2): Accuracy. An ideal pattern-matching system should have high initial accuracy upon deployment and consistently maintain accurate identification of network traffic patterns during operation.

Requirement 3 (R3): Manageability. In practical deployment, pattern matching systems need to continuously update their pattern sets to identify new traffic patterns (e.g., new applications and new cyber attacks). However, updating patterns requires experts to analyze large-scale network traffic and extract representative patterns, which is labor-intensive. An ideal pattern-matching system should be manageable and be able to update and maintain patterns with low overhead.

2.3 Limitations of Prior Studies

Many efforts have been made to accelerate pattern matching, including software-based algorithmic optimization, hardware acceleration, and programmable switch acceleration.

Software-based algorithmic optimization. Algorithmic optimization boosts throughput by reducing memory consumption [5, 10, 59] or increasing the number of characters per state transition in NFA [42, 51, 54, 62]. However, since servers' CPUs are not designed specifically for high-throughput packet processing, the performance of software-based packet processing is still constrained. Actually, a server-based pattern matching engine still can not achieve 70 Gbps packet processing throughput [53, 54, 58]. Although scaling up the number of servers can achieve higher throughput, it also induces high capital and management costs [15]. To sum up, software-based algorithmic optimization methods do not satisfy R1.

Hardware acceleration. Some works leverage GPUs for pattern matching as the single instruction multiple threads (SIMT) architecture can effectively parallelize an algorithm, providing multi-10 Gbps throughput [24, 49, 50, 60]. However, it is difficult to maintain GPU's performance at the peak rate as it requires all computational units to have the same instruction stream [34]. Additionally, GPUs also come with higher power costs. FPGA-based solutions utilize circuit-level parallelism to accelerate pattern matching. While these methods improve the overall throughput of pattern matching [13, 35, 44, 64], they do not support dynamic updates well due to the FPGA programming process. Such inflexibility increases the investment in capital and research and development efforts. Furthermore, the performance of these hardware alternatives still struggles to keep pace with the rapidly increasing network traffic since these hardware alternatives are typically connected to servers via PCIe which provides limited bandwidth [44]. To sum up, the traditional hardware acceleration does not satisfy R1.

Traditional Programmable switch acceleration. Programmable switches, centering around Protocol-Independent Switch Architecture (PISA) [7], provide flexible packet processing capabilities at high throughput, presenting opportunities for network functions offloading [26, 32, 33, 56]. They exhibit similar power consumption and capital costs as traditional fixed-function switches, significantly reducing costs compared to CPUs or other hardware. However, PISA faces computation and storage constraints. Complex instructions such as multiplications and divisions are not allowed. PISA has limited match action unit (MAU) stages (e.g., Tofino 1 has 12 stages) to implement the main calculation logic. In addition, storage is limited, e.g., on Tofino 1, the SRAM of each pipeline is 120MB and the TCAM is 6.2MB. Previous works utilize programmable switches to accelerate multi-string matching [25, 52]. They mainly convert the multi-string matching patterns into NFAs and then perform optimization on state encoding and state transition. However, prior works perform sequential state transitions, where the number of MAU stages and TCAM requirements increase linearly as the average length and the total number of multi-string patterns grow. As a result, they are unable to achieve a complete deployment toward large-scale multi-string patterns. Moreover, prior works can not be extended to more general and complex pattern matching. To sum up, traditional programmable switch solutions do not satisfy R1 and R2.

Common limitations of prior studies. The deployed patterns can not always remain effective over ever-changing traffic patterns. However, prior arts primarily focus on the acceleration of pattern matching and do not provide efficient mechanisms for pattern update. Therefore, existing pattern-matching systems can not meet R3.

3 Trochilus Overview

3.1 Target Scenarios

Trochilus acts as a functional instance of network applications, inspecting payloads at the byte level and taking corresponding actions (such as alerting, forwarding, and dropping). Via network data planes, Trochilus can be deployed as a link middlebox or as a dedicated application for traffic analysis, such as redirecting traffic to Trochilus. The current mainstream data plane includes programmable switches and SmartNICs [36].

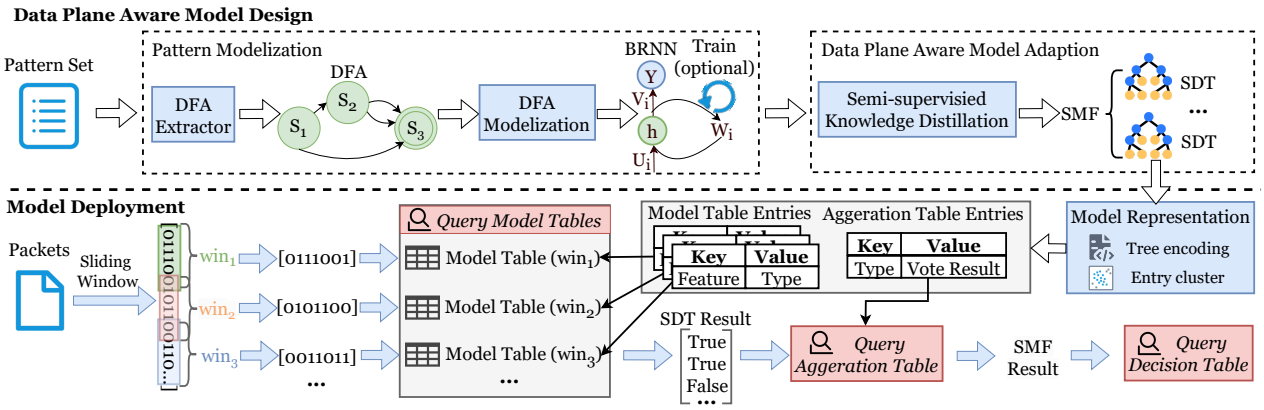


Figure 1: Trochilus framework.

These two types of hardware have drastically different characteristics: programmable switches have much higher throughput, with much limited computation capability. In this paper, we focus on PISA-driven pattern matching design, which can be integrated into other data plane hardware (e.g., SmartNICs) with minor changes. Note that when the depth of the payload exceeds the single pipeline processing capability of the programmable switch, Trochilus will recirculate the packets and discard the inspected parts of the packets.

In this paper, we assume that a processed packet contains an Ethernet header, an IP header, a UDP or TCP header, and a payload to be inspected. Note that our approach can support other header and payload fields (e.g., ICMP). The packet payload can be encoded in any form, such as ASCII and UTF-8. In this paper, we adopt ASCII encoding, where each character is encoded as one byte. Our approach is devoted to scenarios where the payload only contains plain text that is not encrypted. For processing encrypted traffic, Trochilus can be combined with previously proposed decryption mechanisms [43] to handle encrypted traffic.

3.2 Trochilus Workflow

As illustrated in Figure 1, Trochilus consists of two components: *data plane aware model design* and *model deployment*.

Data plane aware model design combines *pattern modelization* and *data plane aware model adaption* to generate data plane-friendly learning models from patterns, enabling Trochilus to replace traditional pattern matching with model inference. Given patterns extracted from the signature database (e.g. Snort community rules [41]), Trochilus first utilizes the *DFA extractor* to convert the patterns to a DFA. Second, the *DFA modelization* transforms the DFA into a BRNN, which can effectively preserve the patterns' expert knowledge and achieves high initial accuracy. When labeled data is available, the BRNN can automatically capture underlying patterns in the data through training to improve accuracy. Although the BRNN is effective, it cannot be deployed in the data plane. To generate a data plane aware model, Trochilus then leverages semi-supervised knowledge distillation (SSKD) to distill a soft multi-view forest (SMF), which consists of multiple soft decision trees (SDTs), under the instruction of the BRNN.

The key component of *model deployment* is *model representation*, which targets efficiently translating an SMF into multiple model tables and one aggregation table through *tree encoding*. Each model table is encoded from an SDT in the SMF, and the aggregation table stores the SMF inference results voted by each model table. When deploying model tables, Trochilus designs an efficient *entry cluster* algorithm to greatly reduce the TCAM requirements.

For incoming packets, Trochilus first utilizes a sliding window mechanism to extract payload segments for different windows as features. Then, these features are used to query model tables to obtain inference results of SDTs. The SMF result is voted by inference results of SDTs through querying the aggregation table. Finally, based on the SMF result, users can design post-processing actions encoded in the decision table, such as dropping, passing, alerting, or forwarding.

The conversion of the SMF model into data-plane table entries is lossless. Trochilus supports online updates of the pattern matching system by collecting new labeled attack traffic, incrementally training the model offline, and converting the updated rules into table entries. These entries can then be installed on the switch without interrupting ongoing services. This allows Trochilus to adapt to new attack patterns with minimal impact on system operations.

4 Data Plane Aware Model Design

In this section, we show how we convert patterns to byte-level recurrent neural networks (BRNNs) and how we further transform the BRNNs to soft multi-view forests (SMFs) with semi-supervised knowledge distillation (SSKD).

4.1 Pattern Modelization

In this subsection, we utilize the byte-based DFA extractor to construct a DFA from patterns. Then we leverage the DFA modelization to convert the DFA into a BRNN as the teacher model in SSKD. Compared to other existing teacher models, BRNN is the most suitable for pattern matching. Specifically, compared to DFAs, when labeled data is available, the BRNN can capture new undefined patterns in the data through training, leading to an improvement in accuracy and boosting the performance of the following knowledge distillation. Compared to other learning models (e.g., LSTM [20], GRU [12]), the BRNN preserves the provided patterns' expert knowledge and retains a high accuracy under cold start.

Byte-based DFA extractor. Network security applications, e.g., Suricata [45] and Snort [41], usually use Perl-compatible regular expression (PCRE) [39] syntax to define patterns. We also use the PCRE syntax as the standard syntax, which can easily be extended to other syntax flavors (e.g., POSIX [16]). The supported PCRE syntax is shown in Appendix A.

Consider an example pattern in Snort "$\backslash x26cvv\backslash x3d[0\text{-}2]\{3,5\}$". Since this pattern can appear anywhere, we first convert the pattern to "$.*\backslash x26cvv\backslash x3d[0\text{-}2]\{3,5\}.*$". Here, '.' is the wildcard that can match any character. '$*$' is the Kleene star operator to match the preceding subexpression zero or more times. Braces "$\{a,b\}$" are the range matching to match the preceding subexpression a to b times. To make this pattern consistent with the packet payload format (i.e., bytes), we convert each character in the pattern except the special operators in PCRE (e.g. '$*$') into its corresponding one-byte ASCII code. So the final converted pattern is "$.*$ $0x26$ $0x63$ $0x76$ $0x76$ $0x3d$ $[0\text{-}2]\{3,5\}$ $.*$".

Next, we construct the DFA for the converted pattern. We use Thompson's construction algorithm [47] to transform a pattern into an equivalent NFA. Then we convert the NFA to a DFA with the powerset construction algorithm [40]. We minimize the number of states of the DFA by the DFA minimization algorithm [21]. Formally, a DFA consists of a 5-tuple $\mathcal{A} = (\Sigma, \mathcal{G}, \mathcal{W}, \alpha_0, \alpha_\infty)$, whose elements are defined as follows. Σ: the input vocabulary, whose size is $V = 2^8 = 256$; $\mathcal{G}$: a finite set of states, where $|\mathcal{G}| = K$; $\mathcal{W} \in \mathbb{R}^{V\times K\times K}$: transition weights; $\mathcal{W}[\sigma, g_i, g_j]$: the weight of transferring g_i to g_j according to the input σ; α_0: initial weight; $\alpha_\infty[i]$: the final weight of g_i after reading the whole input; α_∞: final weights; $\alpha_\infty[i]$: the final weight of g_i after reading the whole input. Consider an input sequence $\mathcal{X} = \{x_1, x_2, ..., x_N\}$ and a path $\mathcal{P} = \{u_1, u_2, ..., u_{N+1}\}$, where u_i is the index of state concerning x_i. The score $\mathcal{B}(\mathcal{A}, \mathcal{P})$ of path $\mathcal{P}$ is defined as:

$$\mathcal{B}(\mathcal{A}, \mathcal{P}) = \alpha_0[u_1] \cdot \left(\prod_{i=1}^{N} \mathcal{W}[x_i, u_i, u_{i+1}] \right) \cdot \alpha_\infty[u_{N+1}]. \quad (1)$$

Let $\pi(\mathcal{X})$ be the set of all paths starting from any state in $\mathcal{G}_0$ and ending at any state in $\mathcal{G}_\infty$, where $\mathcal{G}_0$ is the set of start states and $\mathcal{G}_\infty$ is the set of final states. The sum of path scores, $\mathcal{B}_{fw}(\mathcal{A}, \mathcal{X})$, can be computed by the Forward algorithm [4]:

$$\mathcal{B}_{\text{fw}}(\mathcal{A}, \mathcal{X}) = \sum_{\mathcal{P}\in\pi(x)} \mathcal{B}(\mathcal{A}, \mathcal{P}) = \alpha_0^T \cdot \left(\prod_{i=1}^{N} \mathcal{W}[x_i] \right) \cdot \alpha_\infty. \quad (2)$$

DFA modelization. The DFA is parameterized by Θ, which is defined as $< \mathcal{W}, \alpha_0, \alpha_\infty >$. Let $h_t \in \mathbb{R}^K$ be the forward score vector after considering the first t words $\{x_1, x_2, ..., x_t\}$ of $\mathcal{X}$. We can rewrite the forward score into a recurrent form:

$$h_0 = \alpha_0^T, \quad (3)$$

$$h_t = h_{t-1} \cdot \mathcal{W}[x_t], 1 \le t \le N, \quad (4)$$

$$\mathcal{B}_{\text{fw}}(\mathcal{A}, \mathcal{X}) = h_N \cdot \alpha_\infty. \quad (5)$$

Here, we treat matching on a pattern as a binary classification task. To this end, we expand $\mathcal{B}_{\text{fw}}(\mathcal{A}, \mathcal{X})$ to a vector $[1, 0]$ when $\mathcal{B}_{\text{fw}}(\mathcal{A}, \mathcal{X})$ is 0, and $[0, 1]$ otherwise. It can be easily generalized to multi-classification tasks as well. Recall that the hidden state computation in the forward propagation of a recurrent neural network (RNN) is given by $h_t = \theta(Ux^t + Wh^{t-1} + b)$, where θ denotes the activation function. When $U = 0, b = 0, W = \mathcal{W}$, and θ is the identify function, the forward score computation of the DFA (as shown in Equation 4) becomes analogous to the forward propagation process of the RNN. On this basis, we convert the DFA with byte-format transitions into an RNN with Θ, called BRNN. In practice, fine-grained classification may be desired, which considers the different combinations of final states, i.e., different terminal state combinations correspond to different output categories. In this case, we can use an MLP after BRNN as the aggregation layer. The BRNN can retain the accuracy of the original patterns and be put into production immediately without waiting for data collection. When enough labeled data is collected, the performance of the BRNN can be further improved through training.

4.2 Data Plane Aware Model Adaption

The BRNN obtained in the previous step is difficult to directly deploy on the data plane, since it involves float and nonlinear operations, and requires huge storage resources. To promote easy deployment on the data plane, we further convert BRNN, an unwieldy teacher model, into a lightweight ensemble model called SMF using SSKD.

Existing knowledge distillation approaches face two major problems when applied in network scenarios. First, labeled data only accounts for a very small proportion. It is challenging to distill a student model with high accuracy in such data-scarce scenarios. Second, due to the limited computational and memory resources on programmable switches, there are stringent restrictions on the selection of student models. Specifically, prior studies mostly deploy tree-based models (e.g., decision trees) [55,57,65]. However, the generalization of the decision tree is too limited to represent complex

traffic patterns. Training decision trees on large-scale data are prone to overfitting, resulting in poor classification accuracy.

Algorithm 1: Data Plane Aware Adaption.

Input: Teacher model $BRNN$, Labeled data D_l, Unlabeled data D_u, Tree set T_{set}

1 $\mathcal{Y}_l^{soft}, \mathcal{Y}_u^{soft}, \mathcal{Y}_u^{hard} \leftarrow BRNN(D_l, D_u)$;
2 $\mathcal{Y}_l^{mix} \leftarrow \beta \cdot \mathcal{Y}_l^{hard} + (1-\beta) \cdot \mathcal{Y}_l^{soft}$;
3 $\mathcal{Y}_u^{mix} \leftarrow \beta \cdot \mathcal{Y}_u^{hard} + (1-\beta) \cdot \mathcal{Y}_u^{soft}$;
4 $\mathcal{Y}^{mix} \leftarrow \mathcal{Y}_l^{mix} \cup \mathcal{Y}_u^{mix}$;
5 $w \leftarrow [1] * (D_l.size + D_u.size)$;
6 $\mathcal{X}_{filtered}, \mathcal{Y}_{filtered}^{mix} \leftarrow \mathcal{X}, \mathcal{Y}^{mix}$;
7 **for** m **in** $range(n_t)$ **do**
8 $\quad T_m \leftarrow TrainSDT(\mathcal{X}_{filtered}, \mathcal{Y}_{filtered}^{mix}, w)$;
9 $\quad T_{set} \leftarrow T_m \cup T_{set}$;
10 $\quad error \leftarrow TestSDT(\mathcal{X}, \mathcal{Y}^{mix}, T_m)$;
11 $\quad w \leftarrow UpdateWeight(error, w)$;
12 $\quad \mathcal{X}_{filtered}, \mathcal{Y}_{filtered}^{mix} \leftarrow SampleFilter(\mathcal{X}, \mathcal{Y}^{mix}, w)$;
13 **end for**
14 **return** T_{set}

In Trochilus, we propose SSKD and SMF, which overcome the above-mentioned problems with two key ideas. First, we introduce the semi-supervised learning strategy into the knowledge distillation. Attributing to the cold start characteristic, the BRNN can effectively preserve the expert knowledge from patterns and hence offers adequate accuracy without training data. Moreover, when enough labeled data is collected, BRNNs' performance can be further improved. As a result, we jointly utilize the ground truth of labeled data and inference results of BRNN on both labeled and unlabeled data to train the student model. Second, we propose SMF as the student model instead of the decision tree (DT) used in the previous studies [55]. SMF is an ensemble model which is composed of multiple soft decision trees (SDT). Each SDT is trained in a weight-based iterative manner and is applied with binary features to adapt to programmable switches. The submodels (SDTs) of the ensemble model (SMF) are trained with samples with different weights to achieve multi-view observation and learn the inherent patterns.

SSKD aims to generate soft labels that contain knowledge of the teacher model and applies the soft labels to train the student model. We define the labeled data as $D_l = \{(\mathcal{X}_1, \mathcal{Y}_1), \ldots, (\mathcal{X}_n, \mathcal{Y}_n)\}$ and unlabeled data as $D_u = \{\mathcal{X}_{n+1}, \mathcal{X}_{n+2}, \ldots, \mathcal{X}_{n+m}\}$, where each $\mathcal{X}$ is composed of the bit sequences extracted from the packet payload (binary feature).

The training phase of SSKD includes three steps as shown in Algorithm 1. First, we acquire the output logits of BRNN (line 1 in Algorithm 1), which are the output values of the Softmax layer, for both labeled and unlabeled data, and define them as the soft labels ($\mathcal{Y}^{soft}$). Second, we calculate the mixed label $\mathcal{Y}^{mix} = \beta \cdot \mathcal{Y}^{hard} + (1-\beta) \cdot \mathcal{Y}^{soft}$ of a sample, where β is a hyperparameter ranging from 0 to 1, $\mathcal{Y}^{hard}$ is the hard label of the sample (line 2-4 in Algorithm 1). For labeled data, the hard label is its ground truth; for unlabeled data, the hard label is the output label of BRNN. Third, we use all data obtained in the second step to train the SMF (line 5-13 in Algorithm 1). We define the number of iterations (number of trees in SMF) as n_t. The SMF training process includes four steps, as illustrated in Algorithm 1.

(1) We initialize each sample weight w to 1 so that all samples have the same significance at the beginning (line 5-6 in Algorithm 1).

(2) We train an SDT referred to as T_m using the weighted samples for m-th iteration rounds (line 8-9 in Algorithm 1). w affects the training process of SDT including the calculation of the *Gini* index, node partition, and the label selection of leaf nodes. The training process of *TrainSDT* is similar to that of the CART decision tree [9], which utilizes purity to split nodes. The purity after splitting the node via feature A is

$$P(D,A) = \frac{|D_l^A|}{|D|} Gini(D_l^A) + \frac{|D_r^A|}{|D|} Gini(D_r^A), \tag{6}$$

where D is the sample set of the parent node, and D_l^A and D_r^A are the sample sets of the left and right child split by feature A, respectively. The feature with the smallest $P(D,A)$ will be used for node splitting. Different from the calculation of the *Gini* index for hard labels, for the sample set D that contains mixed labels, we calculate its *Gini* index $Gini(D)$ as

$$Gini(D) = 1 - \sum_{y^i \in \mathcal{Y}} \left(\frac{\sum_{(\mathcal{X},\mathcal{Y}) \in D} w \cdot y^i}{|D|} \right)^2, \tag{7}$$

where $\mathcal{Y} = (y^0, y^1, ..., y^C)$, y^i represents the mixed label of type i, and $\mathcal{N}_C$ is the number of classes. The end condition of the SDT training process is consistent with the CART.

(3) In lines 10-11, we test the T_m trained in step (2) on all training sets and update w according to

$$w_i = \frac{w_i^{former}}{Z} exp(-\log \frac{1-error}{error} \cdot \mathcal{Y}_i^{hard} \cdot T_m(\mathcal{X}_i)), \tag{8}$$

where w_i is the weight of sample i, $error = 1 - accuracy$, and Z is the normalization coefficient to make the sum of weight constant. Intuitively, the weights for misclassified samples will be increased.

(4) We filter out samples with weights smaller than tw to reduce the complexity of a single SDT (line 12 in Algorithm 1). The above training is conducted for n_t iteration rounds.

5 Model Deployment

Trochilus deploys trained models onto the resource-limited data plane through model representation and efficiently identifies target patterns using the sliding window algorithm.

Algorithm 2: Entry Cluster.

Input: Entry set E, Maximum table number $\mathcal{T}$, Maximum iterations l, Subset number k

1 Randomly init the center set $c^{(1)} = \{c_1^{(1)}, ..., c_k^{(1)}\}$;
2 Init the target entry set $S = \{S_1, ..., S_k\}, S_i = \emptyset$; iteration round $h = 1$;
3 **while** $h < l$ & $c^{(h)} \neq c^{(h-1)}$ **do**
4 **for** e_i **in** E **do**
5 **for** $c_j^{(h)}$ **in** $c^{(h)}$ **do**
6 $d_j \leftarrow Jaccard(e_i, c_j^{(h)})$;
7 **end for**
8 Assign e_i to the cluster with smallest d;
9 **end for**
10 **for** $c_i^{(h)}$ **in** $c^{(h)}$ **do**
11 Update $c_i^{(h)}$ for S_i;
12 **end for**
13 $h \leftarrow h + 1$
14 **end while**
15 **return** $S^{(h)}$

5.1 Data Plane Model Representation

We present how to represent SMFs in the data plane. Our SMF consists of multiple SDTs. We start with the data plane representation of a single SDT. Prior studies often adopt decimal features to train tree-based models [57, 65, 66]. When converting the rules of these trained models into table entries on the data plane, we need to perform range matching for each feature. Since range matching is not supported by all programmable switches, it has to be expanded into multiple ternary matching using the traditional prefix method [31]. However, when concatenating multiple feature ranges within a classification rule, the number of extensions is multiplied, leading to the problem of entry explosion [66].

Tree encoding. Trochilus solves the above entry combinatorial explosion problem by introducing binary features [55]. For an SDT with binary features, a sequential decision process classifies a given example by selecting a path from the root node to a leaf node. Starting from the root node, a branch is selected by checking the value (0/1) of the current node's bit feature. The tree repeats this process until reaching a leaf node and then returns the classification result. Each path corresponds to a classification rule, and for each rule, we only need to check the bit values of different features. Trochilus converts an SDT to a model table that conducts ternary matching with the concatenation of binary features and extracts table entries from the classification rules of the SDT.

Formally, given the input feature vector $\mathcal{X} = x_0 \cdots x_n$, where $x_i \in \{0,1\}$, the entry set extracted from the SDT model is denoted as $E = \{e_1, e_2, \cdots\}$, where $e_i = (M_i, V_i)$ is a ternary matching rule, with $M_i = m_0 \cdots m_n, m_i \in \{0,1\}$ being the ternary mask, and $V_i = v_0 \cdots v_n, v_i \in \{0,1\}$ being the ternary value. If the condition of $\mathcal{X} = M_i \; AND \; V_i$ is satisfied, a hit in e_i is indicated and the corresponding action will be executed. This implies that for each e_i, we only need to consider the indices $pos(e_i) = \{p | M_{ip} = 1\}$, and $\forall j \in pos(e_i), x_j = v_{ij}$. For example, let $\mathcal{X} = x_0 x_1 \cdots x_7$ denote an input sequence, which is an 8-bit feature. The path to leaf node 5 in Figure 2 is: If $x_1 = 1$ *and* $x_4 = 1$ *and* $x_8 = 0$; Then $class \leftarrow 5$. As feature values in positions 1, 4, and 8 need to be considered, the ternary masks M is $0b10010001$ and the corresponding ternary value V is $0b10010000$. By employing wild card matching denoted by $*$, we can specify this entry as "$1**1***0$".

Model table resource minimization problem. In model deployment, we observe that most table entries converted from SDTs have redundant information. Based on our statistics, over 90% of the binary data in the masks are zero, indicating that most entries do not require the full matching width of n. Therefore, we can further optimize our algorithm by splitting the original model table that requires the matching of all features into k sub-tables, so that each table only has to match a subset of the features, thereby reducing the overall TCAM usage. We formalize this process as a model table resource minimization problem as follows:

$$\min_{\{S_1,...,S_k\}} \sum_{i=1}^{k} \left| \bigcup_{e_i \in S_i} pos(e_i) \right| \cdot |S_i| \quad s.t. \begin{cases} C_1 : \bigcup_{i=1,...,k} S_i = E, \\ C_2 : S_j \cap S_j = \emptyset, \forall 0 \leq i, j \leq k, i \neq j, \\ C_3 : k \leq \mathcal{T}. \end{cases} \tag{9}$$

Object and Constraints: The optimization goal is to find a k-subset partition of E that minimizes the total TCAM requirement. The total TCAM requirement of each subset can be calculated as the product of the required match length and the number of entries in the subset. There are three constraints. C_1 ensures that the union of all subsets S_i is E. C_2 ensures that there is no overlap between any two subsets S_i and S_j. C_3 ensures that the number of subset k does not exceed the allowed maximum number of tables $\mathcal{T}$, as each subset requires a match-action table.

Entry cluster. The task of partitioning N table entries into k subsets to minimize the overall TCAM requirement is computationally NP-hard. We design a highly efficient heuristic algorithm to solve the aforementioned problem. As depicted in Algorithm 2, we first randomly select k entries from E, each as a cluster center, initialize k clusters of entries as empty sets, and set iteration round h to 1 (line 1-2). Entry cluster proceeds by alternating between two steps.

(1) In the assignment step (line 4-9 in Algorithm 2), we apply the Jaccard distance to measure the dissimilarity between each e_i and each center c_j. Afterwards, the entry e_i is assigned to the closest center in Jaccard distance.

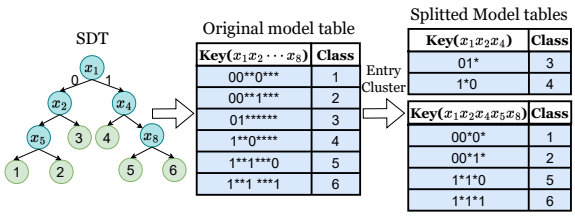


Figure 2: Entry cluster.

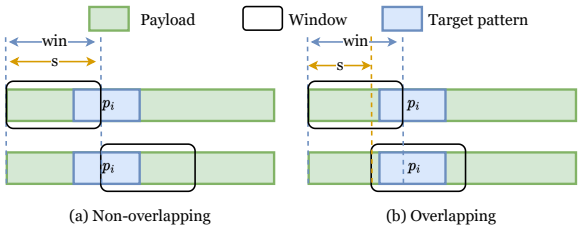


Figure 3: Sliding window.

(2) In the update step, the cluster center is reassigned by summing the distances from each element in the cluster to other elements and selecting the element with the minimum sum (line 10-11 in Algorithm 2).

The algorithm terminates when the entry assignment no longer changes or the maximum iteration number is reached.

Complexity analysis: Let z denote the number of entries, k the number of clusters, n the average feature size for computing Jaccard distance, and l the maximum number of iterations. Each iteration requires computing Jaccard distances between n entries and k centers, resulting in a time complexity of $O(zkn)$. The total worst-case complexity is therefore $O(lzkn)$. In the best case, the algorithm converges after a single iteration, with complexity $O(zkn)$. In the worst case, the Entry Cluster algorithm performs no merging, resulting in the same resource usage as the original SMF. In the best case, all rules are merged into a single wildcard entry, reducing storage usage by a factor of $(z-1)/z$.

Taking the SDT in Figure 2 as an example, the original model table encodes from the SDT takes an 8-bit vector as the ternary match key and has six entries, thus requiring $8*6=48$ bits TCAM at least. Through the entry cluster algorithm, it can be partitioned into two new tables. These two tables respectively utilize a 3-bit and a 5-bit vector as the match key, requiring a total of $3*2+5*4=26$ bits TCAM.

Inference aggregation. We aggregate the results of each SDT through a voting mechanism. We append an aggregation table behind the model tables to match the concatenation of the inference classes of each SDT and obtain the majority of individual inference classes. Since the number of SDTs and inference classes is small, the aggregation table will not induce the problem of excessive entries caused by the combination.

5.2 Sliding Window

The length of the input sequence of each SDT must be limited within the maximum matching length supported by the ternary matching table (e.g., 66 bytes for the Tofino 1). However, the maximum packet length is typically 1500 bytes, which means a packet payload may have to be matched across multiple windows. A naive solution can utilize multiple non-overlapping windows to inspect continuous segments of the payload. However, a target pattern p_i (described in Section 2.1) may occur at arbitrary positions in the packet payload, as shown in Figure 3(a). In this scenario, p_i may span across windows, resulting in a failure to detect p_i in any window.

To address this issue, we propose a sliding window mechanism. The window size is denoted as win, and the maximum length of a pattern is denoted as $\mathcal{L}$. As depicted in Figure 3(b), each time we move the window for s bytes, and hence retain a $win-s$ overlap with the previous window. To ensure that any p_i can be matched, $\mathcal{L} \leq win-s$ must be satisfied.

The maximum payload length that can be inspected by a single pipeline with the sliding window is $win+MS\cdot s$, where MS is the maximum number of MAU stages that can be utilized. Different win and s will influence the depth that can be inspected in a single pipeline and subsequently affect the overall throughput, as recirculation is required for data packets that have not been matched completely. As will be shown in Section 6.6, the sliding window mechanism can significantly enhance the accuracy of pattern matching. This mechanism introduces some additional inspection of packet segments, which can increase resource overhead and affect throughput. Network administrators can deploy Trochilus with carefully chosen window sizes based on the available resources in the target scenarios to achieve the desired trade-off between accuracy, resource overhead, and throughput.

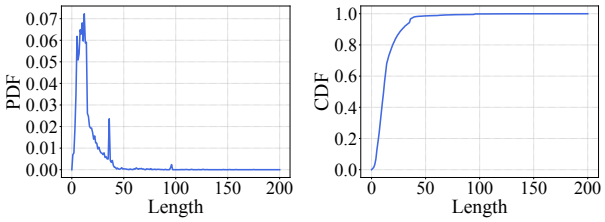
6 Evaluation

We conducted extensive experiments to demonstrate that:

- Trochilus can achieve high initial accuracy comparable to existing pattern-matching systems and can further improve accuracy by around 10% through training (R1).
- Trochilus can achieve Tbps-level throughput for pattern matching while consuming minimal data plane storage resources. (R2).
- Trochilus can automatically update its rules, thereby consistently maintaining high pattern-matching accuracy in the face of changing traffic patterns (R3).

6.1 Dataset and Pattern Set

The pattern sets used are constructed from Snort 2.9.7.0 and Suricata 5.0 provided by ET-OPEN [28]. We collected packet-level traces at gateways of a large data center, which belongs



(a) PDF (b) CDF

Figure 4: Matched packet segments lengths statistics.

to one of the largest public cloud providers and serves tens of Tbps traffic for customers with diverse cloud access requirements. The collected traces are labeled by the advanced attack detection system and application identification system deployed in the data center. We use 10 minutes of traffic traces collected in different time periods of a week. The collected traffic traces mainly cover five categories of the pattern sets, including *info*, *web specific app*, *malware*, *sql*, and *exploit*, containing around 4000 patterns. The total size of collected traffic traces is 534GB. We divide them into the training, test sets with a ratio of 7 : 3. We use the patterns to match the collected traces. The PDF and CDF of matched packet segment lengths are illustrated in Figure 4. The patterns with a length of fewer than 50 accounts for more than 95%.

Ethical Considerations. All data analysis was approved by our cooperation units. We did not investigate human behavior, surface or investigate any individual flows or IP addresses, or store any traffic or individual records to disk. To protect user privacy, all packets in the collected traces are anonymized. We restricted all analysis to aggregate network statistics directly output by Trochilus.

6.2 Baselines and Metrics

Baselines. To evaluate Trochilus's accuracy, throughput, and overhead, we conducted experiments under both, full pattern matching and multi-string matching scenarios. We compared Trochilus with several solutions:

- Traditional Pattern Matching System (TPS): It represents the system implemented by Snort [41] or Suricata [45] based on the given pattern set.
- To evaluate the effectiveness of pattern modelization, in addition to TPS, we also compare BRNN with traditional RNNs, including LSTM [20] and GRU [12], as well as a 4-layer CNN [27] and DAN [23].
- For data plane aware model adaption, we compare our SMF with decision tree (DT) [9], random forest (RF) [8], multi-view forest (MF), soft decision tree (SDT), and soft random forest (SRF). Note that SDT and SRF are the results of distilled DT and RF via SSKD.
- BOLT [52]: As far as we know, no solution currently can support full pattern matching in the data plane. Only BOLT can effectively implement multi-string matching in the data plane. We extract multi-string matching rules from the pattern sets to implement BOLT. To maximize the throughput of BOLT, we set the number of bytes consumed per state transition to 5 and the number of MAU stages used to 12.
- T-MSM-*i*: a trimmed version of Trochilus that only implements multi-string matching rules from the pattern set. Trochilus-*i*: a full version of Trochilus that implements full pattern matching. Here, *i* denotes the number of MAU stages used on the programmable switch.

Metrics. *Accuracy* is the weighted ratio of correctly classified packets in all packets. The weight of each category is the product of the other categories' packet numbers, addressing the imbalance of collected traces. # *table entries* is the number of table entries. *TCAM requirement* is the product of the table entry number and the width of the matching key.

6.3 Experimental Setup

Implementation. We develop a Trochilus prototype which includes about 2000 lines of $P4_{16}$ [6] code for the data plane and 4000 lines of Python code for the control plane. In the data plane, to implement the sliding window mechanism, we replicate both the model and the aggregation table for each window. For an incoming packet, the switch first extracts the payload segments for different windows into customized headers. Afterward, each model table takes different payload segments as the match field, and its action is to obtain the classification result of an SDT. In the next stage, the aggregation table matches the results of each model table and flags the packet if some patterns get matched. In the control plane, we use *automata_tools* library [29] to convert patterns to DFAs, *Pytorch* library [38] to implement BRNN, and Python's *Numpy* library [18] to implement SSKD and SMF.

Resource breakdown. We deployed Trochilus and BOLT to a 12-stage 6.4Tb/s EdgeCore wedge 100BF-65X Tofino switch with a local CPU of Intel (R) Xeon (R) Gold 5218 CPU @ 2.30GHz. Two Dell R230 servers (Intel XeonE2620 v4, 64GB RAM, 40Gbps Intel Network Interface Card) are used for data transmission. We install the DPDK Pktgen [1] on each server to achieve high-performance traffic replay. We run Snort 3 and Suricata 7 software experiments on a Dell R230 server. For pattern modelization, we set the learning rate to 10^{-4}, the batch size to 500, and the hidden state size to 100 for each model. We use the cross-entropy loss as the objective function. We train each model for 200 epochs and use early stopping to avoid overfitting. For data plane aware model adaption, we set β to 0.5 in SSKD. To achieve a balance between model complexity and accuracy, we set n_t to 5, and $thre_w$ to 0.5 in SMF. In each SDT we set *minimum samples* to 15, which is the minimum number of samples required to be at a leaf node. For model deployment, we set $k = 5$ in the entry cluster algorithm, and $s = 30, w = 64$ in the sliding

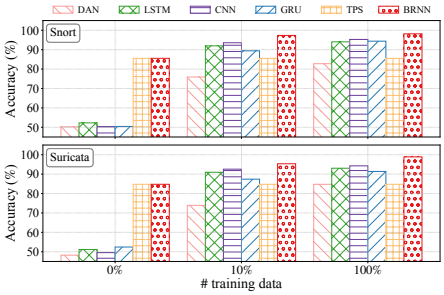


Figure 5: Teacher model accuracy of pattern matching (Snort and Suricata).

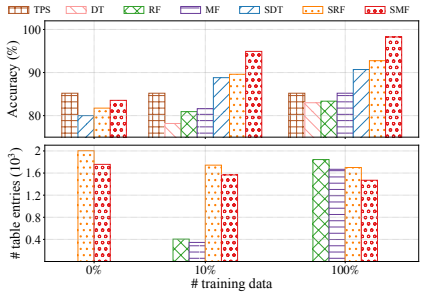


Figure 6: Student model performance of pattern matching (Snort).

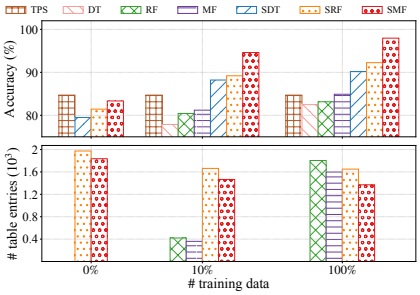


Figure 7: Student model performance of pattern matching (Suricata).

window.

6.4 Data Plane Aware Model Performance

Pattern matching accuracy. Trochilus achieves initial accuracy comparable to TPS. Moreover, through training with labeled data, both the teacher model BRNN and the student model SMF improve accuracy by over 11% compared to TPS.

For pattern modelization, we compare the performance of BRNN and baseline models trained with 0%, 10%, 100% training data. Figure 5 shows the average accuracy of BRNN and each baseline on all categories, and details are shown in the Appendix A. In zero-shot scenarios (i.e., 0% training data), both BRNN and TPS perform the best among all, achieving 85.2% accuracy, demonstrating that BRNN can effectively preserve the full knowledge of original patterns to achieve high initial accuracy. On the contrary, the other baselines only have 50% accuracies since they literally perform random guesses. In few-shot scenarios (i.e., 10% training data), BRNN boosts the accuracy to 96.3%, showing superior accuracy over all baselines. Among the baselines, TPS performs the worst since current pattern matching systems like Snort and Suricata cannot improve accuracy through training like machine learning models do. DAN also performs poorly, only obtaining an accuracy of 74.9% when given 10% training data. The best-performing one among the baselines, i.e., CNN also underperforms BRNN. For full training, BRNN is again the best performing one among all, achieving an accuracy of 98.5%. DAN only achieves 83.7% accuracy. LSTM, CNN, and GRU achieve accuracies of around 93%, which are still inferior to that of BRNN. In a word, BRNN achieves the best accuracy in all scenarios. This justifies the selection of BRNN as the teacher model in the following SSKD.

For data plane model adaption, we compare SMF with baselines trained with 0%, 10%, 100% training data, as shown in Figure 6 and Figure 7, and details are shown in the Appendix A. In the zero-shot scenario, SMF also preserves the expert knowledge of original patterns, achieving an initial accuracy of 83.7%, which is comparable to TPS. DT, RF, and MF are not available in zero-shot scenarios since there is

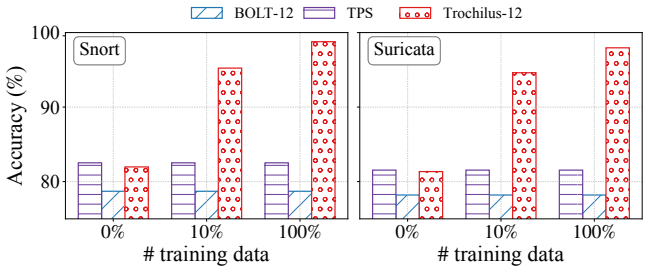


Figure 8: Accuracy comparison of multi-string matching.

no labeled data. In the few-shot scenario, attributing to the SSKD, the accuracy of SMF is more than 12% better than all non-distilled models. Compared with SRF, our solution still presents an advantage of about 5% in accuracy. When trained with full data, SMF achieves over 97.1% accuracy. More importantly, compared to TPS, SMF can achieve an accuracy improvement of over 11% through data training.

We also evaluate # table entries converted from RF, MF, SRF, and SMF. As shown in Figure 6 and 7, # table entries in SMF are less than that in SRF, indicating the lightweightness of the SMF. In the few-shot scenario, SMF and SRF introduce more table entries compared to RF and MF since the SSKD leverages more unlabeled data to train to boost the accuracy. Therefore, SMF is the best choice for the data plane aware model in terms of accuracy and resource efficiency.

Multi-string matching accuracy. Since existing solutions can only perform multi-string matching in the data plane rather than complete pattern matching, we compared the trimmed version of Trochilus, T-MSM-12, with BOLT on multi-string matching. T-MSM-12 consistently achieves higher accuracy than BOLT. Additionally, T-MSM-12 still achieves an initial accuracy comparable to TPS, and through data training, it can improve accuracy by over 10%.

Figure 8 illustrates the accuracy of T-MSM-12, BOLT, and TPS based on the given multi-string pattern set under different training data proportions. In zero-shot scenarios (0% training data), T-MSM-12 achieves similar accuracy with TPS, which is 2% higher than BOLT. Because BOLT matches rules based

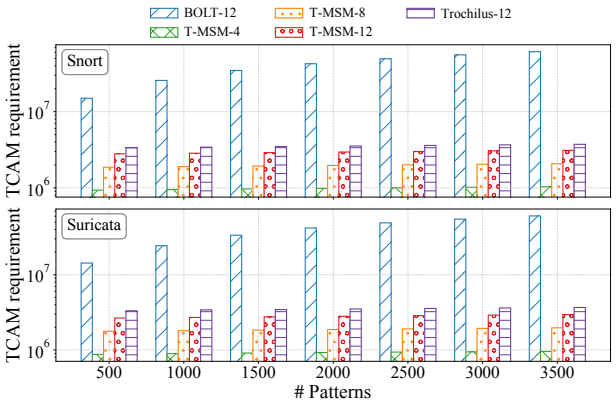


Figure 9: Comparison with BOLT on TCAM requirement. BOLT only supports multi-string matching and uses 12 stages. T-MM-*i* represents Trochilus using *i* stages for multi-string matching. Trochilus-12 represents Trochilus using 12 stages for complete pattern matching.

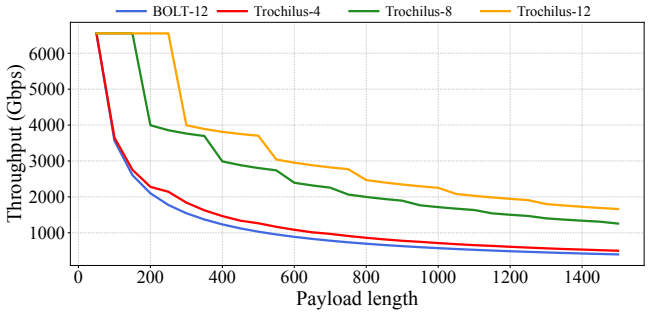


Figure 10: Comparison of throughput. Note that BOLT utilizes 12 MAU stages to achieve maximum throughput and only supports multi-string matching. Trochilus supports full pattern matching.

on serialized state transitions of automaton, it can not handle complex and elongated rules in programmable switches with limited storage and computing resources. In contrast, T-MSM-12 transforms rule matching into model inference, which allows it to handle complex and elongated rules with high effectiveness. In the few-shot and the full training scenarios, T-MSM-12's accuracy is more than 15% higher than BOLT's and more than 10% higher than TPS's. Obviously, like TPS, BOLT is also unable to improve its accuracy through training. Specifically, BOLT cannot expand its state machine converted from multi-string patterns via learning from training data, so its accuracy remains at 82%. Meanwhile, the accuracy of Trochilus can be significantly improved through training attributed to its learning ability.

6.5 Model Representation Performance

We conducted experiments on pattern matching and multi-string matching with varying pattern numbers to evaluate the TCAM requirement. We also conducted experiments with varying payload lengths to evaluate the throughput.

TCAM requirement. Figure 9 shows a comparison of TCAM requirements by Trochilus and BOLT on multi-string matching. T-MSM-4, T-MSM-8, and T-MSM-12 reduce the 97.8%, 95.6%, and 93.4% TCAM requirement of BOLT, respectively. The TCAM requirement of BOLT increases rapidly with the increase of the pattern number, due to the dramatic increase of the state number after the patterns are converted to automatons. Specifically, when the number of patterns reaches 3500, the TCAM requirement of BOLT is 4 times that of when the number of patterns is 500. However, Trochilus only increases about 10% TCAM requirement on average. Although more patterns add complexity to the BRNN model, SSKD can effectively distill the knowledge and add it to the data plane aware model, resulting in limited TCAM requirements. Note that T-MSM-4 achieves a similar throughput with BOLT. However, Trochilus can further improve its throughput using more MAU stages, which will be discussed in Section 6.6.

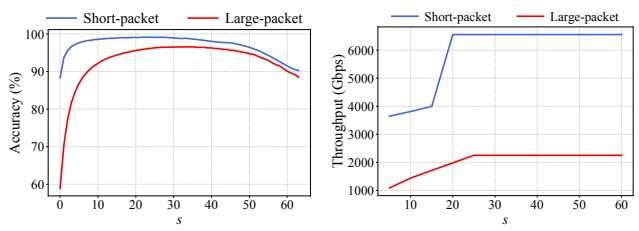
Figure 9 also shows the TCAM requirements of Trochilus for pattern matching under different pattern numbers. We can find that the TCAM requirement for pattern matching is about 15% higher than that for multi-string matching since full patterns have more complex syntax. However, the TCAM requirements of Trochilus on pattern matching are still much less than that of BOLT on multi-string matching. Meanwhile, there is still adequate space for other services after Trochilus has deployed all the patterns in the programmable switches.

Throughput. Trochilus achieves multi-Tbps throughput when processing packets of varying sizes. Restricted by the capability of our traffic generator, we can not fully cover the bandwidth of the Tofino switch used. We use a similar method in BOLT [52] to simulate the theoretical upper limit of the throughput of Trochilus under different pattern sets and workloads. We set the total number of multi-string patterns in Trochilus and BOLT to 3500. For the deployment of Trochilus, we set s to 30 bytes, and use 4, 8, and 12 MAU stages. As shown in Figure 10, with the increasing payload length, both Trochilus and BOLT experience a decrease in throughput due to more recirculations. Nevertheless, even with just 4 stages, Trochilus's single pipeline is able to detect deeper depths than BOLT, thus having higher throughput. Leveraging more MAU stages can improve the inspection depth of a pipeline. Therefore, Trochilus-8 and Trochilus-12 achieve 2.3, and 2.8 times the throughput of BOLT, demonstrating their ability to achieve high throughput in various scenarios.

6.6 Trochilus Deep Dive

We explored the hyperparameters affecting Trochilus in the sliding window and the entry cluster. Note that we extend Trochilus to 12 MAU stages.

Sliding window parameters s and win. The window size



(a) Accuracy (b) Throughput

Figure 11: Analysis of the sliding window step s.

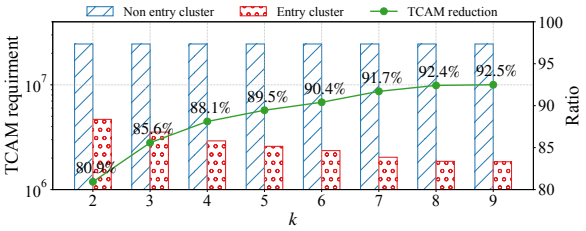


Figure 12: TCAM reduction achieved by entry cluster under different cluster number k.

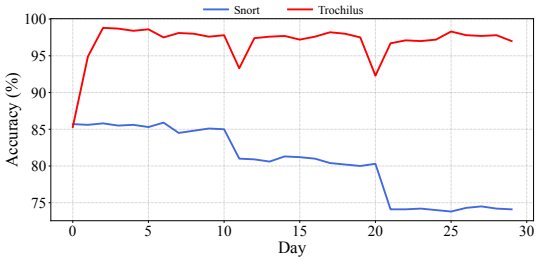


Figure 13: Comparison of accuracy over a 30-day experiment.

win and moving step s are important parameters affecting Trochilus's accuracy and throughput. The larger the *win* is, the larger the packet length that can be inspected by a single window. However, since the key length supported by a ternary match-action table is limited (e.g., no more than 66 bytes in Tofino 1), we fixed the *win* at 64 bytes. Afterward, we apply different s to demonstrate how s affects accuracy and throughput. The short-packet traces and large-packet traces are selected from our dataset, with an average length of 200 bytes and 1000 bytes, respectively. Figure 11(a) shows the impact of different s on accuracy under two traces. As s increases, accuracy first increases and then decreases. The reason behind this is that as s increases, the number of inspection windows decreases, reducing the probability of misclassification. However, it also reduces the probability of a single window capturing patterns, thus decreasing the accuracy.

The throughput of Trochilus over two traces are shown in Figure 11(b). Trochilus can provide high throughput for short packets under different s because they require only a few or even no recirculations which consume negligible bandwidth. However, Trochilus provides relatively low throughput for large packets due to more recirculations. Meanwhile, the PHV resource for storing the extracted payload in the switch restricts the maximum depth that can be resolved in a single pipeline. As a result, the overall throughput for large packets has an upper bound.

Entry cluster parameter k. To further evaluate the performance of the entry cluster, we vary k from 2 to 9 and illustrate the results of the TCAM requirements in Figure 12. When k increases, the TCAM reduction ratio improves from 80.9% to 92.5%, revealing the superior performance of the entry cluster on saving resources. However, the increase of k results in that more tables should be defined and configured in the data plane, introducing a trade-off between the complexity of table management and the TCAM requirement. However, an increase in k results in the need to define and configure more tables in the data plane. Users need to make a trade-off between the complexity of table management and the TCAM requirements.

6.7 Why is Trochilus Easy-to-maintain?

We evaluated Trochilus over multiple weeks of traffic to assess its low maintenance requirements through continuous training. We sampled packet-level traffic for 30 consecutive days at gateways of a large data center and replay the traffic to the Trochilus system and the Snort system. When replaying traffic, we continuously injected different types of zero-day attack traffic on the 10th and 20th days, where the amount of the new attack traffic accounts for 10% of the original attack traffic. On the initial day, Trochilus employed the original pattern set to cold-start since there was no labeled data. At the end of each day, Trochilus was automatically updated based on labeled data collected offline that day.

As shown in Figure 13, Snort exhibits a significant accuracy decline of about 4% and 6% on the 10th and 20th days, respectively. This is because the existing rules are unable to cover new attacks. While Trochilus experiences an accuracy drop of 3% and 4% on the 10th and 20th days, respectively, its accuracy rebounds to about 97% on the 11th and 21st days. Through training with labeled data, Trochilus can automatically mine data patterns and update the transition weights $\mathcal{W}$ of the BRNN. Based on $\mathcal{W}$, we can reconstruct the DFA by mapping the weights into $\{0,1\}$ and convert a trained BRNN back to patterns. Therefore, updating Trochilus parameters through training is equivalent to adding representative new rules. However, Snort requires manual rule extraction from attack traffic to detect new threats, which is both labor-intensive and necessitates special expertise. Conversely, the automated updating process of Trochilus is not dependent on expert knowledge or human efforts, which significantly enhances the system's manageability compared to existing

pattern-matching systems.

In summary, in long-term deployments, traditional pattern-matching systems face challenges in adapting to emerging attacks, while Trochilus sustains a stable accuracy through automatic updates.

7 Discussion

Security analysis. We discuss several adaptive (or white-box) attacks targeting Trochilus. The first type is a throughput exhaustion attack, in which the adversary sends a large volume of jumbo frames (Ethernet packets with payloads exceeding the standard maximum transmission unit (MTU) of 1500 bytes). These oversized packets trigger excessive recirculations within the programmable switches, rapidly depleting their processing capacity. To mitigate such attacks, we propose integrating existing network measurement techniques [14,63] to track both the size and recirculation count of network flows. When statistical indicators surpass predefined thresholds, the system can promptly block the attack traffic and alert network administrators.

The second type is the adversarial attack, where the attacker feeds adversarial inputs to the model to cause model misclassification due to small adversarial perturbations. For example, attackers can add benign bytes to malicious packets to bypass network intrusion detection systems. To tackle this problem, Trochilus can be extended to fuzzy matching to improve its robustness against adversarial attacks. Fuzzy matching (also called approximate pattern matching) is a technique that evaluates the similarity between an observed traffic payload and known malicious patterns, rather than requiring exact matches. Specifically, we can modify our SSKD and use the probability values output by BRNN to train the regression tree instead of the SDT as the fundamental unit of the SMF. We then compare the probability values output by the SMF with a predefined threshold to achieve fuzzy matching.

Scalability analysis. As the number of patterns increases, the required storage may exceed the capacity of a single programmable switch. To address this, the model can be distributed across multiple switches. During packet processing, switches cooperate by forwarding packets to other nodes based on intermediate detection results, thereby forming a complete pattern matching system. We plan to further investigate and optimize this distributed approach in future work. Real-world network applications also require support for encrypted traffic and emerging transport protocols. To handle encrypted traffic, Trochilus can be integrated with existing decryption mechanisms (e.g., [43]). For new protocols, Trochilus can collect relevant packets and update the model accordingly. As data collection is performed offline and model updates are incremental, the system's operation remains unaffected by peak traffic conditions.

8 Related Work

Intelligent model deployment in the data plane. To leverage both the inference ability of the learning model and the line-rate processing ability of the programmable switch, existing methods try to embed learning models into the data plane [55, 57, 65, 66]. However, they need to train the model for a long time using large-scale datasets. The current frameworks and models can not adapt to pattern matching that involves parsing of payloads.

Using programmable switches to accelerate pattern matching. As for the existing methods using programmable switches to accelerate pattern matching, some of them can only support simple multi-string matching [25, 52], while others just use switches for packet forwarding and offload complex pattern matching functions onto additional dedicated hardware (e.g., network processors [22]). Trochilus is the first to implement complex pattern matching entirely in the programmable switches and can be integrated into other data plane hardware (e.g., SmartNICs [36]) with minor changes.

9 Conclusion

In this paper, we propose Trochilus, a learning-enhanced, high-throughput pattern matching framework that leverages programmable data planes to address the pressing challenges of scalability, accuracy, and manageability in modern network pattern matching systems. By integrating model inference with low-cost data plane processing, Trochilus delivers a significant improvement in throughput, achieving multi-Tbps throughput without compromising on pattern matching accuracy. Through innovative techniques like BRNN-based pattern modelization, semi-supervised knowledge distillation, and entry cluster optimization, Trochilus efficiently deploys high-performance models in resource-constrained data planes while ensuring automatic system updates, crucial for real-time adaptation to emerging network traffic and threats. As network bandwidth continues to grow and new applications emerge, Trochilus provides a future-proof pattern matching framework that can adapt and scale to meet the evolving needs of next-generation network infrastructures. Looking ahead, our framework opens several avenues for future research, including the integration of Trochilus with encrypted traffic analysis, SmartNICs, and other hardware-accelerated platforms.

Acknowledgments

This work is supported by the Major Key Project of PCL under grant No. PCL2023A06, Shenzhen R&D Program under grant No. KJZD20230923114059020, and in part by Shenzhen Key Laboratory of Software Defined Networking under Grant ZDSYS20140509172959989.

References

[1] Dpdk-pktgen. https://github.com/Pktgen/Pktgen-DPDK, 2023.

[2] Accton. Towards 800g and 1600g ethernet. https://www.accton.com/Technology-Brief/towards-800g-and-1600g-ethernet/, 2023.

[3] Anonymous. Source code of trochilus. https://anonymous.4open.science/r/Trochilus-RE, 2023.

[4] Leonard E. Baum and Ted Petrie. Statistical Inference for Probabilistic Functions of Finite State Markov Chains. *The Annals of Mathematical Statistics*, 37(6):1554 – 1563, 1966.

[5] Michela Becchi and Srihari Cadambi. Memory-efficient regular expression search using state merging. In *INFOCOM 2007. 26th IEEE International Conference on Computer Communications, Joint Conference of the IEEE Computer and Communications Societies, 6-12 May 2007, Anchorage, Alaska, USA*, pages 1064–1072, 2007.

[6] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: programming protocol-independent packet processors. *Comput. Commun. Rev.*, 44(3):87–95, 2014.

[7] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando A. Mujica, and Mark Horowitz. Forwarding metamorphosis: fast programmable match-action processing in hardware for SDN. In *ACM SIGCOMM 2013 Conference*, pages 99–110, 2013.

[8] Leo Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.

[9] Leo Breiman, Jerome H Friedman, Richard A Olshen, and Charles J Stone. *Classification and regression trees*. Routledge, 2017.

[10] Yeim-Kuan Chang and Ching-Hsuan Shih. A memory efficient pattern matching scheme for regular expressions. In *14th International Conference on Mobile Systems and Pervasive Computing (MobiSPC 2017) / 12th International Conference on Future Networks and Communications (FNC 2017) / Affiliated Workshops, July 24-26, 2017, Leuven, Belgium*, volume 110, pages 250–257, 2017.

[11] Xiang Chen, Hongyan Liu, Dong Zhang, Qun Huang, Haifeng Zhou, Chunming Wu, and Qiang Yang. Empowering ddos attack mitigation with programmable switches. *IEEE Network*, pages 1–7, 2022.

[12] Junyoung Chung, Çaglar Gülçehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *CoRR*, abs/1412.3555, 2014.

[13] Christopher R. Clark and David E. Schimmel. Scalable pattern matching for high speed networks. In *12th IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2004), 20-23 April 2004, Napa, CA, USA, Proceedings*, pages 249–257, 2004.

[14] Graham Cormode and S. Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *J. Algorithms*, 55(1):58–75, 2005.

[15] Seyed Kaveh Fayaz, Yoshiaki Tobioka, Vyas Sekar, and Michael Bailey. Bohatei: Flexible and elastic ddos defense. In *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015*, pages 817–832. USENIX Association, 2015.

[16] The Open Group. Posix. https://pubs.opengroup.org/onlinepubs/9699919799/, 2018.

[17] Sahil Gupta, Devashish Gosain, Minseok Kwon, and Hrishikesh B. Acharya. Deep4r: Deep packet inspection in P4 using packet recirculation. In *IEEE INFOCOM 2023 - IEEE Conference on Computer Communications, New York City, NY, USA, May 17-20, 2023*, pages 1–10. IEEE, 2023.

[18] Charles R Harris, K Jarrod Millman, Stéfan J Van Der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J Smith, et al. Array programming with numpy. *Nature*, 585:357–362, 2020.

[19] Nguyen Phong Hoang, Arian Akhavan Niaki, Jakub Dalek, Jeffrey Knockel, Pellaeon Lin, Bill Marczak, Masashi Crete-Nishihata, Phillipa Gill, and Michalis Polychronakis. How great is the great firewall? measuring china's DNS censorship. In *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, pages 3381–3398. USENIX Association, 2021.

[20] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Comput.*, 9(8):1735–1780, 1997.

[21] John Hopcroft. An nlogn algorithm for minimizing states in a finite automaton. In *Theory of Machines and Computations*, pages 189–196. Academic Press, 1971.

[22] Joel Hypolite, John Sonchack, Shlomo Hershkop, Nathan Dautenhahn, André DeHon, and Jonathan M. Smith. Deepmatch: practical deep packet inspection in the data plane using network processors. In *CoNEXT '20: The 16th International Conference on emerging Networking EXperiments and Technologies, Barcelona, Spain, December, 2020*, pages 336–350, 2020.

[23] Mohit Iyyer and Varun Manjunatha. Deep unordered composition rivals syntactic methods for text classification. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing of the Asian Federation of Natural Language Processing*, pages 1681–1691, 2015.

[24] Nigel Jacob and Carla E. Brodley. Offloading IDS computation to the GPU. In *22nd Annual Computer Security Applications Conference (ACSAC 2006), 11-15 December 2006, Miami Beach, Florida, USA*, pages 371–380, 2006.

[25] Theo Jepsen, Daniel Álvarez, Nate Foster, Changhoon Kim, Jeongkeun Lee, Masoud Moshref, and Robert Soulé. Fast string searching on PISA. In *Proceedings of the 2019 ACM Symposium on SDN Research, SOSR 2019, San Jose, CA, USA, April 3-4, 2019*, pages 21–28, 2019.

[26] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. Netcache: Balancing key-value stores with fast in-network caching. In *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*, pages 121–136, 2017.

[27] Yoon Kim. Convolutional neural networks for sentence classification. *CoRR*, abs/1408.5882, 2014.

[28] ET Labs. Emerging threats open ruleset. https://rules.emergingthreats.net/OPEN_download_instructions.html, 2022.

[29] LinOneTwo. Automata tools. https://pypi.org/project/automata-tools/, 2020.

[30] Hang Lint, Weiwei Lint, Jing Lint, Longlong Zhu, Dong Zhang, and Chunming Wu. P4CTM: compressed traffic pattern matching based on programmable data plane. In *IEEE Symposium on Computers and Communications, ISCC 2023, Gammarth, Tunisia, July 9-12, 2023*, pages 342–347. IEEE, 2023.

[31] Huan Liu. Efficient mapping of range classifier into ternary-cam. In *10th Annual IEEE Symposium on High Performance Interconnects (HOTIC 2002), August 21 - 23, 2002, Stanford, CA, USA*, pages 95–100, 2002.

[32] Zaoxing Liu, Hun Namkung, Georgios Nikolaidis, Jeongkeun Lee, Changhoon Kim, Xin Jin, Vladimir Braverman, Minlan Yu, and Vyas Sekar. Jaqen: A high-performance switch-native approach for detecting and mitigating volumetric ddos attacks with programmable switches. In *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, pages 3829–3846, 2021.

[33] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication, SIGCOMM 2017, Los Angeles, CA, USA, August 21-25, 2017*, pages 15–28, 2017.

[34] Jaehyun Nam, Muhammad Jamshed, Byungkwon Choi, Dongsu Han, and KyoungSoo Park. Haetae: Scaling the performance of network intrusion detection with many-core processors. In *Research in Attacks, Intrusions, and Defenses - 18th International Symposium, RAID 2015, Kyoto, Japan, November 2-4, 2015, Proceedings*, volume 9404, pages 89–110. Springer, 2015.

[35] Jaehyun Nam, Seung Ho Na, Seungwon Shin, and Taejune Park. Reconfigurable regular expression matching architecture for real-time pattern update and payload inspection. *J. Netw. Comput. Appl.*, 208:103507, 2022.

[36] Netronome. Agilio lx smartnics. https://www.netronome.com/products/agilio-lx/, 2022.

[37] ntop. ndpi. https://www:ntop:org, 2023.

[38] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017.

[39] Philip Hazel. Pcre - perl compatible regular expressions. http://www.pcre.org/, 2022.

[40] Michael O. Rabin and Dana S. Scott. Finite automata and their decision problems. *IBM J. Res. Dev.*, 3(2):114–125, 1959.

[41] Martin Roesch. Snort: Lightweight intrusion detection for networks. In *Proceedings of the 13th Conference on Systems Administration (LISA-99), Seattle, WA, USA, November 7-12, 1999*, pages 229–238. USENIX, 1999.

[42] Elaheh Sadredini, Reza Rahimi, Marzieh Lenjani, Mircea Stan, and Kevin Skadron. Impala: Algorithm/architecture co-design for in-memory multi-stride pattern matching. In *IEEE International Symposium on High Performance Computer Architecture, HPCA 2020, San Diego, CA, USA, February 22-26, 2020*, pages 86–98, 2020.

[43] Justine Sherry, Chang Lan, Raluca Ada Popa, and Sylvia Ratnasamy. Blindbox: Deep packet inspection over encrypted traffic. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication, SIGCOMM 2015, London, United Kingdom, August 17-21, 2015*, pages 213–226, 2015.

[44] David Sidler, Zsolt István, Muhsen Owaida, and Gustavo Alonso. Accelerating pattern matching queries in hybrid CPU-FPGA architectures. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, pages 403–415, 2017.

[45] Suricata. Suricata is far more than an ids/ips. https://suricata.io/, 2021.

[46] Thomas Scheibe. Announcing the first cisco 800g nexus switch. https://blogs.cisco.com/datacenter/cisco-nexus-connect-cloud-scale-performance\-and-sustainability, 2022.

[47] Ken Thompson. Programming techniques: Regular expression search algorithm. *Commun. ACM*, 11(6):419–422, jun 1968.

[48] Trustwave. Modsecurity. https://www.modsecurity.org/, 2023.

[49] Giorgos Vasiliadis, Michalis Polychronakis, and Sotiris Ioannidis. Midea: a multi-parallel intrusion detection architecture. In *Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS 2011, Chicago, Illinois, USA, October 17-21, 2011*, pages 297–308, 2011.

[50] Giorgos Vasiliadis, Michalis Polychronakis, and Sotiris Ioannidis. Parallelization and characterization of pattern matching using gpus. In *IISWC 2011, Austin, TX, USA, November 6-8, 2011*, pages 216–225, 2011.

[51] Lucas Vespa, Ning Weng, and Ramaswamy Ramaswamy. MS-DFA: multiple-stride pattern matching for scalable deep packet inspection. *Comput. J.*, 54(2):285–303, 2011.

[52] Shicheng Wang, Menghao Zhang, Guanyu Li, Chang Liu, Ying Liu, Xuya Jia, and Mingwei Xu. Making multi-string pattern matching scalable and cost-efficient with programmable switching asics. In *40th IEEE Conference on Computer Communications, INFOCOM 2021, Vancouver, BC, Canada, May 10-13, 2021*, pages 1–10, 2021.

[53] Xiang Wang, Yang Hong, Harry Chang, KyoungSoo Park, Geoff Langdale, Jiayu Hu, and Heqing Zhu. Hyperscan: A fast multi-pattern regex matcher for modern cpus. In *16th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2019, Boston, MA, February 26-28, 2019*, pages 631–648, 2019.

[54] Xiaofei Wang, Yang Xu, Junchen Jiang, Olga Ormond, Bin Liu, and Xiaojun Wang. Strifa: Stride finite automata for high-speed regular expression matching in network intrusion detection systems. *IEEE Syst. J.*, 7(3):374–384, 2013.

[55] Guorui Xie, Qing Li, Yutao Dong, Guanglin Duan, Yong Jiang, and Jingpu Duan. Mousika: Enable general in-network intelligence in programmable switches by knowledge distillation. In *IEEE INFOCOM 2022 - IEEE Conference on Computer Communications, London, United Kingdom, May 2-5, 2022*, pages 1938–1947, 2022.

[56] Jiarong Xing, Qiao Kang, and Ang Chen. Netwarden: Mitigating network covert channels while preserving performance. In *USENIX Security 2020, August 12-14, 2020*, pages 2039–2056, 2020.

[57] Zhaoqi Xiong and Noa Zilberman. Do switches dream of machine learning?: Toward in-network classification. In *Proceedings of the 18th ACM Workshop on Hot Topics in Networks, HotNets 2019, Princeton, NJ, USA, November 13-15, 2019*, pages 25–33, 2019.

[58] Hao Xu, Harry Chang, Wenjun Zhu, Yang Hong, Geoff Langdale, Kun Qiu, and Jin Zhao. Harry: A scalable simd-based multi-literal pattern matching engine for deep packet inspection. In *IEEE INFOCOM 2023 - IEEE Conference on Computer Communications, New York City, NY, USA, May 17-20, 2023*, pages 1–10. IEEE, 2023.

[59] Liu Yang, Rezwana Karim, Vinod Ganapathy, and Randy Smith. Fast, memory-efficient regular expression matching with nfa-obdds. *Comput. Networks*, 55(15):3376–3393, 2011.

[60] Xiaodong Yu and Michela Becchi. GPU acceleration of regular expression matching for large datasets: exploring the implementation space. In *Computing Frontiers Conference, CF'13, Ischia, Italy, May 14 - 16, 2013*, pages 18:1–18:10, 2013.

[61] Zeek. An open source network security monitoring tool. https://zeek.org/, 2023.

[62] Shuzhuang Zhang, Hao Luo, Binxing Fang, and Xiaochun Yun. Fast and memory-efficient regular expression matching using transition sharing. *IEICE Trans. Inf. Syst.*, 92-D(10):1953–1960, 2009.

[63] Zhengxin Zhang, Qing Li, Guanglin Duan, Dan Zhao, Jingyu Xiao, Guorui Xie, and Yong Jiang. Pontus: Finding waves in data streams. *Proc. ACM Manag. Data*, 1(1):106:1–106:26, 2023.

[64] Zhipeng Zhao, Hugo Sadok, Nirav Atre, James C. Hoe, Vyas Sekar, and Justine Sherry. Achieving 100gbps intrusion prevention on a single server. In *OSDI 2020, Virtual Event, November 4-6, 2020*, pages 1083–1100, 2020.

[65] Changgang Zheng and Noa Zilberman. Planter: seeding trees within switches. In *SIGCOMM '21: ACM SIGCOMM 2021 Conference, Virtual Event, August 23-27, 2021, Poster and Demo Sessions*, pages 12–14, 2021.

[66] Guangmeng Zhou and Zhuotao Liu. An efficient design of intelligent network data plane. In *32nd USENIX Security Symposium (USENIX Security 23). Anaheim, CA: USENIX Association*, 2023.

A supplementary materials

We show supported PCRE syntax in Table 2. The detailed results of pattern modelization and data Plane aware model under different pattern sets are shown in Table 3-6.

Table 2: PCRE syntaxes supported in this paper.

Syntax	Description
α	Matches a single character.
$\alpha\beta$	CONCAT operation. Matches $\alpha\beta$.
$\alpha\|\beta$	OR (|) operation. Matches α or β.
$\alpha*$	Kleene ($*$) star. Matches α zero or more times.
.	Wildcard. Matches any character.
$\alpha+$	PLUS ($+$) operation. Matches α one or more times.
$\hat{}\alpha$	Matches α only appears at the beginning of the string.
$\$\alpha$	Matches α only appears at the ending of the string.
$[\alpha-\beta]$	Character class. The character class uses OR operation to match a character included in the character class.
$\alpha\{\beta,\delta\}$	Range Matching. Matches α subexpression β to δ times.
$\alpha\{\beta,\}$	AtLeast Matching. Matches α subexpression β or more times.
$\alpha\{\beta\}$	Exactly Matching. Matches α subexpression β times.
$\backslash d$	Matches any number, equivalent to $[0-9]$.
$\backslash D$	Matches any non-number.
$\backslash w$	Matches any letter, equivalent to $[a-zA-Z]$.
$\backslash W$	Matches any non-letter.
$\backslash s$	Matches any non-whitespace character.
$\backslash S$	Matches any whitespace character.

Table 3: The results of teacher model on Snort.

	# TD	info	web specific app	malware	sql	exploit
TPS	-	86.14	85.24	85.54	85.84	84.94
LSTM	0%	52.27	51.98	52.86	52.57	51.67
	10%	93.16	88.96	92.56	92.26	92.86
	100%	94.64	93.75	94.43	94.32	94.24
CNN	0%	51.42	48.12	49.04	51.13	51.71
	10%	95.48	92.79	94.86	92.47	92.20
	100%	96.25	95.82	95.58	94.64	93.96
GRU	0%	51.48	52.04	48.93	51.71	48.02
	10%	88.76	87.79	91.27	91.29	87.88
	100%	93.10	95.33	95.94	92.23	95.29
DAN	0%	53.00	49.56	49.86	49.53	49.26
	10%	75.85	75.86	75.56	76.14	76.15
	100%	83.38	83.69	83.39	79.97	83.35
BRNN	0%	84.368	84.967	88.71	84.378	85.268
	10%	98.254	96.164	97.374	96.094	98.814
	100%	99.17	96.69	97.84	97.36	99.75

Table 4: The results of student model on Snort.

	# TD	info	web specific app	malware	sql	exploit
TPS	-	86.14	85.24	85.54	85.84	84.94
DT	0%	52.27	51.98	52.86	52.57	51.67
	10%	93.16	88.96	92.56	92.26	92.86
	100%	81.24	82.14	85.14	80.94	85.44
RF	10%	80.26	70.95	80.56	82.96	80.86
	100%	82.89	84.08	83.43	82.88	83.45
MF	10%	81.04	81.00	84.16	80.76	81.02
	100%	85.78	82.63	85.71	86.09	85.79
SDT	0%	81.21	80.83	77.16	80.54	80.26
	10%	88.27	91.34	87.91	88536	87.97
	100%	90.80	91.88	91.25	89.46	90.07
SRF	0%	84.35	80.66	80.91	81.52	81.24
	10%	90.24	90.83	90.85	88.08	87.99
	100%	93.99	91.65	92.36	93.67	93.08
SMF	0%	85.35	85.02	81.99	83.00	82.33
	10%	96.08	96.34	93.22	92.58	96.32
	100%	98.93	96.73	99.54	99.59	96.44

Table 5: The results of teacher model on Suricata.

	# TD	info	web specific app	malware	sql	exploit
TPS	-	84.56	85.08	85.44	84.55	84.21
LSTM	0%	53.41	50.08	49.40	49.74	53.14
	10%	91.69	88.91	91.39	93.96	94.00
	100%	93.81	90.08	93.35	95.96	95.23
CNN	0%	46.66	50.08	50.42	50.69	49.81
	10%	90.95	90.05	93.53	93.80	94.43
	100%	91.96	96.01	95.32	95.97	95.08
GRU	0%	52.26	51.72	51.38	54.81	52.01
	10%	87.87	87.25	87.30	86.97	87.59
	100%	90.76	90.47	91.06	94.16	90.42
DAN	0%	46.98	47.30	46.73	50.38	49.79
	10%	73.90	73.84	74.13	73.83	73.89
	100%	86.47	86.53	86.17	82.45	82.16
BRNN	0%	84.56	85.08	85.44	84.55	84.21
	10%	93.65	97.96	93.67	97.09	94.32
	100%	99.40	98.31	99.36	98.72	98.34

Table 6: The results of student model on Suricata.

	# TD	info	web specific app	malware	sql	exploit
TPS	-	84.56	85.08	85.44	84.55	84.21
DT	0%	53.41	50.08	49.40	49.74	53.14
	10%	91.69	88.91	91.39	93.96	94.00
	100%	81.24	82.50	82.48	84.68	81.53
RF	10%	80.02	79.79	80.05	82.26	80.01
	100%	82.03	81.17	85.39	84.83	82.31
MF	10%	83.25	83.22	80.10	79.57	79.85
	100%	86.40	87.28	83.32	84.23	83.26
SDT	0%	81.24	78.74	78.48	77.79	81.22
	10%	90.74	87.50	87.58	88.16	87.04
	100%	91.98	89.11	88.80	92.52	88.55
SRF	0%	80.76	80.12	83.21	83.22	79.86
	10%	87.66	91.10	88.00	90.86	88.34
	100%	93.66	91.39	93.96	91.09	91.12
SMF	0%	85.35	82.28	81.40	85.68	81.98
	10%	95.21	95.45	95.17	95.16	92.04
	100%	98.93	95.46	96.70	99.78	98.86